Biopython Tutorial, Cookbook, and API Documentation

Release 1.84

Jeff Chang, Brad Chapman, Iddo Friedberg, Thomas Hamelryck, Michiel de Hoon, Peter Cock, Tiago Antao, Eric Talevich, Bartek Wilczyński, and The Biopython Contributors

Jun 28, 2024

TABLE OF CONTENTS

This is from Biopython 1.84.

Part I

Biopython Tutorial & Cookbook

Named authors: Jeff Chang, Brad Chapman, Iddo Friedberg, Thomas Hamelryck, Michiel de Hoon, Peter Cock, Tiago Antao, Eric Talevich, Bartek Wilczyński.

This is from Biopython 1.84.

CHAPTER

ONE

INTRODUCTION

1.1 What is Biopython?

Biopython is a collection of freely available Python (https://www.python.org) modules for computational molecular biology. Python is an object oriented, interpreted, flexible language that is widely used for scientific computing. Python is easy to learn, has a very clear syntax and can easily be extended with modules written in C, C++ or FORTRAN. Since its inception in 2000 [Chapman2000], Biopython has been continuously developed and maintained by a large group of volunteers worldwide.

The Biopython web site (http://www.biopython.org) provides an online resource for modules, scripts, and web links for developers of Python-based software for bioinformatics use and research. Biopython includes parsers for various bioinformatics file formats (BLAST, Clustalw, FASTA, Genbank, ...), access to online services (NCBI, Expasy, ...), a standard sequence class, sequence alignment and motif analysis tools, clustering algorithms, a module for structural biology, and a module for phylogenetics analysis.

1.2 What can I find in the Biopython package

The main Biopython releases have lots of functionality, including:

- The ability to parse bioinformatics files into Python utilizable data structures, including support for the following formats:
  - Blast output – both from standalone and WWW Blast
  - Clustalw
  - FASTA
  - GenBank
  - PubMed and Medline
  - ExPASy files, like Enzyme and Prosite
  - SCOP, including 'dom' and 'lin' files
  - UniGene
  - SwissProt
- Files in the supported formats can be iterated over record by record or indexed and accessed via a Dictionary interface.
- Code to deal with popular on-line bioinformatics destinations such as:
  - NCBI – Blast, Entrez and PubMed services

  - ExPASy – Swiss-Prot and Prosite entries, as well as Prosite searches
- Interfaces to common bioinformatics programs such as:
  - Standalone Blast from NCBI
  - Clustalw alignment program
  - EMBOSS command line tools
- A standard sequence class that deals with sequences, ids on sequences, and sequence features.
- Tools for performing common operations on sequences, such as translation, transcription and weight calculations.
- Code to perform classification of data using k Nearest Neighbors, Naive Bayes or Support Vector Machines.
- Code for dealing with alignments, including a standard way to create and deal with substitution matrices.
- Code making it easy to split up parallelizable tasks into separate processes.
- GUI-based programs to do basic sequence manipulations, translations, BLASTing, etc.
- Extensive documentation and help with using the modules, including this file, on-line wiki documentation, the web site, and the mailing list.
- Integration with BioSQL, a sequence database schema also supported by the BioPerl and BioJava projects.

We hope this gives you plenty of reasons to download and start using Biopython!

1.3 Installing Biopython

All of the installation information for Biopython was separated from this document to make it easier to keep updated.

The short version is use `pip install biopython`, see the main README file for other options.

1.4 Frequently Asked Questions (FAQ)

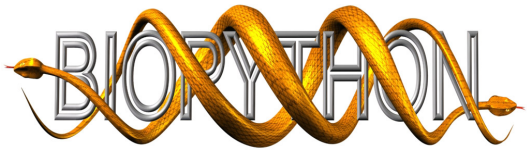
1. *How do I cite Biopython in a scientific publication?*
   Please cite our application note [Cock2009] as the main Biopython reference. In addition, please cite any publications from the following list if appropriate, in particular as a reference for specific modules within Biopython (more information can be found on our website):
   - For the official project announcement: Chapman and Chang, 2000 [Chapman2000];
   - For `Bio.PDB`: Hamelryck and Manderick, 2003 [Hamelryck2003A];
   - For `Bio.Cluster`: De Hoon *et al.*, 2004 [DeHoon2004];
   - For `Bio.Graphics.GenomeDiagram`: Pritchard *et al.*, 2006 [Pritchard2006];
   - For `Bio.Phylo` and `Bio.Phylo.PAML`: Talevich *et al.* 2012 [Talevich2012];
   - For the FASTQ file format as supported in Biopython, BioPerl, BioRuby, BioJava, and EMBOSS: Cock *et al.*, 2010 [Cock2010].
2. *How should I capitalize "Biopython"? Is "BioPython" OK?*
   The correct capitalization is "Biopython", not "BioPython" (even though that would have matched BioPerl, BioJava and BioRuby).
3. *How is the Biopython software licensed?*

Biopython is distributed under the *Biopython License Agreement*. However, since the release of Biopython 1.69, some files are explicitly dual licensed under your choice of the *Biopython License Agreement* or the *BSD 3-Clause License*. This is with the intention of later offering all of Biopython under this dual licensing approach.

4. *What is the Biopython logo and how is it licensed?*

   As of July 2017 and the Biopython 1.70 release, the Biopython logo is a yellow and blue snake forming a double helix above the word "biopython" in lower case. It was designed by Patrick Kunzmann and this logo is dual licensed under your choice of the *Biopython License Agreement* or the *BSD 3-Clause License*.

   Prior to this, the Biopython logo was two yellow snakes forming a double helix around the word "BIOPYTHON", designed by Henrik Vestergaard and Thomas Hamelryck in 2003 as part of an open competition.



5. *Do you have a change-log listing what's new in each release?*

   See the file `NEWS.rst` included with the source code (originally called just `NEWS`), or read the latest NEWS file on GitHub.

6. *What is going wrong with my print commands?*

   As of Biopython 1.77, we only support Python 3, so this tutorial uses the Python 3 style print *function*.

7. *How do I find out what version of Biopython I have installed?*

   Use this:

```
>>> import Bio
>>> print(Bio.__version__)
```

   If the "`import Bio`" line fails, Biopython is not installed. Note that those are double underscores before and after version. If the second line fails, your version is *very* out of date.

   If the version string ends with a plus like "`1.66+`", you don't have an official release, but an old snapshot of the in development code *after* that version was released. This naming was used until June 2016 in the run-up to Biopython 1.68.

   If the version string ends with "`.dev<number>`" like "`1.68.dev0`", again you don't have an official release, but instead a snapshot of the in development code *before* that version was released.

8. *Where is the latest version of this document?*

   If you download a Biopython source code archive, it will include the relevant version in both HTML and PDF formats. The latest published version of this document (updated at each release) is online:

   - http://biopython.org/DIST/docs/tutorial/Tutorial.html
   - http://biopython.org/DIST/docs/tutorial/Tutorial.pdf

9. *What is wrong with my sequence comparisons?*
   There was a major change in Biopython 1.65 making the `Seq` and `MutableSeq` classes (and subclasses) use simple string-based comparison which you can do explicitly with `str(seq1) == str(seq2)`.

   Older versions of Biopython would use instance-based comparison for `Seq` objects which you can do explicitly with `id(seq1) == id(seq2)`.

   If you still need to support old versions of Biopython, use these explicit forms to avoid problems. See Section *Comparing Seq objects*.

10. *What file formats do* `Bio.SeqIO` *and* `Bio.AlignIO` *read and write?*
    Check the built-in docstrings (`from Bio import SeqIO`, then `help(SeqIO)`), or see http://biopython.org/wiki/SeqIO and http://biopython.org/wiki/AlignIO on the wiki for the latest listing.

11. *Why won't the* `Bio.SeqIO` *and* `Bio.AlignIO` *functions* `parse`*,* `read` *and* `write` *take filenames? They insist on handles!*
    You need Biopython 1.54 or later, or just use handles explicitly (see Section *What the heck is a handle?*). It is especially important to remember to close output handles explicitly after writing your data.

12. *Why won't the* `Bio.SeqIO.write()` *and* `Bio.AlignIO.write()` *functions accept a single record or alignment? They insist on a list or iterator!*
    You need Biopython 1.54 or later, or just wrap the item with `[...]` to create a list of one element.

13. *Why doesn't* `str(...)` *give me the full sequence of a* `Seq` *object?*
    You need Biopython 1.45 or later.

14. *Why doesn't* `Bio.Blast` *work with the latest plain text NCBI blast output?*
    The NCBI keep tweaking the plain text output from the BLAST tools, and keeping our parser up to date is/was an ongoing struggle. If you aren't using the latest version of Biopython, you could try upgrading. However, we (and the NCBI) recommend you use the XML output instead, which is designed to be read by a computer program.

15. *Why has my script using* `Bio.Entrez.efetch()` *stopped working?*
    This could be due to NCBI changes in February 2012 introducing EFetch 2.0. First, they changed the default return modes - you probably want to add `retmode="text"` to your call. Second, they are now stricter about how to provide a list of IDs – Biopython 1.59 onwards turns a list into a comma separated string automatically.

16. *Why doesn't* `Bio.Blast.NCBIWWW.qblast()` *give the same results as the NCBI BLAST website?*
    You need to specify the same options – the NCBI often adjust the default settings on the website, and they do not match the QBLAST defaults anymore. Check things like the gap penalties and expectation threshold.

17. *Why can't I add* `SeqRecord` *objects together?*
    You need Biopython 1.53 or later.

18. *Why doesn't* `Bio.SeqIO.index_db()` *work? The module imports fine but there is no ``index_db`` function!*
    You need Biopython 1.57 or later (and a Python with SQLite3 support).

19. *Where is the* `MultipleSeqAlignment` *object? The* `Bio.Align` *module imports fine but this class isn't there!*
    You need Biopython 1.54 or later. Alternatively, the older `Bio.Align.Generic.Alignment` class supports some of its functionality, but using this is now discouraged.

20. *Why can't I run command line tools directly from the application wrappers?*
    You need Biopython 1.55 or later, but these were deprecated in Biopython 1.78. Consider using the Python `subprocess` module directly.

21. *I looked in a directory for code, but I couldn't find the code that does something. Where's it hidden?*
    One thing to know is that we put code in `__init__.py` files. If you are not used to looking for code in this file this can be confusing. The reason we do this is to make the imports easier for users. For instance, instead of

having to do a "repetitive" import like `from Bio.GenBank import GenBank`, you can just use `from Bio import GenBank`.

22. *Why doesn't* `Bio.Fasta` *work?*
    We deprecated the `Bio.Fasta` module in Biopython 1.51 (August 2009) and removed it in Biopython 1.55 (August 2010). There is a brief example showing how to convert old code to use `Bio.SeqIO` instead in the DEPRECATED.rst file.

For more general questions, the Python FAQ pages https://docs.python.org/3/faq/index.html may be useful.

CHAPTER

TWO

QUICK START – WHAT CAN YOU DO WITH BIOPYTHON?

This section is designed to get you started quickly with Biopython, and to give a general overview of what is available and how to use it. All of the examples in this section assume that you have some general working knowledge of Python, and that you have successfully installed Biopython on your system. If you think you need to brush up on your Python, the main Python web site provides quite a bit of free documentation to get started with (https://docs.python.org/3/).

Since much biological work on the computer involves connecting with databases on the internet, some of the examples will also require a working internet connection in order to run.

Now that that is all out of the way, let's get into what we can do with Biopython.

2.1 General overview of what Biopython provides

As mentioned in the introduction, Biopython is a set of libraries to provide the ability to deal with "things" of interest to biologists working on the computer. In general this means that you will need to have at least some programming experience (in Python, of course!) or at least an interest in learning to program. Biopython's job is to make your job easier as a programmer by supplying reusable libraries so that you can focus on answering your specific question of interest, instead of focusing on the internals of parsing a particular file format (of course, if you want to help by writing a parser that doesn't exist and contributing it to Biopython, please go ahead!). So Biopython's job is to make you happy!

One thing to note about Biopython is that it often provides multiple ways of "doing the same thing." Things have improved in recent releases, but this can still be frustrating as in Python there should ideally be one right way to do something. However, this can also be a real benefit because it gives you lots of flexibility and control over the libraries. The tutorial helps to show you the common or easy ways to do things so that you can just make things work. To learn more about the alternative possibilities, look in the Cookbook (Chapter *Cookbook – Cool things to do with it*, this has some cools tricks and tips), and built-in "docstrings" (via the Python help command or `Bio` and `BioSQL`), or ultimately the code itself.

2.2 Working with sequences

Disputably (of course!), the central object in bioinformatics is the sequence. Thus, we'll start with a quick introduction to the Biopython mechanisms for dealing with sequences, the `Seq` object, which we'll discuss in more detail in Chapter *Sequence objects*.

Most of the time when we think about sequences we have in my mind a string of letters like `AGTACACTGGT`. You can create such `Seq` object with this sequence as follows - the >>> represents the Python prompt followed by what you would type in:

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AGTACACTGGT")
```

(continues on next page)

(continued from previous page)

```
>>> my_seq
Seq('AGTACACTGGT')
>>> print(my_seq)
AGTACACTGGT
```

The `Seq` object differs from the Python string in the methods it supports. You can't do this with a plain string:

```
>>> my_seq
Seq('AGTACACTGGT')
>>> my_seq.complement()
Seq('TCATGTGACCA')
>>> my_seq.reverse_complement()
Seq('ACCAGTGTACT')
```

The next most important class is the `SeqRecord` or Sequence Record. This holds a sequence (as a `Seq` object) with additional annotation including an identifier, name and description. The `Bio.SeqIO` module for reading and writing sequence file formats works with `SeqRecord` objects, which will be introduced below and covered in more detail by Chapter *Sequence Input/Output*.

This covers the basic features and uses of the Biopython sequence class. Now that you've got some idea of what it is like to interact with the Biopython libraries, it's time to delve into the fun, fun world of dealing with biological file formats!

2.3 A usage example

Before we jump right into parsers and everything else to do with Biopython, let's set up an example to motivate everything we do and make life more interesting. After all, if there wasn't any biology in this tutorial, why would you want you read it?

Since I love plants, I think we're just going to have to have a plant based example (sorry to all the fans of other organisms out there!). Having just completed a recent trip to our local greenhouse, we've suddenly developed an incredible obsession with Lady Slipper Orchids (if you wonder why, have a look at some Lady Slipper Orchids photos on Flickr, or try a Google Image Search).

Of course, orchids are not only beautiful to look at, they are also extremely interesting for people studying evolution and systematics. So let's suppose we're thinking about writing a funding proposal to do a molecular study of Lady Slipper evolution, and would like to see what kind of research has already been done and how we can add to that.

After a little bit of reading up we discover that the Lady Slipper Orchids are in the Orchidaceae family and the Cypripedioideae sub-family and are made up of 5 genera: *Cypripedium*, *Paphiopedilum*, *Phragmipedium*, *Selenipedium* and *Mexipedium*.

That gives us enough to get started delving for more information. So, let's look at how the Biopython tools can help us. We'll start with sequence parsing in Section *Parsing sequence file formats*, but the orchids will be back later on as well - for example we'll search PubMed for papers about orchids and extract sequence data from GenBank in Chapter *Accessing NCBI's Entrez databases*, extract data from Swiss-Prot from certain orchid proteins in Chapter *Swiss-Prot and ExPASy*, and work with ClustalW multiple sequence alignments of orchid proteins in Section *ClustalW*.

2.4 Parsing sequence file formats

A large part of much bioinformatics work involves dealing with the many types of file formats designed to hold biological data. These files are loaded with interesting biological data, and a special challenge is parsing these files into a format so that you can manipulate them with some kind of programming language. However the task of parsing these files can be frustrated by the fact that the formats can change quite regularly, and that formats may contain small subtleties which can break even the most well designed parsers.

We are now going to briefly introduce the `Bio.SeqIO` module – you can find out more in Chapter *Sequence Input/Output*. We'll start with an online search for our friends, the lady slipper orchids. To keep this introduction simple, we're just using the NCBI website by hand. Let's just take a look through the nucleotide databases at NCBI, using an Entrez online search (https://www.ncbi.nlm.nih.gov/nuccore/?term=Cypripedioideae) for everything mentioning the text Cypripedioideae (this is the subfamily of lady slipper orchids).

When this tutorial was originally written, this search gave us only 94 hits, which we saved as a FASTA formatted text file and as a GenBank formatted text file (files ls_orchid.fasta and ls_orchid.gbk, also included with the Biopython source code under `Doc/examples/`).

If you run the search today, you'll get hundreds of results! When following the tutorial, if you want to see the same list of genes, just download the two files above or copy them from `docs/examples/` in the Biopython source code. In Section *Connecting with biological databases* we will look at how to do a search like this from within Python.

2.4.1 Simple FASTA parsing example

If you open the lady slipper orchids FASTA file ls_orchid.fasta in your favorite text editor, you'll see that the file starts like this:

```
>gi|2765658|emb|Z78533.1|CIZ78533 C.irapeanum 5.8S rRNA gene and ITS1 and ITS2 DNA
CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGGAATAAACGATCGAGTG
AATCCGGAGGACCGGTGTACTCAGCTCACCGGGGGCATTGCTCCCGTGGTGACCCTGATTTGTTGTTGGG
...
```

It contains 94 records, each has a line starting with > (greater-than symbol) followed by the sequence on one or more lines. Now try this in Python:

```
>>> from Bio import SeqIO
>>> for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta"):
...     print(seq_record.id)
...     print(repr(seq_record.seq))
...     print(len(seq_record))
...
```

You should get something like this on your screen:

```
gi|2765658|emb|Z78533.1|CIZ78533
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC')
740
...
gi|2765564|emb|Z78439.1|PBZ78439
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC')
592
```

2.4.2 Simple GenBank parsing example

Now let's load the GenBank file ls_orchid.gbk instead - notice that the code to do this is almost identical to the snippet used above for the FASTA file - the only difference is we change the filename and the format string:

```
>>> from Bio import SeqIO
>>> for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank"):
...     print(seq_record.id)
...     print(repr(seq_record.seq))
...     print(len(seq_record))
...
```

This should give:

```
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC')
740
...
Z78439.1
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC')
592
```

You'll notice that a shorter string has been used as the `seq_record.id` in this case.

2.4.3 I love parsing – please don't stop talking about it!

Biopython has a lot of parsers, and each has its own little special niches based on the sequence format it is parsing and all of that. Chapter *Sequence Input/Output* covers `Bio.SeqIO` in more detail, while Chapter *Sequence alignments* introduces `Bio.Align` for sequence alignments.

While the most popular file formats have parsers integrated into `Bio.SeqIO` and/or `Bio.AlignIO`, for some of the rarer and unloved file formats there is either no parser at all, or an old parser which has not been linked in yet. Please also check the wiki pages http://biopython.org/wiki/SeqIO and http://biopython.org/wiki/AlignIO for the latest information, or ask on the mailing list. The wiki pages should include an up to date list of supported file types, and some additional examples.

The next place to look for information about specific parsers and how to do cool things with them is in the Cookbook (Chapter *Cookbook – Cool things to do with it* of this Tutorial). If you don't find the information you are looking for, please consider helping out your poor overworked documentors and submitting a cookbook entry about it! (once you figure out how to do it, that is!)

2.5 Connecting with biological databases

One of the very common things that you need to do in bioinformatics is extract information from biological databases. It can be quite tedious to access these databases manually, especially if you have a lot of repetitive work to do. Biopython attempts to save you time and energy by making some on-line databases available from Python scripts. Currently, Biopython has code to extract information from the following databases:

- Entrez (and PubMed) from the NCBI – See Chapter *Accessing NCBI's Entrez databases*.
- ExPASy – See Chapter *Swiss-Prot and ExPASy*.
- SCOP – See the `Bio.SCOP.search()` function.

The code in these modules basically makes it easy to write Python code that interact with the CGI scripts on these pages, so that you can get results in an easy to deal with format. In some cases, the results can be tightly integrated with the Biopython parsers to make it even easier to extract information.

2.6 What to do next

Now that you've made it this far, you hopefully have a good understanding of the basics of Biopython and are ready to start using it for doing useful work. The best thing to do now is finish reading this tutorial, and then if you want start snooping around in the source code, and looking at the automatically generated documentation.

Once you get a picture of what you want to do, and what libraries in Biopython will do it, you should take a peak at the Cookbook (Chapter *Cookbook – Cool things to do with it*), which may have example code to do something similar to what you want to do.

If you know what you want to do, but can't figure out how to do it, please feel free to post questions to the main Biopython list (see http://biopython.org/wiki/Mailing_lists). This will not only help us answer your question, it will also allow us to improve the documentation so it can help the next person do what you want to do.

Enjoy the code!

CHAPTER

THREE

SEQUENCE OBJECTS

Biological sequences are arguably the central object in Bioinformatics, and in this chapter we'll introduce the Biopython mechanism for dealing with sequences, the `Seq` object. Chapter *Sequence annotation objects* will introduce the related `SeqRecord` object, which combines the sequence information with any annotation, used again in Chapter *Sequence Input/Output* for Sequence Input/Output.

Sequences are essentially strings of letters like `AGTACACTGGT`, which seems very natural since this is the most common way that sequences are seen in biological file formats.

The most important difference between `Seq` objects and standard Python strings is they have different methods. Although the `Seq` object supports many of the same methods as a plain string, its `translate()` method differs by doing biological translation, and there are also additional biologically relevant methods like `reverse_complement()`.

3.1 Sequences act like strings

In most ways, we can deal with Seq objects as if they were normal Python strings, for example getting the length, or iterating over the elements:

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("GATCG")
>>> for index, letter in enumerate(my_seq):
...     print("%i %s" % (index, letter))
...
0 G
1 A
2 T
3 C
4 G
>>> print(len(my_seq))
5
```

You can access elements of the sequence in the same way as for strings (but remember, Python counts from zero!):

```
>>> print(my_seq[0])  # first letter
G
>>> print(my_seq[2])  # third letter
T
>>> print(my_seq[-1])  # last letter
G
```

The `Seq` object has a `.count()` method, just like a string. Note that this means that like a Python string, this gives a *non-overlapping* count:

```
>>> from Bio.Seq import Seq
>>> "AAAA".count("AA")
2
>>> Seq("AAAA").count("AA")
2
```

For some biological uses, you may actually want an overlapping count (i.e. 3 in this trivial example). When searching for single letters, this makes no difference:

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC")
>>> len(my_seq)
32
>>> my_seq.count("G")
9
>>> 100 * (my_seq.count("G") + my_seq.count("C")) / len(my_seq)
46.875
```

While you could use the above snippet of code to calculate a GC%, note that the `Bio.SeqUtils` module has several GC functions already built. For example:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqUtils import gc_fraction
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC")
>>> gc_fraction(my_seq)
0.46875
```

Note that using the `Bio.SeqUtils.gc_fraction()` function should automatically cope with mixed case sequences and the ambiguous nucleotide S which means G or C.

Also note that just like a normal Python string, the `Seq` object is in some ways "read-only". If you need to edit your sequence, for example simulating a point mutation, look at the Section *MutableSeq objects* below which talks about the `MutableSeq` object.

3.2 Slicing a sequence

A more complicated example, let's get a slice of the sequence:

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC")
>>> my_seq[4:12]
Seq('GATGGGCC')
```

Note that 'Seq' objects follow the usual indexing conventions for Python strings, with the first element of the sequence numbered 0. When you do a slice the first item is included (i.e. 4 in this case) and the last is excluded (12 in this case).

Also like a Python string, you can do slices with a start, stop and *stride* (the step size, which defaults to one). For example, we can get the first, second and third codon positions of this DNA sequence:

```
>>> my_seq[0::3]
Seq('GCTGTAGTAAG')
>>> my_seq[1::3]
Seq('AGGCATGCATC')
```

(continues on next page)

(continued from previous page)

```
>>> my_seq[2::3]
Seq('TAGCTAAGAC')
```

Another stride trick you might have seen with a Python string is the use of a -1 stride to reverse the string. You can do this with a `Seq` object too:

```
>>> my_seq[::-1]
Seq('CGCTAAAAGCTAGGATATATCCGGGTAGCTAG')
```

3.3 Turning Seq objects into strings

If you really do just need a plain string, for example to write to a file, or insert into a database, then this is very easy to get:

```
>>> str(my_seq)
'GATCGATGGGCCTATATAGGATCGAAAATCGC'
```

Since calling `str()` on a `Seq` object returns the full sequence as a string, you often don't actually have to do this conversion explicitly. Python does this automatically in the print function:

```
>>> print(my_seq)
GATCGATGGGCCTATATAGGATCGAAAATCGC
```

You can also use the `Seq` object directly with a `%s` placeholder when using the Python string formatting or interpolation operator (`%`):

```
>>> fasta_format_string = ">Name\n%s\n" % my_seq
>>> print(fasta_format_string)
>Name
GATCGATGGGCCTATATAGGATCGAAAATCGC
```

This line of code constructs a simple FASTA format record (without worrying about line wrapping). Section *The format method* describes a neat way to get a FASTA formatted string from a `SeqRecord` object, while the more general topic of reading and writing FASTA format sequence files is covered in Chapter *Sequence Input/Output*.

3.4 Concatenating or adding sequences

Two `Seq` objects can be concatenated by adding them:

```
>>> from Bio.Seq import Seq
>>> seq1 = Seq("ACGT")
>>> seq2 = Seq("AACCGG")
>>> seq1 + seq2
Seq('ACGTAACCGG')
```

Biopython does not check the sequence contents and will not raise an exception if for example you concatenate a protein sequence and a DNA sequence (which is likely a mistake):

```
>>> from Bio.Seq import Seq
>>> protein_seq = Seq("EVRNAK")
>>> dna_seq = Seq("ACGT")
>>> protein_seq + dna_seq
Seq('EVRNAKACGT')
```

You may often have many sequences to add together, which can be done with a for loop like this:

```
>>> from Bio.Seq import Seq
>>> list_of_seqs = [Seq("ACGT"), Seq("AACC"), Seq("GGTT")]
>>> concatenated = Seq("")
>>> for s in list_of_seqs:
...     concatenated += s
...
>>> concatenated
Seq('ACGTAACCGGTT')
```

Like Python strings, Biopython `Seq` also has a `.join` method:

```
>>> from Bio.Seq import Seq
>>> contigs = [Seq("ATG"), Seq("ATCCCG"), Seq("TTGCA")]
>>> spacer = Seq("N" * 10)
>>> spacer.join(contigs)
Seq('ATGNNNNNNNNNNATCCCGNNNNNNNNNNTTGCA')
```

3.5 Changing case

Python strings have very useful `upper` and `lower` methods for changing the case. For example,

```
>>> from Bio.Seq import Seq
>>> dna_seq = Seq("acgtACGT")
>>> dna_seq
Seq('acgtACGT')
>>> dna_seq.upper()
Seq('ACGTACGT')
>>> dna_seq.lower()
Seq('acgtacgt')
```

These are useful for doing case insensitive matching:

```
>>> "GTAC" in dna_seq
False
>>> "GTAC" in dna_seq.upper()
True
```

3.6 Nucleotide sequences and (reverse) complements

For nucleotide sequences, you can easily obtain the complement or reverse complement of a Seq object using its built-in methods:

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC")
>>> my_seq
Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC')
>>> my_seq.complement()
Seq('CTAGCTACCCGGATATATCCTAGCTTTTAGCG')
>>> my_seq.reverse_complement()
Seq('GCGATTTTCGATCCTATATAGGCCCATCGATC')
```

As mentioned earlier, an easy way to just reverse a Seq object (or a Python string) is slice it with -1 step:

```
>>> my_seq[::-1]
Seq('CGCTAAAAGCTAGGATATATCCGGGTAGCTAG')
```

If you do accidentally end up trying to do something weird like taking the (reverse) complement of a protein sequence, the results are biologically meaningless:

```
>>> from Bio.Seq import Seq
>>> protein_seq = Seq("EVRNAK")
>>> protein_seq.complement()
Seq('EBYNTM')
```

Here the letter "E" is not a valid IUPAC ambiguity code for nucleotides, so was not complemented. However, "V" means "A", "C" or "G" and has complement "B", and so on.

The example in Section *Converting a file of sequences to their reverse complements* combines the Seq object's reverse complement method with `Bio.SeqIO` for sequence input/output.

3.7 Transcription

Before talking about transcription, I want to try to clarify the strand issue. Consider the following (made up) stretch of double stranded DNA which encodes a short peptide:

DNA coding strand (aka Crick strand, strand $+1$)

```
5'  ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG  3'
    ||||||||||||||||||||||||||||||||||||||||
3'  TACCGGTAACATTACCCGGCGACTTTCCCACGGGCTATC  5'
```

DNA template strand (aka Watson strand, strand -1)

Transcription of this DNA sequence produces the following RNA sequence:

```
5'  AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG  3'
```

Single-stranded messenger RNA

The actual biological transcription process works from the template strand, doing a reverse complement (TCAG $\rightarrow$ CUGA) to give the mRNA. However, in Biopython and bioinformatics in general, we typically work directly with the coding strand because this means we can get the mRNA sequence just by switching T $\rightarrow$ U.

Now let's actually get down to doing a transcription in Biopython. First, let's create `Seq` objects for the coding and template DNA strands:

```
>>> from Bio.Seq import Seq
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG')
>>> template_dna = coding_dna.reverse_complement()
>>> template_dna
Seq('CTATCGGGCACCCTTTCAGCGGCCCATTACAATGGCCAT')
```

These should match the figure above - remember by convention nucleotide sequences are normally read from the 5' to 3' direction, while in the figure the template strand is shown reversed.

Now let's transcribe the coding strand into the corresponding mRNA, using the `Seq` object's built-in `transcribe` method:

```
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG')
>>> messenger_rna = coding_dna.transcribe()
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG')
```

As you can see, all this does is to replace T by U.

If you do want to do a true biological transcription starting with the template strand, then this becomes a two-step process:

```
>>> template_dna.reverse_complement().transcribe()
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG')
```

The `Seq` object also includes a back-transcription method for going from the mRNA to the coding strand of the DNA. Again, this is a simple U $\rightarrow$ T substitution:

```
>>> from Bio.Seq import Seq
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG")
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG')
>>> messenger_rna.back_transcribe()
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG')
```

Note: The `Seq` object's `transcribe` and `back_transcribe` methods were added in Biopython 1.49. For older releases you would have to use the `Bio.Seq` module's functions instead, see Section *Working with strings directly*.

3.8 Translation

Sticking with the same example discussed in the transcription section above, now let's translate this mRNA into the corresponding protein sequence - again taking advantage of one of the `Seq` object's biological methods:

```
>>> from Bio.Seq import Seq
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG")
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG')
```

(continues on next page)

(continued from previous page)

```
>>> messenger_rna.translate()
Seq('MAIVMGR*KGAR*')
```

You can also translate directly from the coding strand DNA sequence:

```
>>> from Bio.Seq import Seq
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG')
>>> coding_dna.translate()
Seq('MAIVMGR*KGAR*')
```

You should notice in the above protein sequences that in addition to the end stop character, there is an internal stop as well. This was a deliberate choice of example, as it gives an excuse to talk about some optional arguments, including different translation tables (Genetic Codes).

The translation tables available in Biopython are based on those from the NCBI (see the next section of this tutorial). By default, translation will use the *standard* genetic code (NCBI table id 1). Suppose we are dealing with a mitochondrial sequence. We need to tell the translation function to use the relevant genetic code instead:

```
>>> coding_dna.translate(table="Vertebrate Mitochondrial")
Seq('MAIVMGRWKGAR*')
```

You can also specify the table using the NCBI table number which is shorter, and often included in the feature annotation of GenBank files:

```
>>> coding_dna.translate(table=2)
Seq('MAIVMGRWKGAR*')
```

Now, you may want to translate the nucleotides up to the first in frame stop codon, and then stop (as happens in nature):

```
>>> coding_dna.translate()
Seq('MAIVMGR*KGAR*')
>>> coding_dna.translate(to_stop=True)
Seq('MAIVMGR')
>>> coding_dna.translate(table=2)
Seq('MAIVMGRWKGAR*')
>>> coding_dna.translate(table=2, to_stop=True)
Seq('MAIVMGRWKGAR')
```

Notice that when you use the `to_stop` argument, the stop codon itself is not translated - and the stop symbol is not included at the end of your protein sequence.

You can even specify the stop symbol if you don't like the default asterisk:

```
>>> coding_dna.translate(table=2, stop_symbol="@")
Seq('MAIVMGRWKGAR@')
```

Now, suppose you have a complete coding sequence CDS, which is to say a nucleotide sequence (e.g. mRNA – after any splicing) which is a whole number of codons (i.e. the length is a multiple of three), commences with a start codon, ends with a stop codon, and has no internal in-frame stop codons. In general, given a complete CDS, the default translate method will do what you want (perhaps with the `to_stop` option). However, what if your sequence uses a non-standard start codon? This happens a lot in bacteria – for example the gene yaaX in *E. coli* K12:

```
>>> from Bio.Seq import Seq
>>> gene = Seq(
...     "GTGAAAAAGATGCAATCTATCGTACTCGCACTTTCCCTGGTTCTGGTCGCTCCCATGGCA"
...     "GCACAGGCTGCGGAAATTACGTTAGTCCCGTCAGTAAAATTACAGATAGGCGATCGTGAT"
...     "AATCGTGGCTATTACTGGGATGGAGGTCACTGGCGCGACCACGGCTGGTGGAAACAACAT"
...     "TATGAATGGCGAGGCAATCGCTGGCACCTACACGGACCGCCGCCACCGCCGCGCCACCAT"
...     "AAGAAAGCTCCTCATGATCATCACGGCGGTCATGGTCCAGGCAAACATCACCGCTAA"
... )
>>> gene.translate(table="Bacterial")
Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HR*',
ProteinAlpabet())
>>> gene.translate(table="Bacterial", to_stop=True)
Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HHR')
```

In the bacterial genetic code `GTG` is a valid start codon, and while it does *normally* encode Valine, if used as a start codon it should be translated as methionine. This happens if you tell Biopython your sequence is a complete CDS:

```
>>> gene.translate(table="Bacterial", cds=True)
Seq('MKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HHR')
```

In addition to telling Biopython to translate an alternative start codon as methionine, using this option also makes sure your sequence really is a valid CDS (you'll get an exception if not).

The example in Section *Translating a FASTA file of CDS entries* combines the `Seq` object's translate method with `Bio.SeqIO` for sequence input/output.

3.9 Translation Tables

In the previous sections we talked about the `Seq` object translation method (and mentioned the equivalent function in the `Bio.Seq` module – see Section *Working with strings directly*). Internally these use codon table objects derived from the NCBI information at ftp://ftp.ncbi.nlm.nih.gov/entrez/misc/data/gc.prt, also shown on https://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi in a much more readable layout.

As before, let's just focus on two choices: the Standard translation table, and the translation table for Vertebrate Mitochondrial DNA.

```
>>> from Bio.Data import CodonTable
>>> standard_table = CodonTable.unambiguous_dna_by_name["Standard"]
>>> mito_table = CodonTable.unambiguous_dna_by_name["Vertebrate Mitochondrial"]
```

Alternatively, these tables are labeled with ID numbers 1 and 2, respectively:

```
>>> from Bio.Data import CodonTable
>>> standard_table = CodonTable.unambiguous_dna_by_id[1]
>>> mito_table = CodonTable.unambiguous_dna_by_id[2]
```

You can compare the actual tables visually by printing them:

```
>>> print(standard_table)
Table 1 Standard, SGC0

  |  T      |  C      |  A      |  G      |
--+---------+---------+---------+---------+--
```

(continues on next page)

(continued from previous page)

```
T | TTT F   | TCT S   | TAT Y   | TGT C   | T
T | TTC F   | TCC S   | TAC Y   | TGC C   | C
T | TTA L   | TCA S   | TAA Stop| TGA Stop| A
T | TTG L(s)| TCG S   | TAG Stop| TGG W   | G
--+---------+---------+---------+---------+--
C | CTT L   | CCT P   | CAT H   | CGT R   | T
C | CTC L   | CCC P   | CAC H   | CGC R   | C
C | CTA L   | CCA P   | CAA Q   | CGA R   | A
C | CTG L(s)| CCG P   | CAG Q   | CGG R   | G
--+---------+---------+---------+---------+--
A | ATT I   | ACT T   | AAT N   | AGT S   | T
A | ATC I   | ACC T   | AAC N   | AGC S   | C
A | ATA I   | ACA T   | AAA K   | AGA R   | A
A | ATG M(s)| ACG T   | AAG K   | AGG R   | G
--+---------+---------+---------+---------+--
G | GTT V   | GCT A   | GAT D   | GGT G   | T
G | GTC V   | GCC A   | GAC D   | GGC G   | C
G | GTA V   | GCA A   | GAA E   | GGA G   | A
G | GTG V   | GCG A   | GAG E   | GGG G   | G
--+---------+---------+---------+---------+--
```

and:

```
>>> print(mito_table)
Table 2 Vertebrate Mitochondrial, SGC1

  |  T      |  C      |  A      |  G      |
--+---------+---------+---------+---------+--
T | TTT F   | TCT S   | TAT Y   | TGT C   | T
T | TTC F   | TCC S   | TAC Y   | TGC C   | C
T | TTA L   | TCA S   | TAA Stop| TGA W   | A
T | TTG L   | TCG S   | TAG Stop| TGG W   | G
--+---------+---------+---------+---------+--
C | CTT L   | CCT P   | CAT H   | CGT R   | T
C | CTC L   | CCC P   | CAC H   | CGC R   | C
C | CTA L   | CCA P   | CAA Q   | CGA R   | A
C | CTG L   | CCG P   | CAG Q   | CGG R   | G
--+---------+---------+---------+---------+--
A | ATT I(s)| ACT T   | AAT N   | AGT S   | T
A | ATC I(s)| ACC T   | AAC N   | AGC S   | C
A | ATA M(s)| ACA T   | AAA K   | AGA Stop| A
A | ATG M(s)| ACG T   | AAG K   | AGG Stop| G
--+---------+---------+---------+---------+--
G | GTT V   | GCT A   | GAT D   | GGT G   | T
G | GTC V   | GCC A   | GAC D   | GGC G   | C
G | GTA V   | GCA A   | GAA E   | GGA G   | A
G | GTG V(s)| GCG A   | GAG E   | GGG G   | G
--+---------+---------+---------+---------+--
```

You may find these following properties useful – for example if you are trying to do your own gene finding:

```
>>> mito_table.stop_codons
```

(continues on next page)

(continued from previous page)

```
['TAA', 'TAG', 'AGA', 'AGG']
>>> mito_table.start_codons
['ATT', 'ATC', 'ATA', 'ATG', 'GTG']
>>> mito_table.forward_table["ACG"]
'T'
```

3.10 Comparing Seq objects

Sequence comparison is actually a very complicated topic, and there is no easy way to decide if two sequences are equal. The basic problem is the meaning of the letters in a sequence are context dependent - the letter "A" could be part of a DNA, RNA or protein sequence. Biopython can track the molecule type, so comparing two `Seq` objects could mean considering this too.

Should a DNA fragment "ACG" and an RNA fragment "ACG" be equal? What about the peptide "ACG"? Or the Python string "ACG"? In everyday use, your sequences will generally all be the same type of (all DNA, all RNA, or all protein). Well, as of Biopython 1.65, sequence comparison only looks at the sequence and compares like the Python string.

```
>>> from Bio.Seq import Seq
>>> seq1 = Seq("ACGT")
>>> "ACGT" == seq1
True
>>> seq1 == "ACGT"
True
```

As an extension to this, using sequence objects as keys in a Python dictionary is equivalent to using the sequence as a plain string for the key. See also Section *Turning Seq objects into strings*.

3.11 Sequences with unknown sequence contents

In some cases, the length of a sequence may be known but not the actual letters constituting it. For example, GenBank and EMBL files may represent a genomic DNA sequence only by its config information, without specifying the sequence contents explicitly. Such sequences can be represented by creating a `Seq` object with the argument `None`, followed by the sequence length:

```
>>> from Bio.Seq import Seq
>>> unknown_seq = Seq(None, 10)
```

The `Seq` object thus created has a well-defined length. Any attempt to access the sequence contents, however, will raise an `UndefinedSequenceError`:

```
>>> unknown_seq
Seq(None, length=10)
>>> len(unknown_seq)
10
>>> print(unknown_seq)
Traceback (most recent call last):
...
Bio.Seq.UndefinedSequenceError: Sequence content is undefined
>>>
```

3.12 Sequences with partially defined sequence contents

Sometimes the sequence contents is defined for parts of the sequence only, and undefined elsewhere. For example, the following excerpt of a MAF (Multiple Alignment Format) file shows an alignment of human, chimp, macaque, mouse, rat, dog, and opossum genome sequences:

```
s hg38.chr7     117512683 36 + 159345973 TTGAAAACCTGAATGTGAGAGTCAGTCAAGGATAGT
s panTro4.chr7  119000876 36 + 161824586 TTGAAAACCTGAATGTGAGAGTCACTCAAGGATAGT
s rheMac3.chr3  156330991 36 + 198365852 CTGAAATCCTGAATGTGAGAGTCAATCAAGGATGGT
s mm10.chr6      18207101 36 + 149736546 CTGAAAACCTAAGTAGGAGAATCAACTAAGGATAAT
s rn5.chr4       42326848 36 + 248343840 CTGAAAACCTAAGTAGGAGAGACAGTTAAAGATAAT
s canFam3.chr14  56325207 36 +  60966679 TTGAAAAACTGATTATTAGAGTCAATTAAGGATAGT
s monDom5.chr8  173163865 36 + 312544902 TTAAGAAACTGGAAATGAGGGTTGAATGACAAACTT
```

In each row, the first number indicates the starting position (in zero-based coordinates) of the aligned sequence on the chromosome, followed by the size of the aligned sequence, the strand, the size of the full chromosome, and the aligned sequence.

A `Seq` object representing such a partially defined sequence can be created using a dictionary for the `data` argument, where the keys are the starting coordinates of the known sequence segments, and the values are the corresponding sequence contents. For example, for the first sequence we would use

```
>>> from Bio.Seq import Seq
>>> seq = Seq({117512683: "TTGAAAACCTGAATGTGAGAGTCAGTCAAGGATAGT"}, length=159345973)
```

Extracting a subsequence from a partially define sequence may return a fully defined sequence, an undefined sequence, or a partially defined sequence, depending on the coordinates:

```
>>> seq[1000:1020]
Seq(None, length=20)
>>> seq[117512690:117512700]
Seq('CCTGAATGTG')
>>> seq[117512670:117512690]
Seq({13: 'TTGAAAA'}, length=20)
>>> seq[117512700:]
Seq({0: 'AGAGTCAGTCAAGGATAGT'}, length=41833273)
```

Partially defined sequences can also be created by appending sequences, if at least one of the sequences is partially or fully undefined:

```
>>> seq = Seq("ACGT")
>>> undefined_seq = Seq(None, length=10)
>>> seq + undefined_seq + seq
Seq({0: 'ACGT', 14: 'ACGT'}, length=18)
```

3.13 MutableSeq objects

Just like the normal Python string, the Seq object is "read only", or in Python terminology, immutable. Apart from wanting the Seq object to act like a string, this is also a useful default since in many biological applications you want to ensure you are not changing your sequence data:

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA")
```

Observe what happens if you try to edit the sequence:

```
>>> my_seq[5] = "G"
Traceback (most recent call last):
...
TypeError: 'Seq' object does not support item assignment
```

However, you can convert it into a mutable sequence (a MutableSeq object) and do pretty much anything you want with it:

```
>>> from Bio.Seq import MutableSeq
>>> mutable_seq = MutableSeq(my_seq)
>>> mutable_seq
MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA')
```

Alternatively, you can create a MutableSeq object directly from a string:

```
>>> from Bio.Seq import MutableSeq
>>> mutable_seq = MutableSeq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA")
```

Either way will give you a sequence object which can be changed:

```
>>> mutable_seq
MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA')
>>> mutable_seq[5] = "C"
>>> mutable_seq
MutableSeq('GCCATCGTAATGGGCCGCTGAAAGGGTGCCCGA')
>>> mutable_seq.remove("T")
>>> mutable_seq
MutableSeq('GCCACGTAATGGGCCGCTGAAAGGGTGCCCGA')
>>> mutable_seq.reverse()
>>> mutable_seq
MutableSeq('AGCCCGTGGGAAAGTCGCCGGGTAATGCACCG')
```

Note that the MutableSeq object's reverse() method, like the reverse() method of a Python list, reverses the sequence in place.

An important technical difference between mutable and immutable objects in Python means that you can't use a MutableSeq object as a dictionary key, but you can use a Python string or a Seq object in this way.

Once you have finished editing your a MutableSeq object, it's easy to get back to a read-only Seq object should you need to:

```
>>> from Bio.Seq import Seq
>>> new_seq = Seq(mutable_seq)
```

(continues on next page)

(continued from previous page)

```
>>> new_seq
Seq('AGCCCGTGGGAAAGTCGCCGGGTAATGCACCG')
```

You can also get a string from a `MutableSeq` object just like from a `Seq` object (Section *Turning Seq objects into strings*).

3.14 Finding subsequences

Sequence objects have `find`, `rfind`, `index`, and `rindex` methods that perform the same function as the corresponding methods on plain string objects. The only difference is that the subsequence can be a string (`str`), `bytes`, `bytearray`, `Seq`, or `MutableSeq` object:

```
>>> from Bio.Seq import Seq, MutableSeq
>>> seq = Seq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA")
>>> seq.index("ATGGGCCGC")
9
>>> seq.index(b"ATGGGCCGC")
9
>>> seq.index(bytearray(b"ATGGGCCGC"))
9
>>> seq.index(Seq("ATGGGCCGC"))
9
>>> seq.index(MutableSeq("ATGGGCCGC"))
9
```

A `ValueError` is raised if the subsequence is not found:

```
>>> seq.index("ACTG")
Traceback (most recent call last):
...
ValueError: ...
```

while the `find` method returns -1 if the subsequence is not found:

```
>>> seq.find("ACTG")
-1
```

The methods `rfind` and `rindex` search for the subsequence starting from the right hand side of the sequence:

```
>>> seq.find("CC")
1
>>> seq.rfind("CC")
29
```

Use the `search` method to search for multiple subsequences at the same time. This method returns an iterator:

```
>>> for index, sub in seq.search(["CC", "GGG", "CC"]):
...     print(index, sub)
...
1 CC
11 GGG
```

(continues on next page)

(continued from previous page)

```
14 CC
23 GGG
28 CC
29 CC
```

The `search` method also takes plain strings, `bytes`, `bytearray`, `Seq`, and `MutableSeq` objects as subsequences; identical subsequences are reported only once, as in the example above.

3.15 Working with strings directly

To close this chapter, for those you who *really* don't want to use the sequence objects (or who prefer a functional programming style to an object orientated one), there are module level functions in `Bio.Seq` will accept plain Python strings, `Seq` objects or `MutableSeq` objects:

```
>>> from Bio.Seq import reverse_complement, transcribe, back_transcribe, translate
>>> my_string = "GCTGTTATGGGTCGTTGGAAGGGTGGTCGTGCTGCTGGTTAG"
>>> reverse_complement(my_string)
'CTAACCAGCAGCACGACCACCCTTCCAACGACCCATAACAGC'
>>> transcribe(my_string)
'GCUGUUAUGGGUCGUUGGAAGGGUGGUCGUGCUGCUGGUUAG'
>>> back_transcribe(my_string)
'GCTGTTATGGGTCGTTGGAAGGGTGGTCGTGCTGCTGGTTAG'
>>> translate(my_string)
'AVMGRWKGGRAAG*'
```

You are, however, encouraged to work with `Seq` objects by default.

CHAPTER

FOUR

SEQUENCE ANNOTATION OBJECTS

Chapter *Sequence objects* introduced the sequence classes. Immediately "above" the `Seq` class is the Sequence Record or `SeqRecord` class, defined in the `Bio.SeqRecord` module. This class allows higher level features such as identifiers and features (as `SeqFeature` objects) to be associated with the sequence, and is used throughout the sequence input/output interface `Bio.SeqIO` described fully in Chapter *Sequence Input/Output*.

If you are only going to be working with simple data like FASTA files, you can probably skip this chapter for now. If on the other hand you are going to be using richly annotated sequence data, say from GenBank or EMBL files, this information is quite important.

While this chapter should cover most things to do with the `SeqRecord` and `SeqFeature` objects in this chapter, you may also want to read the `SeqRecord` wiki page (http://biopython.org/wiki/SeqRecord), and the built-in documentation (*`Bio.SeqRecord`* and *`Bio.SeqFeature`*):

```
>>> from Bio.SeqRecord import SeqRecord
>>> help(SeqRecord)
```

4.1 The SeqRecord object

The `SeqRecord` (Sequence Record) class is defined in the `Bio.SeqRecord` module. This class allows higher level features such as identifiers and features to be associated with a sequence (see Chapter *Sequence objects*), and is the basic data type for the `Bio.SeqIO` sequence input/output interface (see Chapter *Sequence Input/Output*).

The `SeqRecord` class itself is quite simple, and offers the following information as attributes:

.seq
: The sequence itself, typically a `Seq` object.

.id
: The primary ID used to identify the sequence – a string. In most cases this is something like an accession number.

.name
: A "common" name/id for the sequence – a string. In some cases this will be the same as the accession number, but it could also be a clone name. I think of this as being analogous to the LOCUS id in a GenBank record.

.description
: A human readable description or expressive name for the sequence – a string.

.letter_annotations
: Holds per-letter-annotations using a (restricted) dictionary of additional information about the letters in the sequence. The keys are the name of the information, and the information is contained in the value as a Python sequence (i.e. a list, tuple or string) with the same length as the sequence itself. This is often used for quality scores (e.g. Section *Simple quality filtering for FASTQ files*) or secondary structure information (e.g. from Stockholm/PFAM alignment files).

.annotations
A dictionary of additional information about the sequence. The keys are the name of the information, and the information is contained in the value. This allows the addition of more "unstructured" information to the sequence.

.features
A list of `SeqFeature` objects with more structured information about the features on a sequence (e.g. position of genes on a genome, or domains on a protein sequence). The structure of sequence features is described below in Section *Feature, location and position objects*.

.dbxrefs
A list of database cross-references as strings.

4.2 Creating a SeqRecord

Using a `SeqRecord` object is not very complicated, since all of the information is presented as attributes of the class. Usually you won't create a `SeqRecord` "by hand", but instead use `Bio.SeqIO` to read in a sequence file for you (see Chapter *Sequence Input/Output* and the examples below). However, creating `SeqRecord` can be quite simple.

4.2.1 SeqRecord objects from scratch

To create a `SeqRecord` at a minimum you just need a `Seq` object:

```
>>> from Bio.Seq import Seq
>>> simple_seq = Seq("GATC")
>>> from Bio.SeqRecord import SeqRecord
>>> simple_seq_r = SeqRecord(simple_seq)
```

Additionally, you can also pass the id, name and description to the initialization function, but if not they will be set as strings indicating they are unknown, and can be modified subsequently:

```
>>> simple_seq_r.id
'<unknown id>'
>>> simple_seq_r.id = "AC12345"
>>> simple_seq_r.description = "Made up sequence I wish I could write a paper about"
>>> print(simple_seq_r.description)
Made up sequence I wish I could write a paper about
>>> simple_seq_r.seq
Seq('GATC')
```

Including an identifier is very important if you want to output your `SeqRecord` to a file. You would normally include this when creating the object:

```
>>> from Bio.Seq import Seq
>>> simple_seq = Seq("GATC")
>>> from Bio.SeqRecord import SeqRecord
>>> simple_seq_r = SeqRecord(simple_seq, id="AC12345")
```

As mentioned above, the `SeqRecord` has an dictionary attribute `annotations`. This is used for any miscellaneous annotations that doesn't fit under one of the other more specific attributes. Adding annotations is easy, and just involves dealing directly with the annotation dictionary:

```
>>> simple_seq_r.annotations["evidence"] = "None. I just made it up."
>>> print(simple_seq_r.annotations)
{'evidence': 'None. I just made it up.'}
>>> print(simple_seq_r.annotations["evidence"])
None. I just made it up.
```

Working with per-letter-annotations is similar, `letter_annotations` is a dictionary like attribute which will let you assign any Python sequence (i.e. a string, list or tuple) which has the same length as the sequence:

```
>>> simple_seq_r.letter_annotations["phred_quality"] = [40, 40, 38, 30]
>>> print(simple_seq_r.letter_annotations)
{'phred_quality': [40, 40, 38, 30]}
>>> print(simple_seq_r.letter_annotations["phred_quality"])
[40, 40, 38, 30]
```

The `dbxrefs` and `features` attributes are just Python lists, and should be used to store strings and `SeqFeature` objects (discussed later in this chapter) respectively.

4.2.2 SeqRecord objects from FASTA files

This example uses a fairly large FASTA file containing the whole sequence for *Yersinia pestis biovar Microtus* str. 91001 plasmid pPCP1, originally downloaded from the NCBI. This file is included with the Biopython unit tests under the GenBank folder, or online NC_005816.fna from our website.

The file starts like this - and you can check there is only one record present (i.e. only one line starting with a greater than symbol):

```
>gi|45478711|ref|NC_005816.1| Yersinia pestis biovar Microtus ... pPCP1, complete␣
↪sequence
TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGGGGGTAATCTGCTCTCC
...
```

Back in Chapter *Quick Start – What can you do with Biopython?* you will have seen the function `Bio.SeqIO.parse(...)` used to loop over all the records in a file as `SeqRecord` objects. The `Bio.SeqIO` module has a sister function for use on files which contain just one record which we'll use here (see Chapter *Sequence Input/Output* for details):

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.fna", "fasta")
>>> record
SeqRecord(seq=Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG'), id=
↪'gi|45478711|ref|NC_005816.1|', name='gi|45478711|ref|NC_005816.1|', description=
↪'gi|45478711|ref|NC_005816.1| Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1,
↪ complete sequence', dbxrefs=[])
```

Now, let's have a look at the key attributes of this `SeqRecord` individually – starting with the `seq` attribute which gives you a `Seq` object:

```
>>> record.seq
Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG')
```

Next, the identifiers and description:

```
>>> record.id
'gi|45478711|ref|NC_005816.1|'
>>> record.name
'gi|45478711|ref|NC_005816.1|'
>>> record.description
'gi|45478711|ref|NC_005816.1| Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1,␣
↪complete sequence'
```

As you can see above, the first word of the FASTA record's title line (after removing the greater than symbol) is used for both the `id` and `name` attributes. The whole title line (after removing the greater than symbol) is used for the record description. This is deliberate, partly for backwards compatibility reasons, but it also makes sense if you have a FASTA file like this:

```
>Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1
TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGGGGGTAATCTGCTCTCC
...
```

Note that none of the other annotation attributes get populated when reading a FASTA file:

```
>>> record.dbxrefs
[]
>>> record.annotations
{}
>>> record.letter_annotations
{}
>>> record.features
[]
```

In this case our example FASTA file was from the NCBI, and they have a fairly well defined set of conventions for formatting their FASTA lines. This means it would be possible to parse this information and extract the GI number and accession for example. However, FASTA files from other sources vary, so this isn't possible in general.

4.2.3 SeqRecord objects from GenBank files

As in the previous example, we're going to look at the whole sequence for *Yersinia pestis biovar Microtus* str. 91001 plasmid pPCP1, originally downloaded from the NCBI, but this time as a GenBank file. Again, this file is included with the Biopython unit tests under the GenBank folder, or online NC_005816.gb from our website.

This file contains a single record (i.e. only one LOCUS line) and starts:

```
LOCUS       NC_005816               9609 bp    DNA     circular BCT 21-JUL-2008
DEFINITION  Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete
            sequence.
ACCESSION   NC_005816
VERSION     NC_005816.1  GI:45478711
PROJECT     GenomeProject:10638
...
```

Again, we'll use `Bio.SeqIO` to read this file in, and the code is almost identical to that for used above for the FASTA file (see Chapter *Sequence Input/Output* for details):

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.gb", "genbank")
```

(continues on next page)

(continued from previous page)

```
>>> record
SeqRecord(seq=Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG'), id=
↪'NC_005816.1', name='NC_005816', description='Yersinia pestis biovar Microtus str.␣
↪91001 plasmid pPCP1, complete sequence', dbxrefs=['Project:58037'])
```

```
>>> record.seq
Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG')
```

The `name` comes from the LOCUS line, while the `id` includes the version suffix. The description comes from the DEFINITION line:

```
>>> record.id
'NC_005816.1'
>>> record.name
'NC_005816'
>>> record.description
'Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence'
```

GenBank files don't have any per-letter annotations:

```
>>> record.letter_annotations
{}
```

Most of the annotations information gets recorded in the `annotations` dictionary, for example:

```
>>> len(record.annotations)
13
>>> record.annotations["source"]
'Yersinia pestis biovar Microtus str. 91001'
```

The `dbxrefs` list gets populated from any PROJECT or DBLINK lines:

```
>>> record.dbxrefs
['Project:58037']
```

Finally, and perhaps most interestingly, all the entries in the features table (e.g. the genes or CDS features) get recorded as `SeqFeature` objects in the `features` list.

```
>>> len(record.features)
41
```

We'll talk about `SeqFeature` objects next, in Section *Feature, location and position objects*.

4.3 Feature, location and position objects

4.3.1 SeqFeature objects

Sequence features are an essential part of describing a sequence. Once you get beyond the sequence itself, you need some way to organize and easily get at the more "abstract" information that is known about the sequence. While it is probably impossible to develop a general sequence feature class that will cover everything, the Biopython `SeqFeature` class attempts to encapsulate as much of the information about the sequence as possible. The design is heavily based

on the GenBank/EMBL feature tables, so if you understand how they look, you'll probably have an easier time grasping the structure of the Biopython classes.

The key idea about each `SeqFeature` object is to describe a region on a parent sequence, typically a `SeqRecord` object. That region is described with a location object, typically a range between two positions (see Section *Positions and locations* below).

The `SeqFeature` class has a number of attributes, so first we'll list them and their general features, and then later in the chapter work through examples to show how this applies to a real life example. The attributes of a SeqFeature are:

.type
: This is a textual description of the type of feature (for instance, this will be something like 'CDS' or 'gene').

.location
: The location of the `SeqFeature` on the sequence that you are dealing with, see Section *Positions and locations* below. The `SeqFeature` delegates much of its functionality to the location object, and includes a number of shortcut attributes for properties of the location:

  .ref
  : shorthand for `.location.ref` – any (different) reference sequence the location is referring to. Usually just None.

  .ref_db
  : shorthand for `.location.ref_db` – specifies the database any identifier in `.ref` refers to. Usually just None.

  .strand
  : shorthand for `.location.strand` – the strand on the sequence that the feature is located on. For double stranded nucleotide sequence this may either be 1 for the top strand, -1 for the bottom strand, 0 if the strand is important but is unknown, or `None` if it doesn't matter. This is None for proteins, or single stranded sequences.

.qualifiers
: This is a Python dictionary of additional information about the feature. The key is some kind of terse one-word description of what the information contained in the value is about, and the value is the actual information. For example, a common key for a qualifier might be "evidence" and the value might be "computational (non-experimental)." This is just a way to let the person who is looking at the feature know that it has not be experimentally (i. e. in a wet lab) confirmed. Note that other the value will be a list of strings (even when there is only one string). This is a reflection of the feature tables in GenBank/EMBL files.

.sub_features
: This used to be used to represent features with complicated locations like 'joins' in GenBank/EMBL files. This has been deprecated with the introduction of the `CompoundLocation` object, and should now be ignored.

4.3.2 Positions and locations

The key idea about each `SeqFeature` object is to describe a region on a parent sequence, for which we use a location object, typically describing a range between two positions. Two try to clarify the terminology we're using:

position
: This refers to a single position on a sequence, which may be fuzzy or not. For instance, 5, 20, `<100` and `>200` are all positions.

location
: A location is region of sequence bounded by some positions. For instance `5..20` (i. e. 5 to 20) is a location.

I just mention this because sometimes I get confused between the two.

SimpleLocation object

Unless you work with eukaryotic genes, most `SeqFeature` locations are extremely simple - you just need start and end coordinates and a strand. That's essentially all the basic `SimpleLocation` object does.

In practice of course, things can be more complicated. First of all we have to handle compound locations made up of several regions. Secondly, the positions themselves may be fuzzy (inexact).

CompoundLocation object

Biopython 1.62 introduced the `CompoundLocation` as part of a restructuring of how complex locations made up of multiple regions are represented. The main usage is for handling 'join' locations in EMBL/GenBank files.

Fuzzy Positions

So far we've only used simple positions. One complication in dealing with feature locations comes in the positions themselves. In biology many times things aren't entirely certain (as much as us wet lab biologists try to make them certain!). For instance, you might do a dinucleotide priming experiment and discover that the start of mRNA transcript starts at one of two sites. This is very useful information, but the complication comes in how to represent this as a position. To help us deal with this, we have the concept of fuzzy positions. Basically there are several types of fuzzy positions, so we have five classes to deal with them:

ExactPosition
: As its name suggests, this class represents a position which is specified as exact along the sequence. This is represented as just a number, and you can get the position by looking at the `position` attribute of the object.

BeforePosition
: This class represents a fuzzy position that occurs prior to some specified site. In GenBank/EMBL notation, this is represented as something like `<13`, signifying that the real position is located somewhere less than 13. To get the specified upper boundary, look at the `position` attribute of the object.

AfterPosition
: Contrary to `BeforePosition`, this class represents a position that occurs after some specified site. This is represented in GenBank as `>13`, and like `BeforePosition`, you get the boundary number by looking at the `position` attribute of the object.

WithinPosition
: Occasionally used for GenBank/EMBL locations, this class models a position which occurs somewhere between two specified nucleotides. In GenBank/EMBL notation, this would be represented as `(1.5)`, to represent that the position is somewhere within the range 1 to 5.

OneOfPosition
: Occasionally used for GenBank/EMBL locations, this class deals with a position where several possible values exist, for instance you could use this if the start codon was unclear and there where two candidates for the start of the gene. Alternatively, that might be handled explicitly as two related gene features.

UnknownPosition
: This class deals with a position of unknown location. This is not used in GenBank/EMBL, but corresponds to the '?' feature coordinate used in UniProt.

Here's an example where we create a location with fuzzy end points:

```
>>> from Bio import SeqFeature
>>> start_pos = SeqFeature.AfterPosition(5)
>>> end_pos = SeqFeature.BetweenPosition(9, left=8, right=9)
>>> my_location = SeqFeature.SimpleLocation(start_pos, end_pos)
```

Note that the details of some of the fuzzy-locations changed in Biopython 1.59, in particular for BetweenPosition and WithinPosition you must now make it explicit which integer position should be used for slicing etc. For a start position this is generally the lower (left) value, while for an end position this would generally be the higher (right) value.

If you print out a `SimpleLocation` object, you can get a nice representation of the information:

```
>>> print(my_location)
[>5:(8^9)]
```

We can access the fuzzy start and end positions using the start and end attributes of the location:

```
>>> my_location.start
AfterPosition(5)
>>> print(my_location.start)
>5
>>> my_location.end
BetweenPosition(9, left=8, right=9)
>>> print(my_location.end)
(8^9)
```

If you don't want to deal with fuzzy positions and just want numbers, they are actually subclasses of integers so should work like integers:

```
>>> int(my_location.start)
5
>>> int(my_location.end)
9
```

Similarly, to make it easy to create a position without worrying about fuzzy positions, you can just pass in numbers to the `FeaturePosition` constructors, and you'll get back out `ExactPosition` objects:

```
>>> exact_location = SeqFeature.SimpleLocation(5, 9)
>>> print(exact_location)
[5:9]
>>> exact_location.start
ExactPosition(5)
>>> int(exact_location.start)
5
```

That is most of the nitty gritty about dealing with fuzzy positions in Biopython. It has been designed so that dealing with fuzziness is not that much more complicated than dealing with exact positions, and hopefully you find that true!

Location testing

You can use the Python keyword `in` with a `SeqFeature` or location object to see if the base/residue for a parent coordinate is within the feature/location or not.

For example, suppose you have a SNP of interest and you want to know which features this SNP is within, and lets suppose this SNP is at index 4350 (Python counting!). Here is a simple brute force solution where we just check all the features one by one in a loop:

```
>>> from Bio import SeqIO
>>> my_snp = 4350
>>> record = SeqIO.read("NC_005816.gb", "genbank")
```

(continues on next page)

(continued from previous page)

```
>>> for feature in record.features:
...     if my_snp in feature:
...         print("%s %s" % (feature.type, feature.qualifiers.get("db_xref")))
...
source ['taxon:229193']
gene ['GeneID:2767712']
CDS ['GI:45478716', 'GeneID:2767712']
```

Note that gene and CDS features from GenBank or EMBL files defined with joins are the union of the exons – they do not cover any introns.

4.3.3 Sequence described by a feature or location

A `SeqFeature` or location object doesn't directly contain a sequence, instead the location (see Section *Positions and locations*) describes how to get this from the parent sequence. For example consider a (short) gene sequence with location `5:18` on the reverse strand, which in GenBank/EMBL notation using 1-based counting would be `complement(6..18)`, like this:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqFeature import SeqFeature, SimpleLocation
>>> seq = Seq("ACCGAGACGGCAAAGGCTAGCATAGGTATGAGACTTCCTTCCTGCCAGTGCTGAGGAACTGGGAGCCTAC")
>>> feature = SeqFeature(SimpleLocation(5, 18, strand=-1), type="gene")
```

You could take the parent sequence, slice it to extract `5:18`, and then take the reverse complement. The feature location's start and end are integer-like so this works:

```
>>> feature_seq = seq[feature.location.start : feature.location.end].reverse_complement()
>>> print(feature_seq)
AGCCTTTGCCGTC
```

This is a simple example so this isn't too bad – however once you have to deal with compound features (joins) this is rather messy. Instead, the `SeqFeature` object has an `extract` method to take care of all this (and since Biopython 1.78 can handle trans-splicing by supplying a dictionary of referenced sequences):

```
>>> feature_seq = feature.extract(seq)
>>> print(feature_seq)
AGCCTTTGCCGTC
```

The length of a `SeqFeature` or location matches that of the region of sequence it describes.

```
>>> print(len(feature_seq))
13
>>> print(len(feature))
13
>>> print(len(feature.location))
13
```

For `SimpleLocation` objects the length is just the difference between the start and end positions. However, for a `CompoundLocation` the length is the sum of the constituent regions.

4.4 Comparison

The `SeqRecord` objects can be very complex, but here's a simple example:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> record1 = SeqRecord(Seq("ACGT"), id="test")
>>> record2 = SeqRecord(Seq("ACGT"), id="test")
```

What happens when you try to compare these "identical" records?

```
>>> record1 == record2
```

Perhaps surprisingly older versions of Biopython would use Python's default object comparison for the `SeqRecord`, meaning `record1 == record2` would only return `True` if these variables pointed at the same object in memory. In this example, `record1 == record2` would have returned `False` here!

```
>>> record1 == record2  # on old versions of Biopython!
False
```

As of Biopython 1.67, `SeqRecord` comparison like `record1 == record2` will instead raise an explicit error to avoid people being caught out by this:

```
>>> record1 == record2
Traceback (most recent call last):
...
NotImplementedError: SeqRecord comparison is deliberately not implemented. Explicitly␣
↪compare the attributes of interest.
```

Instead you should check the attributes you are interested in, for example the identifier and the sequence:

```
>>> record1.id == record2.id
True
>>> record1.seq == record2.seq
True
```

Beware that comparing complex objects quickly gets complicated (see also Section *Comparing Seq objects*).

4.5 References

Another common annotation related to a sequence is a reference to a journal or other published work dealing with the sequence. We have a fairly simple way of representing a Reference in Biopython – we have a `Bio.SeqFeature.Reference` class that stores the relevant information about a reference as attributes of an object.

The attributes include things that you would expect to see in a reference like `journal`, `title` and `authors`. Additionally, it also can hold the `medline_id` and `pubmed_id` and a `comment` about the reference. These are all accessed simply as attributes of the object.

A reference also has a `location` object so that it can specify a particular location on the sequence that the reference refers to. For instance, you might have a journal that is dealing with a particular gene located on a BAC, and want to specify that it only refers to this position exactly. The `location` is a potentially fuzzy location, as described in section *Positions and locations*.

Any reference objects are stored as a list in the `SeqRecord` object's `annotations` dictionary under the key "references". That's all there is too it. References are meant to be easy to deal with, and hopefully general enough to cover lots of usage cases.

4.6 The format method

The `format()` method of the `SeqRecord` class gives a string containing your record formatted using one of the output file formats supported by `Bio.SeqIO`, such as FASTA:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> record = SeqRecord(
...     Seq(
...         "MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQALFGD"
...         "GAGAVIVGSDPDLSVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKDVPGLISK"
...         "NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRATREVLSEYGNM"
...         "SSAC"
...     ),
...     id="gi|14150838|gb|AAK54648.1|AF376133_1",
...     description="chalcone synthase [Cucumis sativus]",
... )
>>> print(record.format("fasta"))
```

which should give:

```
>gi|14150838|gb|AAK54648.1|AF376133_1 chalcone synthase [Cucumis sativus]
MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQALFGD
GAGAVIVGSDPDLSVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKDVPGLISK
NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRATREVLSEYGNM
SSAC
```

This `format` method takes a single mandatory argument, a lower case string which is supported by `Bio.SeqIO` as an output format (see Chapter *Sequence Input/Output*). However, some of the file formats `Bio.SeqIO` can write to *require* more than one record (typically the case for multiple sequence alignment formats), and thus won't work via this `format()` method. See also Section *Getting your SeqRecord objects as formatted strings*.

4.7 Slicing a SeqRecord

You can slice a `SeqRecord`, to give you a new `SeqRecord` covering just part of the sequence. What is important here is that any per-letter annotations are also sliced, and any features which fall completely within the new sequence are preserved (with their locations adjusted).

For example, taking the same GenBank file used earlier:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.gb", "genbank")
>>> record
SeqRecord(seq=Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG'), id=
↪'NC_005816.1', name='NC_005816', description='Yersinia pestis biovar Microtus str.␣
↪91001 plasmid pPCP1, complete sequence', dbxrefs=['Project:58037'])
>>> len(record)
```

(continues on next page)

(continued from previous page)

```
9609
>>> len(record.features)
41
```

For this example we're going to focus in on the `pim` gene, `YP_pPCP05`. If you have a look at the GenBank file directly you'll find this gene/CDS has location string `4343..4780`, or in Python counting `4342:4780`. From looking at the file you can work out that these are the twelfth and thirteenth entries in the file, so in Python zero-based counting they are entries 11 and 12 in the `features` list:

```
>>> print(record.features[20])
type: gene
location: [4342:4780](+)
qualifiers:
    Key: db_xref, Value: ['GeneID:2767712']
    Key: gene, Value: ['pim']
    Key: locus_tag, Value: ['YP_pPCP05']

>>> print(record.features[21])
type: CDS
location: [4342:4780](+)
qualifiers:
    Key: codon_start, Value: ['1']
    Key: db_xref, Value: ['GI:45478716', 'GeneID:2767712']
    Key: gene, Value: ['pim']
    Key: locus_tag, Value: ['YP_pPCP05']
    Key: note, Value: ['similar to many previously sequenced pesticin immunity protein␣
↪entries of Yersinia pestis plasmid pPCP, e.g. gi| 16082683|,ref|NP_395230.1| (NC_
↪003132) , gi|1200166|emb|CAA90861.1| (Z54145 ) , gi|1488655| emb|CAA63439.1| (X92856) ,
↪ gi|2996219|gb|AAC62543.1| (AF053945) , and gi|5763814|emb|CAB531 67.1| (AL109969)']
    Key: product, Value: ['pesticin immunity protein']
    Key: protein_id, Value: ['NP_995571.1']
    Key: transl_table, Value: ['11']
    Key: translation, Value: [
↪'MGGGMISKLFCLALIFLSSSGLAEKNTYTAKDILQNLELNTFGNSLSHGIYGKQTTFKQTEFTNIKSNTKKHIALINKDNSWMISLKILGIKRDEYTVCF
↪']
```

Let's slice this parent record from 4300 to 4800 (enough to include the `pim` gene/CDS), and see how many features we get:

```
>>> sub_record = record[4300:4800]
>>> sub_record
SeqRecord(seq=Seq('ATAAATAGATTATTCCAAATAATTTATTTATGTAAGAACAGGATGGGAGGGGGA...TTA'), id=
↪'NC_005816.1', name='NC_005816', description='Yersinia pestis biovar Microtus str.␣
↪91001 plasmid pPCP1, complete sequence', dbxrefs=[])
>>> len(sub_record)
500
>>> len(sub_record.features)
2
```

Our sub-record just has two features, the gene and CDS entries for `YP_pPCP05`:

```
>>> print(sub_record.features[0])
type: gene
```

(continues on next page)

(continued from previous page)

```
location: [42:480](+)
qualifiers:
    Key: db_xref, Value: ['GeneID:2767712']
    Key: gene, Value: ['pim']
    Key: locus_tag, Value: ['YP_pPCP05']

>>> print(sub_record.features[1])
type: CDS
location: [42:480](+)
qualifiers:
    Key: codon_start, Value: ['1']
    Key: db_xref, Value: ['GI:45478716', 'GeneID:2767712']
    Key: gene, Value: ['pim']
    Key: locus_tag, Value: ['YP_pPCP05']
    Key: note, Value: ['similar to many previously sequenced pesticin immunity protein␣
↪entries of Yersinia pestis plasmid pPCP, e.g. gi| 16082683|,ref|NP_395230.1| (NC_
↪003132) , gi|1200166|emb|CAA90861.1| (Z54145 ) , gi|1488655| emb|CAA63439.1| (X92856) ,
↪ gi|2996219|gb|AAC62543.1| (AF053945) , and gi|5763814|emb|CAB531 67.1| (AL109969)']
    Key: product, Value: ['pesticin immunity protein']
    Key: protein_id, Value: ['NP_995571.1']
    Key: transl_table, Value: ['11']
    Key: translation, Value: [
↪'MGGGMISKLFCLALIFLSSSGLAEKNTYTAKDILQNLELNTFGNSLSHGIYGKQTTFKQTEFTNIKSNTKKHIALINKDNSWMISLKILGIKRDEYTVCF
↪']
```

Notice that their locations have been adjusted to reflect the new parent sequence!

While Biopython has done something sensible and hopefully intuitive with the features (and any per-letter annotation), for the other annotation it is impossible to know if this still applies to the sub-sequence or not. To avoid guessing, with the exception of the molecule type, the `.annotations` and `.dbxrefs` are omitted from the sub-record, and it is up to you to transfer any relevant information as appropriate.

```
>>> sub_record.annotations
{'molecule_type': 'DNA'}
>>> sub_record.dbxrefs
[]
```

You may wish to preserve other entries like the organism? Beware of copying the entire annotations dictionary as in this case your partial sequence is no longer circular DNA - it is now linear:

```
>>> sub_record.annotations["topology"] = "linear"
```

The same point could be made about the record `id`, `name` and `description`, but for practicality these are preserved:

```
>>> sub_record.id
'NC_005816.1'
>>> sub_record.name
'NC_005816'
>>> sub_record.description
'Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence'
```

This illustrates the problem nicely though, our new sub-record is *not* the complete sequence of the plasmid, so the description is wrong! Let's fix this and then view the sub-record as a reduced GenBank file using the `format` method described above in Section *The format method*:

```
>>> sub_record.description = (
...     "Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, partial"
... )
>>> print(sub_record.format("genbank")[:200] + "...")
LOCUS       NC_005816                500 bp    DNA     linear   UNK 01-JAN-1980
DEFINITION  Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, partial.
ACCESSION   NC_005816
VERSION     NC_0058...
```

See Sections *Trimming off primer sequences* and *Trimming off adaptor sequences* for some FASTQ examples where the per-letter annotations (the read quality scores) are also sliced.

4.8 Adding SeqRecord objects

You can add `SeqRecord` objects together, giving a new `SeqRecord`. What is important here is that any common per-letter annotations are also added, all the features are preserved (with their locations adjusted), and any other common annotation is also kept (like the id, name and description).

For an example with per-letter annotation, we'll use the first record in a FASTQ file. Chapter *Sequence Input/Output* will explain the `SeqIO` functions:

```
>>> from Bio import SeqIO
>>> record = next(SeqIO.parse("example.fastq", "fastq"))
>>> len(record)
25
>>> print(record.seq)
CCCTTCTTGTCTTCAGCGTTTCTCC
>>> print(record.letter_annotations["phred_quality"])
[26, 26, 18, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 22, 26, 26, 26, 26, 26, 26,␣
↪26, 23, 23]
```

Let's suppose this was Roche 454 data, and that from other information you think the `TTT` should be only `TT`. We can make a new edited record by first slicing the `SeqRecord` before and after the "extra" third `T`:

```
>>> left = record[:20]
>>> print(left.seq)
CCCTTCTTGTCTTCAGCGTT
>>> print(left.letter_annotations["phred_quality"])
[26, 26, 18, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 22, 26, 26, 26, 26]
>>> right = record[21:]
>>> print(right.seq)
CTCC
>>> print(right.letter_annotations["phred_quality"])
[26, 26, 23, 23]
```

Now add the two parts together:

```
>>> edited = left + right
>>> len(edited)
24
>>> print(edited.seq)
CCCTTCTTGTCTTCAGCGTTCTCC
```

(continues on next page)

(continued from previous page)

```
>>> print(edited.letter_annotations["phred_quality"])
[26, 26, 18, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 22, 26, 26, 26, 26, 26, 26,␣
↪23, 23]
```

Easy and intuitive? We hope so! You can make this shorter with just:

```
>>> edited = record[:20] + record[21:]
```

Now, for an example with features, we'll use a GenBank file. Suppose you have a circular genome:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.gb", "genbank")
>>> record
SeqRecord(seq=Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG'), id=
↪'NC_005816.1', name='NC_005816', description='Yersinia pestis biovar Microtus str.␣
↪91001 plasmid pPCP1, complete sequence', dbxrefs=['Project:58037'])
>>> len(record)
9609
>>> len(record.features)
41
>>> record.dbxrefs
['Project:58037']
>>> record.annotations.keys()
dict_keys(['molecule_type', 'topology', 'data_file_division', 'date', 'accessions',
↪'sequence_version', 'gi', 'keywords', 'source', 'organism', 'taxonomy', 'references',
↪'comment'])
```

You can shift the origin like this:

```
>>> shifted = record[2000:] + record[:2000]
>>> shifted
SeqRecord(seq=Seq('GATACGCAGTCATATTTTTTACACAATTCTCTAATCCCGACAAGGTCGTAGGTC...GGA'), id=
↪'NC_005816.1', name='NC_005816', description='Yersinia pestis biovar Microtus str.␣
↪91001 plasmid pPCP1, complete sequence', dbxrefs=[])
>>> len(shifted)
9609
```

Note that this isn't perfect in that some annotation like the database cross references, all the annotations except molecule type, and one of the features (the source feature) have been lost:

```
>>> len(shifted.features)
40
>>> shifted.dbxrefs
[]
>>> shifted.annotations.keys()
dict_keys(['molecule_type'])
```

This is because the `SeqRecord` slicing step is cautious in what annotation it preserves (erroneously propagating annotation can cause major problems). If you want to keep the database cross references or the annotations dictionary, this must be done explicitly:

```
>>> shifted.dbxrefs = record.dbxrefs[:]
>>> shifted.annotations = record.annotations.copy()
```

(continues on next page)

(continued from previous page)

```
>>> shifted.dbxrefs
['Project:58037']
>>> shifted.annotations.keys()
dict_keys(['molecule_type', 'topology', 'data_file_division', 'date', 'accessions',
↪'sequence_version', 'gi', 'keywords', 'source', 'organism', 'taxonomy', 'references',
↪'comment'])
```

Also note that in an example like this, you should probably change the record identifiers since the NCBI references refer to the *original* unmodified sequence.

4.9 Reverse-complementing SeqRecord objects

One of the new features in Biopython 1.57 was the `SeqRecord` object's `reverse_complement` method. This tries to balance easy of use with worries about what to do with the annotation in the reverse complemented record.

For the sequence, this uses the Seq object's reverse complement method. Any features are transferred with the location and strand recalculated. Likewise any per-letter-annotation is also copied but reversed (which makes sense for typical examples like quality scores). However, transfer of most annotation is problematical.

For instance, if the record ID was an accession, that accession should not really apply to the reverse complemented sequence, and transferring the identifier by default could easily cause subtle data corruption in downstream analysis. Therefore by default, the `SeqRecord`'s id, name, description, annotations and database cross references are all *not* transferred by default.

The `SeqRecord` object's `reverse_complement` method takes a number of optional arguments corresponding to properties of the record. Setting these arguments to `True` means copy the old values, while `False` means drop the old values and use the default value. You can alternatively provide the new desired value instead.

Consider this example record:

```
>>> from Bio import SeqIO
>>> rec = SeqIO.read("NC_005816.gb", "genbank")
>>> print(rec.id, len(rec), len(rec.features), len(rec.dbxrefs), len(rec.annotations))
NC_005816.1 9609 41 1 13
```

Here we take the reverse complement and specify a new identifier – but notice how most of the annotation is dropped (but not the features):

```
>>> rc = rec.reverse_complement(id="TESTING")
>>> print(rc.id, len(rc), len(rc.features), len(rc.dbxrefs), len(rc.annotations))
TESTING 9609 41 0 0
```

CHAPTER

FIVE

SEQUENCE INPUT/OUTPUT

In this chapter we'll discuss in more detail the `Bio.SeqIO` module, which was briefly introduced in Chapter *Quick Start – What can you do with Biopython?* and also used in Chapter *Sequence annotation objects*. This aims to provide a simple interface for working with assorted sequence file formats in a uniform way. See also the `Bio.SeqIO` wiki page (http://biopython.org/wiki/SeqIO), and the built-in documentation *`Bio.Seq`*:

```
>>> from Bio import SeqIO
>>> help(SeqIO)
```

The "catch" is that you have to work with `SeqRecord` objects (see Chapter *Sequence annotation objects*), which contain a `Seq` object (see Chapter *Sequence objects*) plus annotation like an identifier and description. Note that when dealing with very large FASTA or FASTQ files, the overhead of working with all these objects can make scripts too slow. In this case consider the low-level `SimpleFastaParser` and `FastqGeneralIterator` parsers which return just a tuple of strings for each record (see Section *Low level FASTA and FASTQ parsers*).

5.1 Parsing or Reading Sequences

The workhorse function `Bio.SeqIO.parse()` is used to read in sequence data as SeqRecord objects. This function expects two arguments:

1. The first argument is a *handle* to read the data from, or a filename. A handle is typically a file opened for reading, but could be the output from a command line program, or data downloaded from the internet (see Section *Parsing sequences from the net*). See Section *What the heck is a handle?* for more about handles.
2. The second argument is a lower case string specifying sequence format – we don't try and guess the file format for you! See http://biopython.org/wiki/SeqIO for a full listing of supported formats.

The `Bio.SeqIO.parse()` function returns an *iterator* which gives `SeqRecord` objects. Iterators are typically used in a for loop as shown below.

Sometimes you'll find yourself dealing with files which contain only a single record. For this situation use the function `Bio.SeqIO.read()` which takes the same arguments. Provided there is one and only one record in the file, this is returned as a `SeqRecord` object. Otherwise an exception is raised.

5.1.1 Reading Sequence Files

In general `Bio.SeqIO.parse()` is used to read in sequence files as `SeqRecord` objects, and is typically used with a for loop like this:

```
from Bio import SeqIO

for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta"):
    print(seq_record.id)
    print(repr(seq_record.seq))
    print(len(seq_record))
```

The above example is repeated from the introduction in Section *Parsing sequence file formats*, and will load the orchid DNA sequences in the FASTA format file ls_orchid.fasta. If instead you wanted to load a GenBank format file like ls_orchid.gbk then all you need to do is change the filename and the format string:

```
from Bio import SeqIO

for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank"):
    print(seq_record.id)
    print(repr(seq_record.seq))
    print(len(seq_record))
```

Similarly, if you wanted to read in a file in another file format, then assuming `Bio.SeqIO.parse()` supports it you would just need to change the format string as appropriate, for example "swiss" for SwissProt files or "embl" for EMBL text files. There is a full listing on the wiki page (http://biopython.org/wiki/SeqIO) and in the built-in documentation *`Bio.SeqIO`*:

Another very common way to use a Python iterator is within a list comprehension (or a generator expression). For example, if all you wanted to extract from the file was a list of the record identifiers we can easily do this with the following list comprehension:

```
>>> from Bio import SeqIO
>>> identifiers = [seq_record.id for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank
↪")]
>>> identifiers
['Z78533.1', 'Z78532.1', 'Z78531.1', 'Z78530.1', 'Z78529.1', 'Z78527.1', ..., 'Z78439.1']
```

There are more examples using `SeqIO.parse()` in a list comprehension like this in Section *Sequence parsing plus simple plots* (e.g. for plotting sequence lengths or GC%).

5.1.2 Iterating over the records in a sequence file

In the above examples, we have usually used a for loop to iterate over all the records one by one. You can use the for loop with all sorts of Python objects (including lists, tuples and strings) which support the iteration interface.

The object returned by `Bio.SeqIO` is actually an iterator which returns `SeqRecord` objects. You get to see each record in turn, but once and only once. The plus point is that an iterator can save you memory when dealing with large files.

Instead of using a for loop, can also use the `next()` function on an iterator to step through the entries, like this:

```
from Bio import SeqIO

record_iterator = SeqIO.parse("ls_orchid.fasta", "fasta")
```

(continues on next page)

(continued from previous page)

```
first_record = next(record_iterator)
print(first_record.id)
print(first_record.description)

second_record = next(record_iterator)
print(second_record.id)
print(second_record.description)
```

Note that if you try to use `next()` and there are no more results, you'll get the special `StopIteration` exception.

One special case to consider is when your sequence files have multiple records, but you only want the first one. In this situation the following code is very concise:

```
from Bio import SeqIO

first_record = next(SeqIO.parse("ls_orchid.gbk", "genbank"))
```

A word of warning here – using the `next()` function like this will silently ignore any additional records in the file. If your files have *one and only one* record, like some of the online examples later in this chapter, or a GenBank file for a single chromosome, then use the new `Bio.SeqIO.read()` function instead. This will check there are no extra unexpected records present.

5.1.3 Getting a list of the records in a sequence file

In the previous section we talked about the fact that `Bio.SeqIO.parse()` gives you a `SeqRecord` iterator, and that you get the records one by one. Very often you need to be able to access the records in any order. The Python `list` data type is perfect for this, and we can turn the record iterator into a list of `SeqRecord` objects using the built-in Python function `list()` like so:

```
from Bio import SeqIO

records = list(SeqIO.parse("ls_orchid.gbk", "genbank"))

print("Found %i records" % len(records))

print("The last record")
last_record = records[-1]  # using Python's list tricks
print(last_record.id)
print(repr(last_record.seq))
print(len(last_record))

print("The first record")
first_record = records[0]  # remember, Python counts from zero
print(first_record.id)
print(repr(first_record.seq))
print(len(first_record))
```

Giving:

```
Found 94 records
The last record
Z78439.1
```

(continues on next page)

(continued from previous page)

```
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC')
592
The first record
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC')
740
```

You can of course still use a for loop with a list of `SeqRecord` objects. Using a list is much more flexible than an iterator (for example, you can determine the number of records from the length of the list), but does need more memory because it will hold all the records in memory at once.

5.1.4 Extracting data

The `SeqRecord` object and its annotation structures are described more fully in Chapter *Sequence annotation objects*. As an example of how annotations are stored, we'll look at the output from parsing the first record in the GenBank file ls_orchid.gbk.

```
from Bio import SeqIO

record_iterator = SeqIO.parse("ls_orchid.gbk", "genbank")
first_record = next(record_iterator)
print(first_record)
```

That should give something like this:

```
ID: Z78533.1
Name: Z78533
Description: C.irapeanum 5.8S rRNA gene and ITS1 and ITS2 DNA.
Number of features: 5
/sequence_version=1
/source=Cypripedium irapeanum
/taxonomy=['Eukaryota', 'Viridiplantae', 'Streptophyta', ..., 'Cypripedium']
/keywords=['5.8S ribosomal RNA', '5.8S rRNA gene', ..., 'ITS1', 'ITS2']
/references=[...]
/accessions=['Z78533']
/data_file_division=PLN
/date=30-NOV-2006
/organism=Cypripedium irapeanum
/gi=2765658
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC')
```

This gives a human readable summary of most of the annotation data for the `SeqRecord`. For this example we're going to use the `.annotations` attribute which is just a Python dictionary. The contents of this annotations dictionary were shown when we printed the record above. You can also print them out directly:

```
print(first_record.annotations)
```

Like any Python dictionary, you can easily get the keys:

```
print(first_record.annotations.keys())
```

or values:

```
print(first_record.annotations.values())
```

In general, the annotation values are strings, or lists of strings. One special case is any references in the file get stored as reference objects.

Suppose you wanted to extract a list of the species from the ls_orchid.gbk GenBank file. The information we want, *Cypripedium irapeanum*, is held in the annotations dictionary under 'source' and 'organism', which we can access like this:

```
>>> print(first_record.annotations["source"])
Cypripedium irapeanum
```

or:

```
>>> print(first_record.annotations["organism"])
Cypripedium irapeanum
```

In general, 'organism' is used for the scientific name (in Latin, e.g. *Arabidopsis thaliana*), while 'source' will often be the common name (e.g. thale cress). In this example, as is often the case, the two fields are identical.

Now let's go through all the records, building up a list of the species each orchid sequence is from:

```
from Bio import SeqIO

all_species = []
for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank"):
    all_species.append(seq_record.annotations["organism"])
print(all_species)
```

Another way of writing this code is to use a list comprehension:

```
from Bio import SeqIO

all_species = [
    seq_record.annotations["organism"]
    for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank")
]
print(all_species)
```

In either case, the result is:

```
['Cypripedium irapeanum', 'Cypripedium californicum', ..., 'Paphiopedilum barbatum']
```

Great. That was pretty easy because GenBank files are annotated in a standardized way.

Now, let's suppose you wanted to extract a list of the species from a FASTA file, rather than the GenBank file. The bad news is you will have to write some code to extract the data you want from the record's description line - if the information is in the file in the first place! Our example FASTA format file ls_orchid.fasta starts like this:

```
>gi|2765658|emb|Z78533.1|CIZ78533 C.irapeanum 5.8S rRNA gene and ITS1 and ITS2 DNA
CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGGAATAAACGATCGAGTG
AATCCGGAGGACCGGTGTACTCAGCTCACCGGGGGCATTGCTCCCGTGGTGACCCTGATTTGTTGTTGGG
...
```

You can check by hand, but for every record the species name is in the description line as the second word. This means if we break up each record's `.description` at the spaces, then the species is there as field number one (field zero is

the record identifier). That means we can do this:

```
>>> from Bio import SeqIO
>>> all_species = []
>>> for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta"):
...     all_species.append(seq_record.description.split()[1])
...
>>> print(all_species)
['C.irapeanum', 'C.californicum', 'C.fasciculatum', ..., 'P.barbatum']
```

The concise alternative using list comprehensions would be:

```
>>> from Bio import SeqIO
>>> all_species = [
...     seq_record.description.split()[1]
...     for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta")
... ]
>>> print(all_species)
['C.irapeanum', 'C.californicum', 'C.fasciculatum', ..., 'P.barbatum']
```

In general, extracting information from the FASTA description line is not very nice. If you can get your sequences in a well annotated file format like GenBank or EMBL, then this sort of annotation information is much easier to deal with.

5.1.5 Modifying data

In the previous section, we demonstrated how to extract data from a `SeqRecord`. Another common task is to alter this data. The attributes of a `SeqRecord` can be modified directly, for example:

```
>>> from Bio import SeqIO
>>> record_iterator = SeqIO.parse("ls_orchid.fasta", "fasta")
>>> first_record = next(record_iterator)
>>> first_record.id
'gi|2765658|emb|Z78533.1|CIZ78533'
>>> first_record.id = "new_id"
>>> first_record.id
'new_id'
```

Note, if you want to change the way FASTA is output when written to a file (see Section *Writing Sequence Files*), then you should modify both the `id` and `description` attributes. To ensure the correct behavior, it is best to include the `id` plus a space at the start of the desired `description`:

```
>>> from Bio import SeqIO
>>> record_iterator = SeqIO.parse("ls_orchid.fasta", "fasta")
>>> first_record = next(record_iterator)
>>> first_record.id = "new_id"
>>> first_record.description = first_record.id + " " + "desired new description"
>>> print(first_record.format("fasta")[:200])
>new_id desired new description
CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGGAATAAA
CGATCGAGTGAATCCGGAGGACCGGTGTACTCAGCTCACCGGGGGCATTGCTCCCGTGGT
GACCCTGATTTGTTGTTGGGCCGCCTCGGGAGCGTCCATGGCGGGT
```

5.2 Parsing sequences from compressed files

In the previous section, we looked at parsing sequence data from a file. Instead of using a filename, you can give `Bio.SeqIO` a handle (see Section *What the heck is a handle?*), and in this section we'll use handles to parse sequence from compressed files.

As you'll have seen above, we can use `Bio.SeqIO.read()` or `Bio.SeqIO.parse()` with a filename - for instance this quick example calculates the total length of the sequences in a multiple record GenBank file using a generator expression:

```
>>> from Bio import SeqIO
>>> print(sum(len(r) for r in SeqIO.parse("ls_orchid.gbk", "gb")))
67518
```

Here we use a file handle instead, using the `with` statement to close the handle automatically:

```
>>> from Bio import SeqIO
>>> with open("ls_orchid.gbk") as handle:
...     print(sum(len(r) for r in SeqIO.parse(handle, "gb")))
...
67518
```

Or, the old fashioned way where you manually close the handle:

```
>>> from Bio import SeqIO
>>> handle = open("ls_orchid.gbk")
>>> print(sum(len(r) for r in SeqIO.parse(handle, "gb")))
67518
>>> handle.close()
```

Now, suppose we have a gzip compressed file instead? These are very commonly used on Linux. We can use Python's `gzip` module to open the compressed file for reading - which gives us a handle object:

```
>>> import gzip
>>> from Bio import SeqIO
>>> with gzip.open("ls_orchid.gbk.gz", "rt") as handle:
...     print(sum(len(r) for r in SeqIO.parse(handle, "gb")))
...
67518
```

Similarly if we had a bzip2 compressed file:

```
>>> import bz2
>>> from Bio import SeqIO
>>> with bz2.open("ls_orchid.gbk.bz2", "rt") as handle:
...     print(sum(len(r) for r in SeqIO.parse(handle, "gb")))
...
67518
```

There is a gzip (GNU Zip) variant called BGZF (Blocked GNU Zip Format), which can be treated like an ordinary gzip file for reading, but has advantages for random access later which we'll talk about later in Section *Indexing compressed files*.

5.3 Parsing sequences from the net

In the previous sections, we looked at parsing sequence data from a file (using a filename or handle), and from compressed files (using a handle). Here we'll use `Bio.SeqIO` with another type of handle, a network connection, to download and parse sequences from the internet.

Note that just because you *can* download sequence data and parse it into a `SeqRecord` object in one go doesn't mean this is a good idea. In general, you should probably download sequences *once* and save them to a file for reuse.

5.3.1 Parsing GenBank records from the net

Section *EFetch: Downloading full records from Entrez* talks about the Entrez EFetch interface in more detail, but for now let's just connect to the NCBI and get a few *Opuntia* (prickly-pear) sequences from GenBank using their GI numbers.

First of all, let's fetch just one record. If you don't care about the annotations and features downloading a FASTA file is a good choice as these are compact. Now remember, when you expect the handle to contain one and only one record, use the `Bio.SeqIO.read()` function:

```
from Bio import Entrez
from Bio import SeqIO

Entrez.email = "A.N.Other@example.com"
with Entrez.efetch(
    db="nucleotide", rettype="fasta", retmode="text", id="6273291"
) as handle:
    seq_record = SeqIO.read(handle, "fasta")
print("%s with %i features" % (seq_record.id, len(seq_record.features)))
```

Expected output:

```
gi|6273291|gb|AF191665.1|AF191665 with 0 features
```

The NCBI will also let you ask for the file in other formats, in particular as a GenBank file. Until Easter 2009, the Entrez EFetch API let you use "genbank" as the return type, however the NCBI now insist on using the official return types of "gb" (or "gp" for proteins) as described on EFetch for Sequence and other Molecular Biology Databases. As a result, in Biopython 1.50 onwards, we support "gb" as an alias for "genbank" in `Bio.SeqIO`.

```
from Bio import Entrez
from Bio import SeqIO

Entrez.email = "A.N.Other@example.com"
with Entrez.efetch(
    db="nucleotide", rettype="gb", retmode="text", id="6273291"
) as handle:
    seq_record = SeqIO.read(handle, "gb")  # using "gb" as an alias for "genbank"
print("%s with %i features" % (seq_record.id, len(seq_record.features)))
```

The expected output of this example is:

```
AF191665.1 with 3 features
```

Notice this time we have three features.

Now let's fetch several records. This time the handle contains multiple records, so we must use the `Bio.SeqIO.parse()` function:

```
from Bio import Entrez
from Bio import SeqIO

Entrez.email = "A.N.Other@example.com"
with Entrez.efetch(
    db="nucleotide", rettype="gb", retmode="text", id="6273291,6273290,6273289"
) as handle:
    for seq_record in SeqIO.parse(handle, "gb"):
        print("%s %s..." % (seq_record.id, seq_record.description[:50]))
        print(
            "Sequence length %i, %i features, from: %s"
            % (
                len(seq_record),
                len(seq_record.features),
                seq_record.annotations["source"],
            )
        )
```

That should give the following output:

```
AF191665.1 Opuntia marenae rpl16 gene; chloroplast gene for c...
Sequence length 902, 3 features, from: chloroplast Opuntia marenae
AF191664.1 Opuntia clavata rpl16 gene; chloroplast gene for c...
Sequence length 899, 3 features, from: chloroplast Grusonia clavata
AF191663.1 Opuntia bradtiana rpl16 gene; chloroplast gene for...
Sequence length 899, 3 features, from: chloroplast Opuntia bradtianaa
```

See Chapter *Accessing NCBI's Entrez databases* for more about the `Bio.Entrez` module, and make sure to read about the NCBI guidelines for using Entrez (Section *Entrez Guidelines*).

5.3.2 Parsing SwissProt sequences from the net

Now let's use a handle to download a SwissProt file from ExPASy, something covered in more depth in Chapter *SwissProt and ExPASy*. As mentioned above, when you expect the handle to contain one and only one record, use the `Bio.SeqIO.read()` function:

```
from Bio import ExPASy
from Bio import SeqIO

with ExPASy.get_sprot_raw("O23729") as handle:
    seq_record = SeqIO.read(handle, "swiss")
print(seq_record.id)
print(seq_record.name)
print(seq_record.description)
print(repr(seq_record.seq))
print("Length %i" % len(seq_record))
print(seq_record.annotations["keywords"])
```

Assuming your network connection is OK, you should get back:

```
O23729
CHS3_BROFI
RecName: Full=Chalcone synthase 3; EC=2.3.1.74; AltName: Full=Naringenin-chalcone␣
↪synthase 3;
Seq('MAPAMEEIRQAQRAEGPAAVLAIGTSTPPNALYQADYPDYYFRITKSEHLTELK...GAE')
Length 394
['Acyltransferase', 'Flavonoid biosynthesis', 'Transferase']
```

5.4 Sequence files as Dictionaries

Looping over the iterator returned by `SeqIO.parse` once will exhaust the file. For self-indexed files, such as files in the twoBit format, the return value of `SeqIO.parse` can also be used as a dictionary, allowing random access to the sequence contents. As in this case parsing is done on demand, the file must remain open as long as the sequence data is being accessed:

```
>>> from Bio import SeqIO
>>> handle = open("sequence.bigendian.2bit", "rb")
>>> records = SeqIO.parse(handle, "twobit")
>>> records.keys()
dict_keys(['seq11111', 'seq222', 'seq3333', 'seq4', 'seq555', 'seq6'])
>>> records["seq222"]
SeqRecord(seq=Seq('TTGATCGGTGACAAATTTTTTACAAAGAACTGTAGGACTTGCTACTTCTCCCTC...ACA'), id=
↪'seq222', name='<unknown name>', description='<unknown description>', dbxrefs=[])
>>> records["seq222"].seq
Seq('TTGATCGGTGACAAATTTTTTACAAAGAACTGTAGGACTTGCTACTTCTCCCTC...ACA')
>>> handle.close()
>>> records["seq222"].seq
Traceback (most recent call last):
...
ValueError: cannot retrieve sequence: file is closed
```

For other file formats, `Bio.SeqIO` provides three related functions module which allow dictionary like random access to a multi-sequence file. There is a trade off here between flexibility and memory usage. In summary:

- `Bio.SeqIO.to_dict()` is the most flexible but also the most memory demanding option (see Section *Sequence files as Dictionaries – In memory*). This is basically a helper function to build a normal Python `dictionary` with each entry held as a `SeqRecord` object in memory, allowing you to modify the records.
- `Bio.SeqIO.index()` is a useful middle ground, acting like a read only dictionary and parsing sequences into `SeqRecord` objects on demand (see Section *Sequence files as Dictionaries – Indexed files*).
- `Bio.SeqIO.index_db()` also acts like a read only dictionary but stores the identifiers and file offsets in a file on disk (as an SQLite3 database), meaning it has very low memory requirements (see Section *Sequence files as Dictionaries – Database indexed files*), but will be a little bit slower.

See the discussion for an broad overview (Section *Discussion*).

5.4.1 Sequence files as Dictionaries – In memory

The next thing that we'll do with our ubiquitous orchid files is to show how to index them and access them like a database using the Python `dictionary` data type (like a hash in Perl). This is very useful for moderately large files where you only need to access certain elements of the file, and makes for a nice quick 'n dirty database. For dealing with larger files where memory becomes a problem, see Section *Sequence files as Dictionaries – Indexed files* below.

You can use the function `Bio.SeqIO.to_dict()` to make a SeqRecord dictionary (in memory). By default this will use each record's identifier (i.e. the `.id` attribute) as the key. Let's try this using our GenBank file:

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.to_dict(SeqIO.parse("ls_orchid.gbk", "genbank"))
```

There is just one required argument for `Bio.SeqIO.to_dict()`, a list or generator giving `SeqRecord` objects. Here we have just used the output from the `SeqIO.parse` function. As the name suggests, this returns a Python dictionary.

Since this variable `orchid_dict` is an ordinary Python dictionary, we can look at all of the keys we have available:

```
>>> len(orchid_dict)
94
```

```
>>> list(orchid_dict.keys())
['Z78484.1', 'Z78464.1', 'Z78455.1', 'Z78442.1', 'Z78532.1', 'Z78453.1', ..., 'Z78471.1']
```

Under Python 3 the dictionary methods like ".keys()" and ".values()" are iterators rather than lists.

If you really want to, you can even look at all the records at once:

```
>>> list(orchid_dict.values())  # lots of output!
```

We can access a single `SeqRecord` object via the keys and manipulate the object as normal:

```
>>> seq_record = orchid_dict["Z78475.1"]
>>> print(seq_record.description)
P.supardii 5.8S rRNA gene and ITS1 and ITS2 DNA
>>> seq_record.seq
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGATCACAT...GGT')
```

So, it is very easy to create an in memory "database" of our GenBank records. Next we'll try this for the FASTA file instead.

Note that those of you with prior Python experience should all be able to construct a dictionary like this "by hand". However, typical dictionary construction methods will not deal with the case of repeated keys very nicely. Using the `Bio.SeqIO.to_dict()` will explicitly check for duplicate keys, and raise an exception if any are found.

Specifying the dictionary keys

Using the same code as above, but for the FASTA file instead:

```
from Bio import SeqIO

orchid_dict = SeqIO.to_dict(SeqIO.parse("ls_orchid.fasta", "fasta"))
print(orchid_dict.keys())
```

This time the keys are:

```
['gi|2765596|emb|Z78471.1|PDZ78471', 'gi|2765646|emb|Z78521.1|CCZ78521', ...
 ..., 'gi|2765613|emb|Z78488.1|PTZ78488', 'gi|2765583|emb|Z78458.1|PHZ78458']
```

You should recognize these strings from when we parsed the FASTA file earlier in Section *Simple FASTA parsing example*. Suppose you would rather have something else as the keys - like the accession numbers. This brings us nicely to `SeqIO.to_dict()`'s optional argument `key_function`, which lets you define what to use as the dictionary key for your records.

First you must write your own function to return the key you want (as a string) when given a `SeqRecord` object. In general, the details of function will depend on the sort of input records you are dealing with. But for our orchids, we can just split up the record's identifier using the "pipe" character (the vertical line) and return the fourth entry (field three):

```
def get_accession(record):
    """Given a SeqRecord, return the accession number as a string.

    e.g. "gi|2765613|emb|Z78488.1|PTZ78488" -> "Z78488.1"
    """
    parts = record.id.split("|")
    assert len(parts) == 5 and parts[0] == "gi" and parts[2] == "emb"
    return parts[3]
```

Then we can give this function to the `SeqIO.to_dict()` function to use in building the dictionary:

```
from Bio import SeqIO

orchid_dict = SeqIO.to_dict(
    SeqIO.parse("ls_orchid.fasta", "fasta"), key_function=get_accession
)
print(orchid_dict.keys())
```

Finally, as desired, the new dictionary keys:

```
>>> print(orchid_dict.keys())
['Z78484.1', 'Z78464.1', 'Z78455.1', 'Z78442.1', 'Z78532.1', 'Z78453.1', ..., 'Z78471.1']
```

Not too complicated, I hope!

Indexing a dictionary using the SEGUID checksum

To give another example of working with dictionaries of `SeqRecord` objects, we'll use the SEGUID checksum function. This is a relatively recent checksum, and collisions should be very rare (i.e. two different sequences with the same checksum), an improvement on the CRC64 checksum.

Once again, working with the orchids GenBank file:

```
from Bio import SeqIO
from Bio.SeqUtils.CheckSum import seguid

for record in SeqIO.parse("ls_orchid.gbk", "genbank"):
    print(record.id, seguid(record.seq))
```

This should give:

```
Z78533.1 JUEoWn6DPhgZ9nAyowsgtoD9TTo
Z78532.1 MN/s0q9zDoCVEEc+k/IFwCNF2pY
...
Z78439.1 H+JfaShya/4yyAj7IbMqgNkxdxQ
```

Now, recall the `Bio.SeqIO.to_dict()` function's `key_function` argument expects a function which turns a `SeqRecord` into a string. We can't use the `seguid()` function directly because it expects to be given a `Seq` object (or a string). However, we can use Python's `lambda` feature to create a "one off" function to give to `Bio.SeqIO.to_dict()` instead:

```
>>> from Bio import SeqIO
>>> from Bio.SeqUtils.CheckSum import seguid
>>> seguid_dict = SeqIO.to_dict(
...     SeqIO.parse("ls_orchid.gbk", "genbank"), lambda rec: seguid(rec.seq)
... )
>>> record = seguid_dict["MN/s0q9zDoCVEEc+k/IFwCNF2pY"]
>>> print(record.id)
Z78532.1
>>> print(record.description)
C.californicum 5.8S rRNA gene and ITS1 and ITS2 DNA
```

That should have retrieved the record `Z78532.1`, the second entry in the file.

5.4.2 Sequence files as Dictionaries – Indexed files

As the previous couple of examples tried to illustrate, using `Bio.SeqIO.to_dict()` is very flexible. However, because it holds everything in memory, the size of file you can work with is limited by your computer's RAM. In general, this will only work on small to medium files.

For larger files you should consider `Bio.SeqIO.index()`, which works a little differently. Although it still returns a dictionary like object, this does *not* keep *everything* in memory. Instead, it just records where each record is within the file – when you ask for a particular record, it then parses it on demand.

As an example, let's use the same GenBank file as before:

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index("ls_orchid.gbk", "genbank")
>>> len(orchid_dict)
94
```

```
>>> orchid_dict.keys()
['Z78484.1', 'Z78464.1', 'Z78455.1', 'Z78442.1', 'Z78532.1', 'Z78453.1', ..., 'Z78471.1']
```

```
>>> seq_record = orchid_dict["Z78475.1"]
>>> print(seq_record.description)
P.supardii 5.8S rRNA gene and ITS1 and ITS2 DNA
>>> seq_record.seq
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGATCACAT...GGT')
>>> orchid_dict.close()
```

Note that `Bio.SeqIO.index()` won't take a handle, but only a filename. There are good reasons for this, but it is a little technical. The second argument is the file format (a lower case string as used in the other `Bio.SeqIO` functions). You can use many other simple file formats, including FASTA and FASTQ files (see the example in Section *Indexing*

a FASTQ file). However, alignment formats like PHYLIP or Clustal are not supported. Finally as an optional argument you can supply a key function.

Here is the same example using the FASTA file - all we change is the filename and the format name:

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index("ls_orchid.fasta", "fasta")
>>> len(orchid_dict)
94
>>> orchid_dict.keys()
['gi|2765596|emb|Z78471.1|PDZ78471', 'gi|2765646|emb|Z78521.1|CCZ78521', ...
 ..., 'gi|2765613|emb|Z78488.1|PTZ78488', 'gi|2765583|emb|Z78458.1|PHZ78458']
```

Specifying the dictionary keys

Suppose you want to use the same keys as before? Much like with the `Bio.SeqIO.to_dict()` example in Section *Specifying the dictionary keys*, you'll need to write a tiny function to map from the FASTA identifier (as a string) to the key you want:

```
def get_acc(identifier):
    """Given a SeqRecord identifier string, return the accession number as a string.

    e.g. "gi|2765613|emb|Z78488.1|PTZ78488" -> "Z78488.1"
    """
    parts = identifier.split("|")
    assert len(parts) == 5 and parts[0] == "gi" and parts[2] == "emb"
    return parts[3]
```

Then we can give this function to the `Bio.SeqIO.index()` function to use in building the dictionary:

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index("ls_orchid.fasta", "fasta", key_function=get_acc)
>>> print(orchid_dict.keys())
['Z78484.1', 'Z78464.1', 'Z78455.1', 'Z78442.1', 'Z78532.1', 'Z78453.1', ..., 'Z78471.1']
```

Easy when you know how?

Getting the raw data for a record

The dictionary-like object from `Bio.SeqIO.index()` gives you each entry as a `SeqRecord` object. However, it is sometimes useful to be able to get the original raw data straight from the file. For this use the `get_raw()` method which takes a single argument (the record identifier) and returns a bytes string (extracted from the file without modification).

A motivating example is extracting a subset of a records from a large file where either `Bio.SeqIO.write()` does not (yet) support the output file format (e.g. the plain text SwissProt file format) or where you need to preserve the text exactly (e.g. GenBank or EMBL output from Biopython does not yet preserve every last bit of annotation).

Let's suppose you have download the whole of UniProt in the plain text SwissPort file format from their FTP site (ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/complete/uniprot_sprot.dat.gz) and uncompressed it as the file `uniprot_sprot.dat`, and you want to extract just a few records from it:

```
>>> from Bio import SeqIO
>>> uniprot = SeqIO.index("uniprot_sprot.dat", "swiss")
>>> with open("selected.dat", "wb") as out_handle:
```

(continues on next page)

(continued from previous page)

```
...     for acc in ["P33487", "P19801", "P13689", "Q8JZQ5", "Q9TRC7"]:
...         out_handle.write(uniprot.get_raw(acc))
...
```

Note with Python 3 onwards, we have to open the file for writing in binary mode because the `get_raw()` method returns bytes strings.

There is a longer example in Section *Sorting a sequence file* using the `SeqIO.index()` function to sort a large sequence file (without loading everything into memory at once).

5.4.3 Sequence files as Dictionaries – Database indexed files

Biopython 1.57 introduced an alternative, `Bio.SeqIO.index_db()`, which can work on even extremely large files since it stores the record information as a file on disk (using an SQLite3 database) rather than in memory. Also, you can index multiple files together (providing all the record identifiers are unique).

The `Bio.SeqIO.index()` function takes three required arguments:

- Index filename, we suggest using something ending `.idx`. This index file is actually an SQLite3 database.
- List of sequence filenames to index (or a single filename)
- File format (lower case string as used in the rest of the `SeqIO` module).

As an example, consider the GenBank flat file releases from the NCBI FTP site, ftp://ftp.ncbi.nih.gov/genbank/, which are gzip compressed GenBank files.

As of GenBank release 210, there are 38 files making up the viral sequences, `gbvrl1.seq`, ..., `gbvrl38.seq`, taking about 8GB on disk once decompressed, and containing in total nearly two million records.

If you were interested in the viruses, you could download all the virus files from the command line very easily with the `rsync` command, and then decompress them with `gunzip`:

```
# For illustration only, see reduced example below
$ rsync -avP "ftp.ncbi.nih.gov::genbank/gbvrl*.seq.gz" .
$ gunzip gbvrl*.seq.gz
```

Unless you care about viruses, that's a lot of data to download just for this example - so let's download *just* the first four chunks (about 25MB each compressed), and decompress them (taking in all about 1GB of space):

```
# Reduced example, download only the first four chunks
$ curl -O ftp://ftp.ncbi.nih.gov/genbank/gbvrl1.seq.gz
$ curl -O ftp://ftp.ncbi.nih.gov/genbank/gbvrl2.seq.gz
$ curl -O ftp://ftp.ncbi.nih.gov/genbank/gbvrl3.seq.gz
$ curl -O ftp://ftp.ncbi.nih.gov/genbank/gbvrl4.seq.gz
$ gunzip gbvrl*.seq.gz
```

Now, in Python, index these GenBank files as follows:

```
>>> import glob
>>> from Bio import SeqIO
>>> files = glob.glob("gbvrl*.seq")
>>> print("%i files to index" % len(files))
4
>>> gb_vrl = SeqIO.index_db("gbvrl.idx", files, "genbank")
```

(continues on next page)

(continued from previous page)

```
>>> print("%i sequences indexed" % len(gb_vrl))
272960 sequences indexed
```

Indexing the full set of virus GenBank files took about ten minutes on my machine, just the first four files took about a minute or so.

However, once done, repeating this will reload the index file `gbvrl.idx` in a fraction of a second.

You can use the index as a read only Python dictionary - without having to worry about which file the sequence comes from, e.g.

```
>>> print(gb_vrl["AB811634.1"].description)
Equine encephalosis virus NS3 gene, complete cds, isolate: Kimron1.
```

Getting the raw data for a record

Just as with the `Bio.SeqIO.index()` function discussed above in Section *Getting the raw data for a record*, the dictionary like object also lets you get at the raw bytes of each record:

```
>>> print(gb_vrl.get_raw("AB811634.1"))
LOCUS       AB811634                723 bp    RNA     linear   VRL 17-JUN-2015
DEFINITION  Equine encephalosis virus NS3 gene, complete cds, isolate: Kimron1.
ACCESSION   AB811634
...
//
```

5.4.4 Indexing compressed files

Very often when you are indexing a sequence file it can be quite large – so you may want to compress it on disk. Unfortunately efficient random access is difficult with the more common file formats like gzip and bzip2. In this setting, BGZF (Blocked GNU Zip Format) can be very helpful. This is a variant of gzip (and can be decompressed using standard gzip tools) popularized by the BAM file format, samtools, and tabix.

To create a BGZF compressed file you can use the command line tool `bgzip` which comes with samtools. In our examples we use a filename extension `*.bgz`, so they can be distinguished from normal gzipped files (named `*.gz`). You can also use the `Bio.bgzf` module to read and write BGZF files from within Python.

The `Bio.SeqIO.index()` and `Bio.SeqIO.index_db()` can both be used with BGZF compressed files. For example, if you started with an uncompressed GenBank file:

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index("ls_orchid.gbk", "genbank")
>>> len(orchid_dict)
94
>>> orchid_dict.close()
```

You could compress this (while keeping the original file) at the command line using the following command – but don't worry, the compressed file is already included with the other example files:

```
$ bgzip -c ls_orchid.gbk > ls_orchid.gbk.bgz
```

You can use the compressed file in exactly the same way:

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index("ls_orchid.gbk.bgz", "genbank")
>>> len(orchid_dict)
94
>>> orchid_dict.close()
```

or:

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index_db("ls_orchid.gbk.bgz.idx", "ls_orchid.gbk.bgz", "genbank")
>>> len(orchid_dict)
94
>>> orchid_dict.close()
```

The `SeqIO` indexing automatically detects the BGZF compression. Note that you can't use the same index file for the uncompressed and compressed files.

5.4.5 Discussion

So, which of these methods should you use and why? It depends on what you are trying to do (and how much data you are dealing with). However, in general picking `Bio.SeqIO.index()` is a good starting point. If you are dealing with millions of records, multiple files, or repeated analyses, then look at `Bio.SeqIO.index_db()`.

Reasons to choose `Bio.SeqIO.to_dict()` over either `Bio.SeqIO.index()` or `Bio.SeqIO.index_db()` boil down to a need for flexibility despite its high memory needs. The advantage of storing the `SeqRecord` objects in memory is they can be changed, added to, or removed at will. In addition to the downside of high memory consumption, indexing can also take longer because all the records must be fully parsed.

Both `Bio.SeqIO.index()` and `Bio.SeqIO.index_db()` only parse records on demand. When indexing, they scan the file once looking for the start of each record and do as little work as possible to extract the identifier.

Reasons to choose `Bio.SeqIO.index()` over `Bio.SeqIO.index_db()` include:

- Faster to build the index (more noticeable in simple file formats)
- Slightly faster access as SeqRecord objects (but the difference is only really noticeable for simple to parse file formats).
- Can use any immutable Python object as the dictionary keys (e.g. a tuple of strings, or a frozen set) not just strings.
- Don't need to worry about the index database being out of date if the sequence file being indexed has changed.

Reasons to choose `Bio.SeqIO.index_db()` over `Bio.SeqIO.index()` include:

- Not memory limited – this is already important with files from second generation sequencing where 10s of millions of sequences are common, and using `Bio.SeqIO.index()` can require more than 4GB of RAM and therefore a 64bit version of Python.
- Because the index is kept on disk, it can be reused. Although building the index database file takes longer, if you have a script which will be rerun on the same datafiles in future, this could save time in the long run.
- Indexing multiple files together
- The `get_raw()` method can be much faster, since for most file formats the length of each record is stored as well as its offset.

5.5 Writing Sequence Files

We've talked about using `Bio.SeqIO.parse()` for sequence input (reading files), and now we'll look at `Bio.SeqIO.write()` which is for sequence output (writing files). This is a function taking three arguments: some `SeqRecord` objects, a handle or filename to write to, and a sequence format.

Here is an example, where we start by creating a few `SeqRecord` objects the hard way (by hand, rather than by loading them from a file):

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord

rec1 = SeqRecord(
    Seq(
        "MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQALFGD"
        "GAGAVIVGSDPDLSVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKDVPGLISK"
        "NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRATREVLSEYGNM"
        "SSAC",
    ),
    id="gi|14150838|gb|AAK54648.1|AF376133_1",
    description="chalcone synthase [Cucumis sativus]",
)

rec2 = SeqRecord(
    Seq(
        "YPDYYFRITNREHKAELKEKFQRMCDKSMIKKRYMYLTEEILKENPSMCEYMAPSLDARQ"
        "DMVVVEIPKLGKEAAVKAIKEWGQ",
    ),
    id="gi|13919613|gb|AAK33142.1|",
    description="chalcone synthase [Fragaria vesca subsp. bracteata]",
)

rec3 = SeqRecord(
    Seq(
        "MVTVEEFRRAQCAEGPATVMAIGTATPSNCVDQSTYPDYYFRITNSEHKVELKEKFKRMC"
        "EKSMIKKRYMHLTEEILKENPNICAYMAPSLDARQDIVVVEVPKLGKEAAQKAIKEWGQP"
        "KSKITHLVFCTTSGVDMPGCDYQLTKLLGLRPSVKRFMMYQQGCFAGGTVLRMAKDLAEN"
        "NKGARVLVVCSEITAVTFRGPNDTHLDSLVGQALFGDGAAAVIIGSDPIPEVERPLFELV"
        "SAAQTLLPDSEGAIDGHLREVGLTFHLLKDVPGLISKNIEKSLVEAFQPLGISDWNSLFW"
        "IAHPGGPAILDQVELKLGLKQEKLKATRKVLSNYGNMSSACVLFILDEMRKASAKEGLGT"
        "TGEGLEWGVLFGFGPGLTVETVVLHSVAT",
    ),
    id="gi|13925890|gb|AAK49457.1|",
    description="chalcone synthase [Nicotiana tabacum]",
)

my_records = [rec1, rec2, rec3]
```

Now we have a list of `SeqRecord` objects, we'll write them to a FASTA format file:

```
from Bio import SeqIO

SeqIO.write(my_records, "my_example.faa", "fasta")
```

And if you open this file in your favorite text editor it should look like this:

```
>gi|14150838|gb|AAK54648.1|AF376133_1 chalcone synthase [Cucumis sativus]
MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQALFGD
GAGAVIVGSDPDLSVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKDVPGLISK
NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRATREVLSEYGNM
SSAC
>gi|13919613|gb|AAK33142.1| chalcone synthase [Fragaria vesca subsp. bracteata]
YPDYYFRITNREHKAELKEKFQRMCDKSMIKKRYMYLTEEILKENPSMCEYMAPSLDARQ
DMVVVEIPKLGKEAAVKAIKEWGQ
>gi|13925890|gb|AAK49457.1| chalcone synthase [Nicotiana tabacum]
MVTVEEFRRAQCAEGPATVMAIGTATPSNCVDQSTYPDYYFRITNSEHKVELKEKFKRMC
EKSMIKKRYMHLTEEILKENPNICAYMAPSLDARQDIVVVEVPKLGKEAAQKAIKEWGQP
KSKITHLVFCTTSGVDMPGCDYQLTKLLGLRPSVKRFMMYQQGCFAGGTVLRMAKDLAEN
NKGARVLVVCSEITAVTFRGPNDTHLDSLVGQALFGDGAAAVIIGSDPIPEVERPLFELV
SAAQTLLPDSEGAIDGHLREVGLTFHLLKDVPGLISKNIEKSLVEAFQPLGISDWNSLFW
IAHPGGPAILDQVELKLGLKQEKLKATRKVLSNYGNMSSACVLFILDEMRKASAKEGLGT
TGEGLEWGVLFGFGPGLTVETVVLHSVAT
```

Suppose you wanted to know how many records the `Bio.SeqIO.write()` function wrote to the handle? If your records were in a list you could just use `len(my_records)`, however you can't do that when your records come from a generator/iterator. The `Bio.SeqIO.write()` function returns the number of `SeqRecord` objects written to the file.

Note - If you tell the `Bio.SeqIO.write()` function to write to a file that already exists, the old file will be overwritten without any warning.

5.5.1 Round trips

Some people like their parsers to be "round-tripable", meaning if you read in a file and write it back out again it is unchanged. This requires that the parser must extract enough information to reproduce the original file *exactly*. `Bio.SeqIO` does *not* aim to do this.

As a trivial example, any line wrapping of the sequence data in FASTA files is allowed. An identical `SeqRecord` would be given from parsing the following two examples which differ only in their line breaks:

```
>YAL068C-7235.2170 Putative promoter sequence
TACGAGAATAATTTCTCATCATCCAGCTTTAACACAAAATTCGCACAGTTTTCGTTAAGA
GAACTTAACATTTTCTTATGACGTAAATGAAGTTTATATATAAATTTCCTTTTTATTGGA

>YAL068C-7235.2170 Putative promoter sequence
TACGAGAATAATTTCTCATCATCCAGCTTTAACACAAAATTCGCA
CAGTTTTCGTTAAGAGAACTTAACATTTTCTTATGACGTAAATGA
AGTTTATATATAAATTTCCTTTTTATTGGA
```

To make a round-tripable FASTA parser you would need to keep track of where the sequence line breaks occurred, and this extra information is usually pointless. Instead Biopython uses a default line wrapping of 60 characters on output. The same problem with white space applies in many other file formats too. Another issue in some cases is that Biopython does not (yet) preserve every last bit of annotation (e.g. GenBank and EMBL).

Occasionally preserving the original layout (with any quirks it may have) is important. See Section *Getting the raw data for a record* about the `get_raw()` method of the `Bio.SeqIO.index()` dictionary-like object for one potential solution.

5.5.2 Converting between sequence file formats

In previous example we used a list of `SeqRecord` objects as input to the `Bio.SeqIO.write()` function, but it will also accept a `SeqRecord` iterator like we get from `Bio.SeqIO.parse()` – this lets us do file conversion by combining these two functions.

For this example we'll read in the GenBank format file ls_orchid.gbk and write it out in FASTA format:

```
from Bio import SeqIO

records = SeqIO.parse("ls_orchid.gbk", "genbank")
count = SeqIO.write(records, "my_example.fasta", "fasta")
print("Converted %i records" % count)
```

Still, that is a little bit complicated. So, because file conversion is such a common task, there is a helper function letting you replace that with just:

```
from Bio import SeqIO

count = SeqIO.convert("ls_orchid.gbk", "genbank", "my_example.fasta", "fasta")
print("Converted %i records" % count)
```

The `Bio.SeqIO.convert()` function will take handles *or* filenames. Watch out though – if the output file already exists, it will overwrite it! To find out more, see the built-in help:

```
>>> from Bio import SeqIO
>>> help(SeqIO.convert)
```

In principle, just by changing the filenames and the format names, this code could be used to convert between any file formats available in Biopython. However, writing some formats requires information (e.g. quality scores) which other files formats don't contain. For example, while you can turn a FASTQ file into a FASTA file, you can't do the reverse. See also Sections *Converting FASTQ files* and *Converting FASTA and QUAL files into FASTQ files* in the cookbook chapter which looks at inter-converting between different FASTQ formats.

Finally, as an added incentive for using the `Bio.SeqIO.convert()` function (on top of the fact your code will be shorter), doing it this way may also be faster! The reason for this is the convert function can take advantage of several file format specific optimizations and tricks.

5.5.3 Converting a file of sequences to their reverse complements

Suppose you had a file of nucleotide sequences, and you wanted to turn it into a file containing their reverse complement sequences. This time a little bit of work is required to transform the `SeqRecord` objects we get from our input file into something suitable for saving to our output file.

To start with, we'll use `Bio.SeqIO.parse()` to load some nucleotide sequences from a file, then print out their reverse complements using the `Seq` object's built-in `.reverse_complement()` method (see Section *Nucleotide sequences and (reverse) complements*):

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("ls_orchid.gbk", "genbank"):
...     print(record.id)
...     print(record.seq.reverse_complement())
...
```

Now, if we want to save these reverse complements to a file, we'll need to make `SeqRecord` objects. We can use the `SeqRecord` object's built-in `.reverse_complement()` method (see Section *Reverse-complementing SeqRecord objects*) but we must decide how to name our new records.

This is an excellent place to demonstrate the power of list comprehensions which make a list in memory:

```
>>> from Bio import SeqIO
>>> records = [
...     rec.reverse_complement(id="rc_" + rec.id, description="reverse complement")
...     for rec in SeqIO.parse("ls_orchid.fasta", "fasta")
... ]
>>> len(records)
94
```

Now list comprehensions have a nice trick up their sleeves, you can add a conditional statement:

```
>>> records = [
...     rec.reverse_complement(id="rc_" + rec.id, description="reverse complement")
...     for rec in SeqIO.parse("ls_orchid.fasta", "fasta")
...     if len(rec) < 700
... ]
>>> len(records)
18
```

That would create an in memory list of reverse complement records where the sequence length was under 700 base pairs. However, we can do exactly the same with a generator expression - but with the advantage that this does not create a list of all the records in memory at once:

```
>>> records = (
...     rec.reverse_complement(id="rc_" + rec.id, description="reverse complement")
...     for rec in SeqIO.parse("ls_orchid.fasta", "fasta")
...     if len(rec) < 700
... )
```

As a complete example:

```
>>> from Bio import SeqIO
>>> records = (
...     rec.reverse_complement(id="rc_" + rec.id, description="reverse complement")
...     for rec in SeqIO.parse("ls_orchid.fasta", "fasta")
...     if len(rec) < 700
... )
>>> SeqIO.write(records, "rev_comp.fasta", "fasta")
18
```

There is a related example in Section *Translating a FASTA file of CDS entries*, translating each record in a FASTA file from nucleotides to amino acids.

5.5.4 Getting your SeqRecord objects as formatted strings

Suppose that you don't really want to write your records to a file or handle – instead you want a string containing the records in a particular file format. The `Bio.SeqIO` interface is based on handles, but Python has a useful built-in module which provides a string based handle.

For an example of how you might use this, let's load in a bunch of `SeqRecord` objects from our orchids GenBank file, and create a string containing the records in FASTA format:

```
from Bio import SeqIO
from io import StringIO

records = SeqIO.parse("ls_orchid.gbk", "genbank")
out_handle = StringIO()
SeqIO.write(records, out_handle, "fasta")
fasta_data = out_handle.getvalue()
print(fasta_data)
```

This isn't entirely straightforward the first time you see it! On the bright side, for the special case where you would like a string containing a *single* record in a particular file format, use the the `SeqRecord` class' `format()` method (see Section *The format method*).

Note that although we don't encourage it, you *can* use the `format()` method to write to a file, for example something like this:

```
from Bio import SeqIO

with open("ls_orchid_long.tab", "w") as out_handle:
    for record in SeqIO.parse("ls_orchid.gbk", "genbank"):
        if len(record) > 100:
            out_handle.write(record.format("tab"))
```

While this style of code will work for a simple sequential file format like FASTA or the simple tab separated format used here, it will *not* work for more complex or interlaced file formats. This is why we still recommend using `Bio.SeqIO.write()`, as in the following example:

```
from Bio import SeqIO

records = (rec for rec in SeqIO.parse("ls_orchid.gbk", "genbank") if len(rec) > 100)
SeqIO.write(records, "ls_orchid.tab", "tab")
```

Making a single call to `SeqIO.write(...)` is also much quicker than multiple calls to the `SeqRecord.format(...)` method.

5.6 Low level FASTA and FASTQ parsers

Working with the low-level `SimpleFastaParser` or `FastqGeneralIterator` is often more practical than `Bio.SeqIO.parse` when dealing with large high-throughput FASTA or FASTQ sequencing files where speed matters. As noted in the introduction to this chapter, the file-format neutral `Bio.SeqIO` interface has the overhead of creating many objects even for simple formats like FASTA.

When parsing FASTA files, internally `Bio.SeqIO.parse()` calls the low-level `SimpleFastaParser` with the file handle. You can use this directly - it iterates over the file handle returning each record as a tuple of two strings, the title line (everything after the > character) and the sequence (as a plain string):

```
>>> from Bio.SeqIO.FastaIO import SimpleFastaParser
>>> count = 0
>>> total_len = 0
>>> with open("ls_orchid.fasta") as in_handle:
...     for title, seq in SimpleFastaParser(in_handle):
...         count += 1
...         total_len += len(seq)
...
>>> print("%i records with total sequence length %i" % (count, total_len))
94 records with total sequence length 67518
```

As long as you don't care about line wrapping (and you probably don't for short read high-throughput data), then outputting FASTA format from these strings is also very fast:

```
...
out_handle.write(">%s\n%s\n" % (title, seq))
...
```

Likewise, when parsing FASTQ files, internally `Bio.SeqIO.parse()` calls the low-level `FastqGeneralIterator` with the file handle. If you don't need the quality scores turned into integers, or can work with them as ASCII strings this is ideal:

```
>>> from Bio.SeqIO.QualityIO import FastqGeneralIterator
>>> count = 0
>>> total_len = 0
>>> with open("example.fastq") as in_handle:
...     for title, seq, qual in FastqGeneralIterator(in_handle):
...         count += 1
...         total_len += len(seq)
...
>>> print("%i records with total sequence length %i" % (count, total_len))
3 records with total sequence length 75
```

There are more examples of this in the Cookbook (Chapter *Cookbook – Cool things to do with it*), including how to output FASTQ efficiently from strings using this code snippet:

```
...
out_handle.write("@%s\n%s\n+\n%s\n" % (title, seq, qual))
...
```

CHAPTER

SIX

SEQUENCE ALIGNMENTS

Sequence alignments are a collection of two or more sequences that have been aligned to each other – usually with the insertion of gaps, and the addition of leading or trailing gaps – such that all the sequence strings are the same length.

Alignments may extend over the full length of each sequence, or may be limited to a subsection of each sequence. In Biopython, all sequence alignments are represented by an `Alignment` object, described in section *Alignment objects*. `Alignment` objects can be obtained by parsing the output of alignment software such as Clustal or BLAT (described in section *Reading and writing alignments*. or by using Biopython's pairwise sequence aligner, which can align two sequences to each other (described in Chapter *Pairwise sequence alignment*).

See Chapter *Multiple Sequence Alignment objects* for a description of the older `MultipleSeqAlignment` class and the parsers in `Bio.AlignIO` that parse the output of sequence alignment software, generating `MultipleSeqAlignment` objects.

6.1 Alignment objects

The `Alignment` class is defined in `Bio.Align`. Usually you would get an `Alignment` object by parsing the output of alignment programs (section *Reading and writing alignments*) or by running Biopython's pairwise aligner (Chapter *Pairwise sequence alignment*). For the benefit of this section, however, we will create an `Alignment` object from scratch.

6.1.1 Creating an Alignment object from sequences and coordinates

Suppose you have three sequences:

```
>>> seqA = "CCGGTTTTT"
>>> seqB = "AGTTTAA"
>>> seqC = "AGGTTT"
>>> sequences = [seqA, seqB, seqC]
```

To create an `Alignment` object, we also need the coordinates that define how the sequences are aligned to each other. We use a NumPy array for that:

```
>>> import numpy as np
>>> coordinates = np.array([[1, 3, 4, 7, 9], [0, 2, 2, 5, 5], [0, 2, 3, 6, 6]])
```

These coordinates define the alignment for the following sequence segments:

- `SeqA[1:3]`, `SeqB[0:2]`, and `SeqC[0:2]` are aligned to each other;
- `SeqA[3:4]` and `SeqC[2:3]` are aligned to each other, with a gap of one nucleotide in `seqB`;

- `SeqA[4:7]`, `SeqB[2:5]`, and `SeqC[3:6]` are aligned to each other;
- `SeqA[7:9]` is not aligned to `seqB` or `seqC`.

Note that the alignment does not include the first nucleotide of `seqA` and last two nucleotides of `seqB`.

Now we can create the `Alignment` object:

```
>>> from Bio.Align import Alignment
>>> alignment = Alignment(sequences, coordinates)
>>> alignment
<Alignment object (3 rows x 8 columns) at ...>
```

The alignment object has an attribute `sequences` pointing to the sequences included in this alignment:

```
>>> alignment.sequences
['CCGGTTTTT', 'AGTTTAA', 'AGGTTT']
```

and an attribute `coordinates` with the alignment coordinates:

```
>>> alignment.coordinates
array([[1, 3, 4, 7, 9],
       [0, 2, 2, 5, 5],
       [0, 2, 3, 6, 6]])
```

Print the `Alignment` object to show the alignment explicitly:

```
>>> print(alignment)
                  1 CGGTTTTT 9
                  0 AG-TTT-- 5
                  0 AGGTTT-- 6
```

with the starting and end coordinate for each sequence are shown to the left and right, respectively, of the alignment.

6.1.2 Creating an Alignment object from aligned sequences

If you start out with the aligned sequences, with dashes representing gaps, then you can calculate the coordinates using the `parse_printed_alignment` class method. This method is primarily employed in Biopython's alignment parsers (see Section *Reading and writing alignments*), but it may be useful for other purposes. For example, you can construct the `Alignment` object from aligned sequences as follows:

```
>>> lines = ["CGGTTTTT", "AG-TTT--", "AGGTTT--"]
>>> for line in lines:
...     print(line)
...
CGGTTTTT
AG-TTT--
AGGTTT--
>>> lines = [line.encode() for line in lines]  # convert to bytes
>>> lines
[b'CGGTTTTT', b'AG-TTT--', b'AGGTTT--']
>>> sequences, coordinates = Alignment.parse_printed_alignment(lines)
>>> sequences
[b'CGGTTTTT', b'AGTTT', b'AGGTTT']
>>> sequences = [sequence.decode() for sequence in sequences]
```

(continues on next page)

(continued from previous page)

```
>>> sequences
['CGGTTTTT', 'AGTTT', 'AGGTTT']
>>> print(coordinates)
[[0 2 3 6 8]
 [0 2 2 5 5]
 [0 2 3 6 6]]
```

The initial `G` nucleotide of `seqA` and the final `CC` nucleotides of `seqB` were not included in the alignment and is therefore missing here. But this is easy to fix:

```
>>> from Bio.Seq import Seq
>>> sequences[0] = "C" + sequences[0]
>>> sequences[1] = sequences[1] + "AA"
>>> sequences
['CCGGTTTTT', 'AGTTTAA', 'AGGTTT']
>>> coordinates[0, :] += 1
>>> print(coordinates)
[[1 3 4 7 9]
 [0 2 2 5 5]
 [0 2 3 6 6]]
```

Now we can create the `Alignment` object:

```
>>> alignment = Alignment(sequences, coordinates)
>>> print(alignment)
                  1 CGGTTTTT 9
                  0 AG-TTT-- 5
                  0 AGGTTT-- 6
```

which identical to the `Alignment` object created above in section *Creating an Alignment object from sequences and coordinates*.

By default, the `coordinates` argument to the `Alignment` initializer is `None`, which assumes that there are no gaps in the alignment. All sequences in an ungapped alignment must have the same length. If the `coordinates` argument is `None`, then the initializer will fill in the `coordinates` attribute of the `Alignment` object for you:

```
>>> ungapped_alignment = Alignment(["ACGTACGT", "AAGTACGT", "ACGTACCT"])
>>> ungapped_alignment
<Alignment object (3 rows x 8 columns) at ...>
>>> print(ungapped_alignment.coordinates)
[[0 8]
 [0 8]
 [0 8]]
>>> print(ungapped_alignment)
                  0 ACGTACGT 8
                  0 AAGTACGT 8
                  0 ACGTACCT 8
```

6.1.3 Common alignment attributes

The following attributes are commonly found on `Alignment` objects:

- `sequences`: This is a list of the sequences aligned to each other. Depending on how the alignment was created, the sequences can have the following types:
  - plain Python string;
  - `Seq`;
  - `MutableSeq`;
  - `SeqRecord`;
  - `bytes`;
  - `bytearray`;
  - NumPy array with data type `numpy.int32`;
  - any other object with a contiguous buffer of format `"c"`, `"B"`, `"i"`, or `"I"`;
  - lists or tuples of objects defined in the `alphabet` attribute of the `PairwiseAligner` object that created the alignment (see section *Generalized pairwise alignments*).

  For pairwise alignments (meaning an alignment of two sequences), the properties `target` and `query` are aliases for `sequences[0]` and `sequences[1]`, respectively.
- `coordinates`: A NumPy array of integers storing the sequence indices defining how the sequences are aligned to each other;
- `score`: The alignment score, as found by the parser in the alignment file, or as calculated by the `PairwiseAligner` (see section *Basic usage*);
- `annotations`: A dictionary storing most other annotations associated with the alignment;
- `column_annotations`: A dictionary storing annotations that extend along the alignment and have the same length as the alignment, such as a consensus sequence (see section *ClustalW* for an example).

An `Alignment` object created by the parser in `Bio.Align` may have additional attributes, depending on the alignment file format from which the alignment was read.

6.2 Slicing and indexing an alignment

Slices of the form `alignment[k, i:j]`, where `k` is an integer and `i` and `j` are integers or are absent, return a string showing the aligned sequence (including gaps) for the target (if `k=0`) or the query (if `k=1`) that includes only the columns `i` through `j` in the printed alignment.

To illustrate this, in the following example the printed alignment has 8 columns:

```
>>> print(alignment)
                  1 CGGTTTTT 9
                  0 AG-TTT-- 5
                  0 AGGTTT-- 6

>>> alignment.length
8
```

To get the aligned sequence strings individually, use

```
>>> alignment[0]
'CGGTTTTT'
>>> alignment[1]
'AG-TTT--'
>>> alignment[2]
'AGGTTT--'
>>> alignment[0, :]
'CGGTTTTT'
>>> alignment[1, :]
'AG-TTT--'
>>> alignment[0, 1:-1]
'GGTTTT'
>>> alignment[1, 1:-1]
'G-TTT-'
```

Columns to be included can also be selected using an iterable over integers:

```
>>> alignment[0, (1, 2, 4)]
'GGT'
>>> alignment[1, range(0, 5, 2)]
'A-T'
```

To get the letter at position `[i, j]` of the printed alignment, use `alignment[i, j]`; this will return `"-"` if a gap is found at that position:

```
>>> alignment[0, 2]
'G'
>>> alignment[2, 6]
'-'
```

To get specific columns in the alignment, use

```
>>> alignment[:, 0]
'CAA'
>>> alignment[:, 1]
'GGG'
>>> alignment[:, 2]
'G-G'
```

Slices of the form `alignment[i:j:k]` return a new `Alignment` object including only sequences `[i:j:k]` of the alignment:

```
>>> alignment[1:]
<Alignment object (2 rows x 6 columns) at ...>
>>> print(alignment[1:])
target            0 AG-TTT 5
                  0 ||-||| 6
query             0 AGGTTT 6
```

Slices of the form `alignment[:, i:j]`, where `i` and `j` are integers or are absent, return a new `Alignment` object that includes only the columns `i` through `j` in the printed alignment.

Extracting the first 4 columns for the example alignment above gives:

```
>>> alignment[:, :4]
<Alignment object (3 rows x 4 columns) at ...>
>>> print(alignment[:, :4])
                   1 CGGT 5
                   0 AG-T 3
                   0 AGGT 4
```

Similarly, extracting the last 6 columns gives:

```
>>> alignment[:, -6:]
<Alignment object (3 rows x 6 columns) at ...>
>>> print(alignment[:, -6:])
                   3 GTTTTT 9
                   2 -TTT-- 5
                   2 GTTT-- 6
```

The column index can also be an iterable of integers:

```
>>> print(alignment[:, (1, 3, 0)])
                   0 GTC 3
                   0 GTA 3
                   0 GTA 3
```

Calling `alignment[:, :]` returns a copy of the alignment.

6.3 Getting information about the alignment

6.3.1 Alignment shape

The number of aligned sequences is returned by `len(alignment)`:

```
>>> len(alignment)
3
```

The alignment length is defined as the number of columns in the alignment as printed. This is equal to the sum of the number of matches, number of mismatches, and the total length of gaps in each sequence:

```
>>> alignment.length
8
```

The `shape` property returns a tuple consisting of the length of the alignment and the number of columns in the alignment as printed:

```
>>> alignment.shape
(3, 8)
```

6.3.2 Comparing alignments

Two alignments are equal to each other (meaning that `alignment1 == alignment2` evaluates to `True`) if each of the sequences in `alignment1.sequences` and `alignment2.sequences` are equal to each other, and `alignment1.coordinates` and `alignment2.coordinates` contain the same coordinates. If either of these conditions is not fulfilled, then `alignment1 == alignment2` evaluates to `False`. Inequality of two alignments (e.g., `alignment1 < alignment2`) is established by first comparing `alignment1.sequences` and `alignment2.sequences`, and if they are equal, by comparing `alignment1.coordinates` to `alignment2.coordinates`.

6.3.3 Finding the indices of aligned sequences

For pairwise alignments, the `aligned` property of an alignment returns the start and end indices of subsequences in the target and query sequence that were aligned to each other. If the alignment between target (t) and query (q) consists of N chunks, you get two tuples of length N:

```
(((t_start1, t_end1), (t_start2, t_end2), ..., (t_startN, t_endN)),
 ((q_start1, q_end1), (q_start2, q_end2), ..., (q_startN, q_endN)))
```

For example,

```
>>> pairwise_alignment = alignment[:2, :]
>>> print(pairwise_alignment)
target            1 CGGTTTTT 9
                  0 .|-|||-- 8
query             0 AG-TTT-- 5

>>> print(pairwise_alignment.aligned)
[[[1 3]
  [4 7]]

 [[0 2]
  [2 5]]]
```

Note that different alignments may have the same subsequences aligned to each other. In particular, this may occur if alignments differ from each other in terms of their gap placement only:

```
>>> pairwise_alignment1 = Alignment(["AAACAAA", "AAAGAAA"],
...                                 np.array([[0, 3, 4, 4, 7], [0, 3, 3, 4, 7]]))  #␣
↪fmt: skip
...
>>> pairwise_alignment2 = Alignment(["AAACAAA", "AAAGAAA"],
...                                 np.array([[0, 3, 3, 4, 7], [0, 3, 4, 4, 7]]))  #␣
↪fmt: skip
...
>>> print(pairwise_alignment1)
target            0 AAAC-AAA 7
                  0 |||--||| 8
query             0 AAA-GAAA 7

>>> print(pairwise_alignment2)
target            0 AAA-CAAA 7
                  0 |||--||| 8
query             0 AAAG-AAA 7
```

(continues on next page)

(continued from previous page)

```
>>> pairwise_alignment1.aligned
array([[[0, 3],
        [4, 7]],

       [[0, 3],
        [4, 7]]])
>>> pairwise_alignment2.aligned
array([[[0, 3],
        [4, 7]],

       [[0, 3],
        [4, 7]]])
```

The property `indices` returns a 2D NumPy array with the sequence index of each letter in the alignment, with gaps indicated by -1:

```
>>> print(alignment)
                  1 CGGTTTTT 9
                  0 AG-TTT-- 5
                  0 AGGTTT-- 6

>>> alignment.indices
array([[ 1,  2,  3,  4,  5,  6,  7,  8],
       [ 0,  1, -1,  2,  3,  4, -1, -1],
       [ 0,  1,  2,  3,  4,  5, -1, -1]])
```

The property `inverse_indices` returns a list of 1D NumPy arrays, one for each of the aligned sequences, with the column index in the alignment for each letter in the sequence. Letters not included in the alignment are indicated by -1:

```
>>> alignment.sequences
['CCGGTTTTT', 'AGTTTAA', 'AGGTTT']
>>> alignment.inverse_indices
[array([-1,  0,  1,  2,  3,  4,  5,  6,  7]),
 array([ 0,  1,  3,  4,  5, -1, -1]),
 array([0, 1, 2, 3, 4, 5])]
```

6.3.4 Counting identities, mismatches, and gaps

The `counts` method calculates the number of identities, mismatches, and gaps of a pairwise alignment. For an alignment of more than two sequences, the number of identities, mismatches, and gaps are calculated and summed for all pairs of sequences in the alignment. The three numbers are returned as an `AlignmentCounts` object, which is a `namedtuple` with fields `gaps`, `identities`, and `mismatches`. This method currently takes no arguments, but in the future will likely be modified to accept optional arguments allowing its behavior to be customized.

```
>>> print(pairwise_alignment)
target            1 CGGTTTTT 9
                  0 .|-|||-- 8
query             0 AG-TTT-- 5
```

(continues on next page)

(continued from previous page)

```
>>> pairwise_alignment.counts()
AlignmentCounts(gaps=3, identities=4, mismatches=1)
>>> print(alignment)
                  1 CGGTTTTT 9
                  0 AG-TTT-- 5
                  0 AGGTTT-- 6

>>> alignment.counts()
AlignmentCounts(gaps=8, identities=14, mismatches=2)
```

6.3.5 Letter frequencies

The `frequencies` method calculates how often each letter appears in each column of the alignment:

```
>>> alignment.frequencies
{'C': array([1., 0., 0., 0., 0., 0., 0., 0.]),
 'G': array([0., 3., 2., 0., 0., 0., 0., 0.]),
 'T': array([0., 0., 0., 3., 3., 3., 1., 1.]),
 'A': array([2., 0., 0., 0., 0., 0., 0., 0.]),
 '-': array([0., 0., 1., 0., 0., 0., 2., 2.])}
```

6.3.6 Substitutions

Use the `substitutions` method to find the number of substitutions between each pair of nucleotides:

```
>>> m = alignment.substitutions
>>> print(m)
    A   C   G   T
A 1.0 0.0 0.0 0.0
C 2.0 0.0 0.0 0.0
G 0.0 0.0 4.0 0.0
T 0.0 0.0 0.0 9.0
```

Note that the matrix is not symmetric: The counts for a row letter R and a column letter C is the number of times letter R in a sequence is replaced by letter C in a sequence appearing below it. For example, the number of `C`'s that are aligned to an `A` in a later sequence is

```
>>> m["C", "A"]
2.0
```

while the number of A's that are aligned to a C in a later sequence is

```
>>> m["A", "C"]
0.0
```

To get a symmetric matrix, use

```
>>> m += m.transpose()
>>> m /= 2.0
>>> print(m)
```

(continues on next page)

(continued from previous page)

```
    A   C   G   T
A 1.0 1.0 0.0 0.0
C 1.0 0.0 0.0 0.0
G 0.0 0.0 4.0 0.0
T 0.0 0.0 0.0 9.0

>>> m["A", "C"]
1.0
>>> m["C", "A"]
1.0
```

The total number of substitutions between `A`'s and `T`'s in the alignment is 1.0 + 1.0 = 2.

6.3.7 Alignments as arrays

Using NumPy, you can turn the `alignment` object into an array of letters. In particular, this may be useful for fast calculations on the alignment content.

```
>>> align_array = np.array(alignment)
>>> align_array.shape
(3, 8)
>>> align_array
array([[b'C', b'G', b'G', b'T', b'T', b'T', b'T', b'T'],
       [b'A', b'G', b'-', b'T', b'T', b'T', b'-', b'-'],
       [b'A', b'G', b'G', b'T', b'T', b'T', b'-', b'-']], dtype='|S1')
```

By default, this will give you an array of `bytes` characters (with data type `dtype='|S1'`). You can create an array of Unicode (Python string) characters by using `dtype='U'`:

```
>>> align_array = np.array(alignment, dtype="U")
```

```
>>> align_array
array([['C', 'G', 'G', 'T', 'T', 'T', 'T', 'T'],
       ['A', 'G', '-', 'T', 'T', 'T', '-', '-'],
       ['A', 'G', 'G', 'T', 'T', 'T', '-', '-']], dtype='<U1')
```

(the printed `dtype` will be '<U1' or '>U1' depending on whether your system is little-endian or big-endian, respectively). Note that the `alignment` object and the NumPy array `align_array` are separate objects in memory - editing one will not update the other!

6.4 Operations on an alignment

6.4.1 Sorting an alignment

The `sort` method sorts the alignment sequences. By default, sorting is done based on the `id` attribute of each sequence if available, or the sequence contents otherwise.

```
>>> print(alignment)
                  1 CGGTTTTT 9
```

(continues on next page)

(continued from previous page)

```
                  0 AG-TTT-- 5
                  0 AGGTTT-- 6

>>> alignment.sort()
>>> print(alignment)
                  0 AGGTTT-- 6
                  0 AG-TTT-- 5
                  1 CGGTTTTT 9
```

Alternatively, you can supply a `key` function to determine the sort order. For example, you can sort the sequences by increasing GC content:

```
>>> from Bio.SeqUtils import gc_fraction
>>> alignment.sort(key=gc_fraction)
>>> print(alignment)
                  0 AG-TTT-- 5
                  0 AGGTTT-- 6
                  1 CGGTTTTT 9
```

Note that the `key` function is applied to the full sequence (including the initial `A` and final `GG` nucleotides of `seqB`), not just to the aligned part.

The `reverse` argument lets you reverse the sort order to obtain the sequences in decreasing GC content:

```
>>> alignment.sort(key=gc_fraction, reverse=True)
>>> print(alignment)
                  1 CGGTTTTT 9
                  0 AGGTTT-- 6
                  0 AG-TTT-- 5
```

6.4.2 Reverse-complementing the alignment

Reverse-complementing an alignment will take the reverse complement of each sequence, and recalculate the coordinates:

```
>>> alignment.sequences
['CCGGTTTTT', 'AGGTTT', 'AGTTTAA']
>>> rc_alignment = alignment.reverse_complement()
>>> print(rc_alignment.sequences)
['AAAAACCGG', 'AAACCT', 'TTAAACT']
>>> print(rc_alignment)
                  0 AAAAACCG 8
                  0 --AAACCT 6
                  2 --AAA-CT 7

>>> alignment[:, :4].sequences
['CCGGTTTTT', 'AGGTTT', 'AGTTTAA']
>>> print(alignment[:, :4])
                  1 CGGT 5
                  0 AGGT 4
                  0 AG-T 3
```

(continues on next page)

(continued from previous page)

```
>>> rc_alignment = alignment[:, :4].reverse_complement()
>>> rc_alignment[:, :4].sequences
['AAAAACCGG', 'AAACCT', 'TTAAACT']
>>> print(rc_alignment[:, :4])
                  4 ACCG 8
                  2 ACCT 6
                  4 A-CT 7
```

Reverse-complementing an alignment preserves its column annotations (in reverse order), but discards all other annotations.

6.4.3 Adding alignments

Alignments can be added together to form an extended alignment if they have the same number of rows. As an example, let's first create two alignments:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> a1 = SeqRecord(Seq("AAAAC"), id="Alpha")
>>> b1 = SeqRecord(Seq("AAAC"), id="Beta")
>>> c1 = SeqRecord(Seq("AAAAG"), id="Gamma")
>>> a2 = SeqRecord(Seq("GTT"), id="Alpha")
>>> b2 = SeqRecord(Seq("TT"), id="Beta")
>>> c2 = SeqRecord(Seq("GT"), id="Gamma")
>>> left = Alignment(
...     [a1, b1, c1], coordinates=np.array([[0, 3, 4, 5], [0, 3, 3, 4], [0, 3, 4, 5]])
... )
>>> left.annotations = {"tool": "demo", "name": "start"}
>>> left.column_annotations = {"stats": "CCCXC"}
>>> right = Alignment(
...     [a2, b2, c2], coordinates=np.array([[0, 1, 2, 3], [0, 0, 1, 2], [0, 1, 1, 2]])
... )
>>> right.annotations = {"tool": "demo", "name": "end"}
>>> right.column_annotations = {"stats": "CXC"}
```

Now, let's look at these two alignments:

```
>>> print(left)
Alpha             0 AAAAC 5
Beta              0 AAA-C 4
Gamma             0 AAAAG 5

>>> print(right)
Alpha             0 GTT 3
Beta              0 -TT 2
Gamma             0 G-T 2
```

Adding the two alignments will combine the two alignments row-wise:

```
>>> combined = left + right
>>> print(combined)
Alpha             0 AAAACGTT 8
```

(continues on next page)

(continued from previous page)

```
Beta              0 AAA-C-TT 6
Gamma             0 AAAAGG-T 7
```

For this to work, both alignments must have the same number of sequences (here they both have 3 rows):

```
>>> len(left)
3
>>> len(right)
3
>>> len(combined)
3
```

The sequences are `SeqRecord` objects, which can be added together. Refer to Chapter *Sequence annotation objects* for details of how the annotation is handled. This example is a special case in that both original alignments shared the same names, meaning when the rows are added they also get the same name.

Any common annotations are preserved, but differing annotation is lost. This is the same behavior used in the `SeqRecord` annotations and is designed to prevent accidental propagation of inappropriate values:

```
>>> combined.annotations
{'tool': 'demo'}
```

Similarly any common per-column-annotations are combined:

```
>>> combined.column_annotations
{'stats': 'CCCXCCXC'}
```

6.4.4 Mapping a pairwise sequence alignment

Suppose you have a pairwise alignment of a transcript to a chromosome:

```
>>> chromosome = "AAAAAAAACCCCCCCAAAAAAAAAAAGGGGGGAAAAAAAA"
>>> transcript = "CCCCCCCGGGGGG"
>>> sequences1 = [chromosome, transcript]
>>> coordinates1 = np.array([[8, 15, 26, 32], [0, 7, 7, 13]])
>>> alignment1 = Alignment(sequences1, coordinates1)
>>> print(alignment1)
target            8 CCCCCCCAAAAAAAAAAAGGGGGG 32
                  0 |||||||-----------|||||| 24
query             0 CCCCCCC-----------GGGGGG 13
```

and a pairwise alignment between the transcript and a sequence (e.g., obtained by RNA-seq):

```
>>> rnaseq = "CCCCGGGG"
>>> sequences2 = [transcript, rnaseq]
>>> coordinates2 = np.array([[3, 11], [0, 8]])
>>> alignment2 = Alignment(sequences2, coordinates2)
>>> print(alignment2)
target            3 CCCCGGGG 11
                  0 ||||||||  8
query             0 CCCCGGGG  8
```

Use the `map` method on `alignment1`, with `alignment2` as argument, to find the alignment of the RNA-sequence to the genome:

```
>>> alignment3 = alignment1.map(alignment2)
>>> print(alignment3)
target            11 CCCCAAAAAAAAAAAGGGG 30
                   0 ||||-----------|||| 19
query              0 CCCC-----------GGGG  8

>>> print(alignment3.coordinates)
[[11 15 26 30]
 [ 0  4  4  8]]
>>> format(alignment3, "psl")
'8\t0\t0\t0\t0\t0\t1\t11\t+\tquery\t8\t0\t8\ttarget\t40\t11\t30\t2\t4,4,\t0,4,\t11,26,\n'
```

To be able to print the sequences, in this example we constructed `alignment1` and `alignment2` using sequences with a defined sequence contents. However, mapping the alignment does not depend on the sequence contents; only the coordinates of `alignment1` and `alignment2` are used to construct the coordinates for `alignment3`.

The map method can also be used to lift over an alignment between different genome assemblies. In this case, self is a DNA alignment between two genome assemblies, and the argument is an alignment of a transcript against one of the genome assemblies:

```
>>> from Bio import Align
>>> chain = Align.read("Blat/panTro5ToPanTro6.over.chain", "chain")
>>> chain.sequences[0].id
'chr1'
>>> len(chain.sequences[0].seq)
228573443
>>> chain.sequences[1].id
'chr1'
>>> len(chain.sequences[1].seq)
224244399
>>> import numpy as np
>>> np.set_printoptions(threshold=5)  # print 5 array elements per row
>>> print(chain.coordinates)
[[122250000 122250400 122250400 ... 122909818 122909819 122909835]
 [111776384 111776784 111776785 ... 112019962 112019962 112019978]]
```

showing that the range 122250000:122909835 of chr1 on chimpanzee genome assembly panTro5 aligns to range 111776384:112019978 of chr1 of chimpanzee genome assembly panTro6. See section *UCSC chain file format* for more information about the chain file format.

```
>>> transcript = Align.read("Blat/est.panTro5.psl", "psl")
>>> transcript.sequences[0].id
'chr1'
>>> len(transcript.sequences[0].seq)
228573443
>>> transcript.sequences[1].id
'DC525629'
>>> len(transcript.sequences[1].seq)
407
>>> print(transcript.coordinates)
[[122835789 122835847 122840993 122841145 122907212 122907314]
 [       32        90        90       242       242       344]]
```

This shows that nucleotide range 32:344 of expressed sequence tag DC525629 aligns to range 122835789:122907314 of chr1 of chimpanzee genome assembly panTro5. Note that the target sequence `chain.sequences[0].seq` and the target sequence `transcript.sequences[0]` have the same length:

```
>>> len(chain.sequences[0].seq) == len(transcript.sequences[0].seq)
True
```

We swap the target and query of the chain such that the query of `chain` corresponds to the target of `transcript`:

```
>>> chain = chain[::-1]
>>> chain.sequences[0].id
'chr1'
>>> len(chain.sequences[0].seq)
224244399
>>> chain.sequences[1].id
'chr1'
>>> len(chain.sequences[1].seq)
228573443
>>> print(chain.coordinates)
[[111776384 111776784 111776785 ... 112019962 112019962 112019978]
 [122250000 122250400 122250400 ... 122909818 122909819 122909835]]
>>> np.set_printoptions(threshold=1000)  # reset the print options
```

Now we can get the coordinates of DC525629 against chimpanzee genome assembly panTro6 by calling `chain.map`, with `transcript` as the argument:

```
>>> lifted_transcript = chain.map(transcript)
>>> lifted_transcript.sequences[0].id
'chr1'
>>> len(lifted_transcript.sequences[0].seq)
224244399
>>> lifted_transcript.sequences[1].id
'DC525629'
>>> len(lifted_transcript.sequences[1].seq)
407
>>> print(lifted_transcript.coordinates)
[[111982717 111982775 111987921 111988073 112009200 112009302]
 [       32        90        90       242       242       344]]
```

This shows that nucleotide range 32:344 of expressed sequence tag DC525629 aligns to range 111982717:112009302 of chr1 of chimpanzee genome assembly panTro6. Note that the genome span of DC525629 on chimpanzee genome assembly panTro5 is 122907314 - 122835789 = 71525 bp, while on panTro6 the genome span is 112009302 - 111982717 = 26585 bp.

6.4.5 Mapping a multiple sequence alignment

Consider a multiple alignment of genomic sequences of chimpanzee, human, macaque, marmoset, mouse, and rat:

```
>>> from Bio import Align
>>> path = "Blat/panTro5.maf"
>>> genome_alignment = Align.read(path, "maf")
>>> for record in genome_alignment.sequences:
...     print(record.id, len(record.seq))
```

(continues on next page)

(continued from previous page)

```
...
panTro5.chr1 228573443
hg19.chr1 249250621
rheMac8.chr1 225584828
calJac3.chr18 47448759
mm10.chr3 160039680
rn6.chr2 266435125
>>> print(genome_alignment.coordinates)
[[133922962 133922962 133922970 133922970 133922972 133922972 133922995
  133922998 133923010]
 [155784573 155784573 155784581 155784581 155784583 155784583 155784606
  155784609 155784621]
 [130383910 130383910 130383918 130383918 130383920 130383920 130383943
  130383946 130383958]
 [  9790455   9790455   9790463   9790463   9790465   9790465   9790488
    9790491   9790503]
 [ 88858039  88858036  88858028  88858026  88858024  88858020  88857997
   88857997  88857985]
 [188162970 188162967 188162959 188162959 188162957 188162953 188162930
  188162930 188162918]]
>>> print(genome_alignment)
panTro5.c 133922962 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAACTTAAAggcc
hg19.chr1 155784573 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAACTTAAAggcc
rheMac8.c 130383910 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAACTTAAAggcc
calJac3.c   9790455 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAGCTTAAAggct
mm10.chr3  88858039 TATAATAATTGTATATGTCACAGAAAAAAATGAATTTTCAAT---GACTTAATAGCC
rn6.chr2  188162970 TACAATAATTG--TATGTCATAGAAAAAAATGAATTTTCAAT---AACTTAATAGCC

panTro5.c 133923010
hg19.chr1 155784621
rheMac8.c 130383958
calJac3.c   9790503
mm10.chr3  88857985
rn6.chr2  188162918
```

Suppose we want to replace the older versions of the genome assemblies (`panTro5`, `hg19`, `rheMac8`, `calJac3`, `mm10`, and `rn6`) by their current versions (`panTro6`, `hg38`, `rheMac10`, `calJac4`, `mm39`, and `rn7`). To do so, we need the pairwise alignment between the old and the new assembly version for each species. These are provided by UCSC as chain files, typically used for UCSC's `liftOver` tool. The `.chain` files in the `Tests/Align` subdirectory in the Biopython source distribution were extracted from UCSC's `.chain` files to only include the relevant genomic region. For example, to lift over `panTro5` to `panTro6`, we use the file `panTro5ToPanTro6.chain` with the following contents:

```
chain 1198066 chr1 228573443 + 133919957 133932620 chr1 224244399 + 130607995 130620657 1
4990    0   2
1362    3   0
6308
```

To lift over the genome assembly for each species, we read in the corresponding `.chain` file:

```
>>> paths = [
...     "Blat/panTro5ToPanTro6.chain",
...     "Blat/hg19ToHg38.chain",
```

(continues on next page)

(continued from previous page)

```
...     "Blat/rheMac8ToRheMac10.chain",
...     "Blat/calJac3ToCalJac4.chain",
...     "Blat/mm10ToMm39.chain",
...     "Blat/rn6ToRn7.chain",
... ]
>>> liftover_alignments = [Align.read(path, "chain") for path in paths]
>>> for liftover_alignment in liftover_alignments:
...     print(liftover_alignment.target.id, liftover_alignment.coordinates[0, :])
...
chr1 [133919957 133924947 133924947 133926309 133926312 133932620]
chr1 [155184381 156354347 156354348 157128497 157128497 157137496]
chr1 [130382477 130383872 130383872 130384222 130384222 130388520]
chr18 [9786631 9787941 9788508 9788508 9795062 9795065 9795737]
chr3 [66807541 74196805 74196831 94707528 94707528 94708176 94708178 94708718]
chr2 [188111581 188158351 188158351 188171225 188171225 188228261 188228261
 188236997]
```

Note that the order of species is the same in `liftover_alignments` and `genome_alignment.sequences`. Now we can lift over the multiple sequence alignment to the new genome assembly versions:

```
>>> genome_alignment = genome_alignment.mapall(liftover_alignments)
>>> for record in genome_alignment.sequences:
...     print(record.id, len(record.seq))
...
chr1 224244399
chr1 248956422
chr1 223616942
chr18 47031477
chr3 159745316
chr2 249053267
>>> print(genome_alignment.coordinates)
[[130611000 130611000 130611008 130611008 130611010 130611010 130611033
  130611036 130611048]
 [155814782 155814782 155814790 155814790 155814792 155814792 155814815
  155814818 155814830]
 [ 95186253  95186253  95186245  95186245  95186243  95186243  95186220
   95186217  95186205]
 [  9758318   9758318   9758326   9758326   9758328   9758328   9758351
    9758354   9758366]
 [ 88765346  88765343  88765335  88765333  88765331  88765327  88765304
   88765304  88765292]
 [174256702 174256699 174256691 174256691 174256689 174256685 174256662
  174256662 174256650]]
```

As the `.chain` files do not include the sequence contents, we cannot print the sequence alignment directly. Instead, we read in the genomic sequence separately (as a `.2bit` file, as it allows lazy loading; see section *Sequence files as Dictionaries*) for each species:

```
>>> from Bio import SeqIO
>>> names = ("panTro6", "hg38", "rheMac10", "calJac4", "mm39", "rn7")
>>> for i, name in enumerate(names):
...     filename = f"{name}.2bit"
```

(continues on next page)

(continued from previous page)

```
...     genome = SeqIO.parse(filename, "twobit")
...     chromosome = genome_alignment.sequences[i].id
...     assert len(genome_alignment.sequences[i]) == len(genome[chromosome])
...     genome_alignment.sequences[i] = genome[chromosome]
...     genome_alignment.sequences[i].id = f"{name}.{chromosome}"
...
>>> print(genome_alignment)
panTro6.c 130611000 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAACTTAAAggcc
hg38.chr1 155814782 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAACTTAAAggcc
rheMac10.  95186253 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAACTTAAAggcc
calJac4.c   9758318 ---ACTAGTTA--CA----GTAACAGAaaataaaatttaaatagaagcttaaaggct
mm39.chr3  88765346 TATAATAATTGTATATGTCACAGAAAAAAATGAATTTTCAAT---GACTTAATAGCC
rn7.chr2  174256702 TACAATAATTG--TATGTCATAGAAAAAAATGAATTTTCAAT---AACTTAATAGCC

panTro6.c 130611048
hg38.chr1 155814830
rheMac10.  95186205
calJac4.c   9758366
mm39.chr3  88765292
rn7.chr2  174256650
```

The `mapall` method can also be used to create a multiple alignment of codon sequences from a multiple sequence alignment of the corresponding amino acid sequences (see Section *Generating a multiple sequence alignment of codon sequences* for details).

6.5 The Alignments class

The `Alignments` (plural) class inherits from `AlignmentsAbstractBaseClass` and from `list`, and can be used as a list to store `Alignment` objects. The behavior of `Alignments` objects is different from that of `list` objects in two important ways:

- An `Alignments` object is its own iterator, consistent with iterators returned by `Bio.Align.parse` (see section *Reading alignments*) or iterators returned by the pairwise aligner (see Section *Pairwise sequence alignment*). Calling `iter` on the iterator will always return the `Alignments` object itself. In contrast, calling `iter` on a list object creates a new iterator each time, allowing you to have multiple independent iterators for a given list.

  In this example, `alignment_iterator1` and `alignment_iterator2` are obtained from a list and act independently of each other:

```
>>> alignment_list = [alignment1, alignment2, alignment3]
>>> alignment_iterator1 = iter(alignment_list)
>>> alignment_iterator2 = iter(alignment_list)
>>> next(alignment_iterator1)
<Alignment object (2 rows x 24 columns) at ...>
>>> next(alignment_iterator2)
<Alignment object (2 rows x 24 columns) at ...>
>>> next(alignment_iterator1)
<Alignment object (2 rows x 8 columns) at ...>
>>> next(alignment_iterator1)
<Alignment object (2 rows x 19 columns) at ...>
>>> next(alignment_iterator2)
```

(continues on next page)

(continued from previous page)

```
<Alignment object (2 rows x 8 columns) at ...>
>>> next(alignment_iterator2)
<Alignment object (2 rows x 19 columns) at ...>
```

In contrast, `alignment_iterator1` and `alignment_iterator2` obtained by calling `iter` on an `Alignments` object are identical to each other:

```
>>> from Bio.Align import Alignments
>>> alignments = Alignments([alignment1, alignment2, alignment3])
>>> alignment_iterator1 = iter(alignments)
>>> alignment_iterator2 = iter(alignments)
>>> alignment_iterator1 is alignment_iterator2
True
>>> next(alignment_iterator1)
<Alignment object (2 rows x 24 columns) at ...>
>>> next(alignment_iterator2)
<Alignment object (2 rows x 8 columns) at ...>
>>> next(alignment_iterator1)
<Alignment object (2 rows x 19 columns) at ...>
>>> next(alignment_iterator2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Calling `iter` on an `Alignments` object resets the iterator to its first item, so you can loop over it again. You can also iterate over the alignments multiple times using a `for`-loop, which implicitly calls `iter` on the iterator:

```
>>> for item in alignments:
...     print(repr(item))
...
<Alignment object (2 rows x 24 columns) at ...>
<Alignment object (2 rows x 8 columns) at ...>
<Alignment object (2 rows x 19 columns) at ...>

>>> for item in alignments:
...     print(repr(item))
...
<Alignment object (2 rows x 24 columns) at ...>
<Alignment object (2 rows x 8 columns) at ...>
<Alignment object (2 rows x 19 columns) at ...>
```

This behavior is consistent with regular Python lists, and with iterators returned by `Bio.Align.parse` (see section *Reading alignments*) or by the pairwise aligner (see Section *Pairwise sequence alignment*).

- Metadata can be stored as attributes on an `Alignments` object, whereas a plain `list` does not accept attributes:

```
>>> alignment_list.score = 100
Traceback (most recent call last):
 ...
AttributeError: 'list' object has no attribute 'score'...
>>> alignments.score = 100
>>> alignments.score
100
```

6.6 Reading and writing alignments

Output from sequence alignment software such as Clustal can be parsed into `Alignment` objects by the `Bio.Align.read` and `Bio.Align.parse` functions. Their usage is analogous to the `read` and `parse` functions in `Bio.SeqIO` (see Section *Parsing or Reading Sequences*): The `read` function is used to read an output file containing a single alignment and returns an `Alignment` object, while the `parse` function returns an iterator to iterate over alignments stored in an output file containing one or more alignments. Section *Alignment file formats* describes the alignment formats that can be parsed in `Bio.Align`. `Bio.Align` also provides a `write` function that can write alignments in most of these formats.

6.6.1 Reading alignments

Use `Bio.Align.parse` to parse a file of sequence alignments. For example, the file `ucsc_mm9_chr10.maf` contains 48 multiple sequence alignments in the MAF (Multiple Alignment Format) format (see section *Multiple Alignment Format (MAF)*):

```
>>> from Bio import Align
>>> alignments = Align.parse("MAF/ucsc_mm9_chr10.maf", "maf")
>>> alignments
<Bio.Align.maf.AlignmentIterator object at 0x...>
```

where `"maf"` is the file format. The alignments object returned by `Bio.Align.parse` may contain attributes that store metadata found in the file, such as the version number of the software that was used to create the alignments. The specific attributes stored for each file format are described in Section *Alignment file formats*. For MAF files, we can obtain the file format version and the scoring scheme that was used:

```
>>> alignments.metadata
{'MAF Version': '1', 'Scoring': 'autoMZ.v1'}
```

As alignment files can be very large, `Align.parse` returns an iterator over the alignments, so you won't have to store all alignments in memory at the same time. You can iterate over these alignments and print out, for example, the number of aligned sequences in each alignment:

```
>>> for a in alignments:
...     print(len(a.sequences))
...
2
4
5
6
...
15
14
7
6
```

You can also call `len` on the alignments to obtain the number of alignments.

```
>>> len(alignments)
48
```

Depending on the file format, the number of alignments may be explicitly stored in the file (for example in the case of bigBed, bigPsl, and bigMaf files), or otherwise the number of alignments is counted by looping over them once (and

returning the iterator to its original position). If the file is large, it may therefore take a considerable amount of time for `len` to return. However, as the number of alignments is cached, subsequent calls to `len` will return quickly.

If the number of alignments is not excessively large and will fit in memory, you can convert the alignments iterator to a list of alignments. To do so, you could call `list` on the `alignments`:

```
>>> alignment_list = list(alignments)
>>> len(alignment_list)
48
>>> alignment_list[27]
<Alignment object (3 rows x 91 columns) at 0x...>
>>> print(alignment_list[27])
mm9.chr10   3019377 CCCCAGCATTCTGGCAGACACAGTG-AAAAGAGACAGATGGTCACTAATAAAATCTGT-A
felCat3.s     46845 CCCAAGTGTTCTGATAGCTAATGTGAAAAAGAAGCATGTGCCCACCAGTAAGCTTTGTGG
canFam2.c  47545247 CCCAAGTGTTCTGATTGCCTCTGTGAAAAAGAAACATGGGCCCGCTAATAagatttgcaa

mm9.chr10   3019435 TAAATTAG-ATCTCAGAGGATGGATGGACCA  3019465
felCat3.s     46785 TGAACTAGAATCTCAGAGGATG---GGACTC    46757
canFam2.c  47545187 tgacctagaatctcagaggatg---ggactc 47545159
```

But this will lose the metadata information:

```
>>> alignment_list.metadata
Traceback (most recent call last):
  ...
AttributeError: 'list' object has no attribute 'metadata'
```

Instead, you can ask for a full slice of the alignments:

```
>>> type(alignments)
<class 'Bio.Align.maf.AlignmentIterator'>
>>> alignments = alignments[:]
>>> type(alignments)
<class 'Bio.Align.Alignments'>
```

This returns a `Bio.Align.Alignments` object, which can be used as a list, while keeping the metadata information:

```
>>> len(alignments)
48
>>> print(alignments[11])
mm9.chr10   3014742 AAGTTCCCTCCATAATTCCTTCCTCCCACCCCCACA 3014778
calJac1.C      6283 AAATGTA-----TGATCTCCCCATCCTGCCCTG---    6311
otoGar1.s    175262 AGATTTC-----TGATGCCCTCACCCCCTCCGTGCA  175231
loxAfr1.s      9317 AGGCTTA-----TG----CCACCCCCCACCCCCACA    9290

>>> alignments.metadata
{'MAF Version': '1', 'Scoring': 'autoMZ.v1'}
```

6.6.2 Writing alignments

To write alignments to a file, use

```
>>> from Bio import Align
>>> target = "myfile.txt"
>>> Align.write(alignments, target, "clustal")
```

where `alignments` is either a single alignment or a list of alignments, `target` is a file name or an open file-like object, and `"clustal"` is the file format to be used. As some file formats allow or require metadata to be stored with the alignments, you may want to use the `Alignments` (plural) class instead of a plain list of alignments (see Section *The Alignments class*), allowing you to store a metadata dictionary as an attribute on the `alignments` object:

```
>>> from Bio import Align
>>> alignments = Align.Alignments(alignments)
>>> metadata = {"Program": "Biopython", "Version": "1.81"}
>>> alignments.metadata = metadata
>>> target = "myfile.txt"
>>> Align.write(alignments, target, "clustal")
```

6.6.3 Printing alignments

For text (non-binary) formats, you can call Python's built-in `format` function on an alignment to get a string showing the alignment in the requested format, or use `Alignment` objects in formatted (f-) strings. If called without an argument, the `format` function returns the string representation of the alignment:

```
>>> str(alignment)
'                  1 CGGTTTTT 9\n                  0 AGGTTT-- 6\n                  0 AG-
↪TTT-- 5\n'
>>> format(alignment)
'                  1 CGGTTTTT 9\n                  0 AGGTTT-- 6\n                  0 AG-
↪TTT-- 5\n'
>>> print(format(alignment))
                  1 CGGTTTTT 9
                  0 AGGTTT-- 6
                  0 AG-TTT-- 5
```

By specifying one of the formats shown in Section *Alignment file formats*, `format` will create a string showing the alignment in the requested format:

```
>>> format(alignment, "clustal")
'sequence_0                          CGGTTTTT\nsequence_1                          ␣
↪AGGTTT--\nsequence_2                          AG-TTT--\n\n\n'
>>> print(format(alignment, "clustal"))
sequence_0                          CGGTTTTT
sequence_1                          AGGTTT--
sequence_2                          AG-TTT--



>>> print(f"*** this is the alignment in Clustal format: ***\n{alignment:clustal}\n***")
*** this is the alignment in Clustal format: ***
sequence_0                          CGGTTTTT
```

(continues on next page)

(continued from previous page)

```
sequence_1                          AGGTTT--
sequence_2                          AG-TTT--



***
>>> format(alignment, "maf")
'a\ns sequence_0 1 8 + 9 CGGTTTTT\ns sequence_1 0 6 + 6 AGGTTT--\ns sequence_2 0 5 + 7␣
↪AG-TTT--\n\n'
>>> print(format(alignment, "maf"))
a
s sequence_0 1 8 + 9 CGGTTTTT
s sequence_1 0 6 + 6 AGGTTT--
s sequence_2 0 5 + 7 AG-TTT--
```

As optional keyword arguments cannot be used with Python's built-in `format` function or with formatted strings, the `Alignment` class has a `format` method with optional arguments to customize the alignment format, as described in the subsections below. For example, we can print the alignment in BED format (see section *Browser Extensible Data (BED)*) with a specific number of columns:

```
>>> print(pairwise_alignment)
target            1 CGGTTTTT 9
                  0 .|-|||-- 8
query             0 AG-TTT-- 5

>>> print(format(pairwise_alignment, "bed"))
target  1   7   query   0   +   1   7   0   2   2,3,    0,3,

>>> print(pairwise_alignment.format("bed"))
target  1   7   query   0   +   1   7   0   2   2,3,    0,3,

>>> print(pairwise_alignment.format("bed", bedN=3))
target  1   7

>>> print(pairwise_alignment.format("bed", bedN=6))
target  1   7   query   0   +
```

6.7 Alignment file formats

The table below shows the alignment formats that can be parsed in Bio.Align. The format argument `fmt` used in `Bio.Align` functions to specify the file format is case-insensitive. Most of these file formats can also be written by `Bio.Align`, as shown in the table.

File format `fmt`	Description	text / binary	Supported by `write`	Subsection
`a2m`	A2M	text	yes	*1.7.11*
`bed`	Browser Extensible Data (BED)	text	yes	*1.7.14*
`bigbed`	bigBed	binary	yes	*1.7.15*
`bigmaf`	bigMaf	binary	yes	*1.7.19*
`bigpsl`	bigPsl	binary	yes	*1.7.17*
`chain`	UCSC chain file	text	yes	*1.7.20*
`clustal`	ClustalW	text	yes	*1.7.2*
`emboss`	EMBOSS	text	no	*1.7.5*
`` exonerate``	Exonerate	text	yes	*1 .7.7*
`fasta`	Aligned FASTA	text	yes	*1.7.1*
`hhr`	HH-suite output files	text	no	*1.7.10*
`maf`	Multiple Alignment Format (MAF)	text	yes	*1.7.18*
`mauve`	Mauve eXtended Multi-FastA (xmfa) format	text	yes	*1.7.12*
`msf`	GCG Multiple Sequence Format (MSF)	text	no	*1.7.6*
`nexus`	NEXUS	text	yes	*1.7.8*
`phylip`	PHYLIP output files	text	yes	*1.7.4*
`psl`	Pattern Space Layout (PSL)	text	yes	*1.7.16*
`sam`	Sequence Al ignment/Map (SAM)	text	yes	*1.7.13*
`` stockholm``	Stockholm	text	yes	*1 .7.3*
`tabular`	Tabular output from BLAST or FASTA	text	no	*1.7.9*

6.7.1 Aligned FASTA

Files in the aligned FASTA format store exactly one (pairwise or multiple) sequence alignment, in which gaps in the alignment are represented by dashes (`-`). Use `fmt="fasta"` to read or write files in the aligned FASTA format. Note that this is different from output generated by William Pearson's FASTA alignment program (parsing such output is described in section *Tabular output from BLAST or FASTA* instead).

The file `probcons.fa` in Biopython's test suite stores one multiple alignment in the aligned FASTA format. The contents of this file is as follows:

```
>plas_horvu
D-VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG-VD-VSKISQEEYLTAPGETFSVTLTV---
↪PGTYGFYCEPHAGAGMVGKVTV
>plas_chlre
--VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG-VN-ADAISRDDYLNAPGETYSVKLTA---
↪AGEYGYYCEPHQGAGMVGKIIV
>plas_anava
--
↪VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKSADLAKSLSHKQLLMSPGQSTSTTFPADAPAGEYTFYCEPHRGAGMVGKITV
>plas_proho
VQIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG-ES-APALSNTKLRIAPGSFYSVTLGT---
↪PGTYSFYCTPHRGAGMVGTITV
>azup_achcy
VHMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG-AE-A-------FKSKINENYKVTFTA---
↪PGVYGVKCTPHYGMGMVGVVEV
```

To read this file, use

```
>>> from Bio import Align
>>> alignment = Align.read("probcons.fa", "fasta")
>>> alignment
<Alignment object (5 rows x 101 columns) at ...>
```

We can print the alignment to see its default representation:

```
>>> print(alignment)
plas_horv         0 D-VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG-VD-VSKISQE
plas_chlr         0 --VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG-VN-ADAISRD
plas_anav         0 --VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKSADLAKSLSHK
plas_proh         0 VQIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG-ES-APALSNT
azup_achc         0 VHMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG-AE-A------

plas_horv        57 EYLTAPGETFSVTLTV---PGTYGFYCEPHAGAGMVGKVTV 95
plas_chlr        56 DYLNAPGETYSVKLTA---AGEYGYYCEPHQGAGMVGKIIV 94
plas_anav        58 QLLMSPGQSTSTTFPADAPAGEYTFYCEPHRGAGMVGKITV 99
plas_proh        56 KLRIAPGSFYSVTLGT---PGTYSFYCTPHRGAGMVGTITV 94
azup_achc        51 -FKSKINENYKVTFTA---PGVYGVKCTPHYGMGMVGVVEV 88
```

or we can print it in the aligned FASTA format:

```
>>> print(format(alignment, "fasta"))
>plas_horvu
D-VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG-VD-VSKISQEEYLTAPGETFSVTLTV---
↪PGTYGFYCEPHAGAGMVGKVTV
>plas_chlre
--VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG-VN-ADAISRDDYLNAPGETYSVKLTA---
↪AGEYGYYCEPHQGAGMVGKIIV
>plas_anava
--
↪VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKSADLAKSLSHKQLLMSPGQSTSTTFPADAPAGEYTFYCEPHRGAGMVGKITV
>plas_proho
VQIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG-ES-APALSNTKLRIAPGSFYSVTLGT---
↪PGTYSFYCTPHRGAGMVGTITV
>azup_achcy
VHMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG-AE-A-------FKSKINENYKVTFTA---
↪PGVYGVKCTPHYGMGMVGVVEV
```

or any other available format, for example Clustal (see section *ClustalW*):

```
>>> print(format(alignment, "clustal"))
plas_horvu                          D-VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG-
plas_chlre                          --VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG-
plas_anava                          --VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKS
plas_proho                          VQIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG-
azup_achcy                          VHMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG-

plas_horvu                          VD-VSKISQEEYLTAPGETFSVTLTV---PGTYGFYCEPHAGAGMVGKVT
plas_chlre                          VN-ADAISRDDYLNAPGETYSVKLTA---AGEYGYYCEPHQGAGMVGKII
plas_anava                          ADLAKSLSHKQLLMSPGQSTSTTFPADAPAGEYTFYCEPHRGAGMVGKIT
plas_proho                          ES-APALSNTKLRIAPGSFYSVTLGT---PGTYSFYCTPHRGAGMVGTIT
azup_achcy                          AE-A-------FKSKINENYKVTFTA---PGVYGVKCTPHYGMGMVGVVE
```

(continues on next page)

(continued from previous page)

```
plas_horvu                          V
plas_chlre                          V
plas_anava                          V
plas_proho                          V
azup_achcy                          V
```

The sequences associated with the alignment are `SeqRecord` objects:

```
>>> alignment.sequences
[SeqRecord(seq=Seq('DVLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSGVDVSKI...VTV'), id=
↪'plas_horvu', name='<unknown name>', description='', dbxrefs=[]), SeqRecord(seq=Seq(
↪'VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSGVNADAIS...IIV'), id='plas_chlre', name=
↪'<unknown name>', description='', dbxrefs=[]), SeqRecord(seq=Seq(
↪'VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKSADLAKS...ITV'), id='plas_anava', name=
↪'<unknown name>', description='', dbxrefs=[]), SeqRecord(seq=Seq(
↪'VQIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDKVPAGESAPALS...ITV'), id='plas_proho', name=
↪'<unknown name>', description='', dbxrefs=[]), SeqRecord(seq=Seq(
↪'VHMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDKGHNVETIKGMIPDGAEAFKS...VEV'), id='azup_achcy', name=
↪'<unknown name>', description='', dbxrefs=[])]
```

Note that these sequences do not contain gaps ("-" characters), as the alignment information is stored in the `coordinates` attribute instead:

```
>>> print(alignment.coordinates)
[[ 0  1  1 33 34 42 44 48 48 50 50 51 58 73 73 95]
 [ 0  0  0 32 33 41 43 47 47 49 49 50 57 72 72 94]
 [ 0  0  0 32 33 41 43 47 48 50 51 52 59 74 77 99]
 [ 0  1  2 34 35 43 43 47 47 49 49 50 57 72 72 94]
 [ 0  1  2 34 34 42 44 48 48 50 50 51 51 66 66 88]]
```

Use `Align.write` to write this alignment to a file (here, we'll use a `StringIO` object instead of a file):

```
>>> from io import StringIO
>>> stream = StringIO()
>>> Align.write(alignment, stream, "FASTA")
1
>>> print(stream.getvalue())
>plas_horvu
D-VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG-VD-VSKISQEEYLTAPGETFSVTLTV---
↪PGTYGFYCEPHAGAGMVGKVTV
>plas_chlre
--VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG-VN-ADAISRDDYLNAPGETYSVKLTA---
↪AGEYGYYCEPHQGAGMVGKIIV
>plas_anava
--
↪VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKSADLAKSLSHKQLLMSPGQSTSTTFPADAPAGEYTFYCEPHRGAGMVGKITV
>plas_proho
VQIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG-ES-APALSNTKLRIAPGSFYSVTLGT---
↪PGTYSFYCTPHRGAGMVGTITV
>azup_achcy
```

(continues on next page)

(continued from previous page)

```
VHMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG-AE-A-------FKSKINENYKVTFTA---
↪PGVYGVKCTPHYGMGMVGVVEV
```

Note that `Align.write` returns the number of alignments written (1, in this case).

6.7.2 ClustalW

Clustal is a set of multiple sequence alignment programs that are available both as standalone programs as as web servers. The file `opuntia.aln` (available online or in the `Doc/examples` subdirectory of the Biopython source code) is an output file generated by Clustal. Its first few lines are

```
CLUSTAL 2.1 multiple sequence alignment


gi|6273285|gb|AF191659.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273284|gb|AF191658.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273287|gb|AF191661.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273286|gb|AF191660.1|AF191      TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273290|gb|AF191664.1|AF191      TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273289|gb|AF191663.1|AF191      TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273291|gb|AF191665.1|AF191      TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
                                    ******* **** **************************************

...
```

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("opuntia.aln", "clustal")
```

The `metadata` attribute on `alignments` stores the information shown in the file header:

```
>>> alignments.metadata
{'Program': 'CLUSTAL', 'Version': '2.1'}
```

You can call `next` on the `alignments` to pull out the first (and only) alignment:

```
>>> alignment = next(alignments)
>>> print(alignment)
gi|627328         0 TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT
gi|627328         0 TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT
gi|627328         0 TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT
gi|627328         0 TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT
gi|627329         0 TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT
gi|627328         0 TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT
gi|627329         0 TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT

gi|627328        60 CTAAATGATATACGATTCCACTATGTAAGGTCTTTGAATCATATCATAAAAGACAATGTA
gi|627328        60 CTAAATGATATACGATTCCACTATGTAAGGTCTTTGAATCATATCATAAAAGACAATGTA
gi|627328        60 CTAAATGATATACGATTCCACTATGTAAGGTCTTTGAATCATATCATAAAAGACAATGTA
gi|627328        60 CTAAATGATATACGATTCCACTA...
```

If you are not interested in the metadata, then it is more convenient to use the `Align.read` function, as anyway each Clustal file contains only one alignment:

```
>>> from Bio import Align
>>> alignment = Align.read("opuntia.aln", "clustal")
```

The consensus line below each alignment block in the Clustal output file contains an asterisk if the sequence is conserved at each position. This information is stored in the `column_annotations` attribute of the `alignment`:

```
>>> alignment.column_annotations
{'clustal_consensus': '******* **** **********************************...
```

Printing the `alignment` in `clustal` format will show the sequence alignment, but does not include the metadata:

```
>>> print(format(alignment, "clustal"))
gi|6273285|gb|AF191659.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273284|gb|AF191658.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273287|gb|AF191661.1|AF191      TATACATT...
```

Writing the `alignments` in `clustal` format will include both the metadata and the sequence alignment:

```
>>> from io import StringIO
>>> stream = StringIO()
>>> Align.write(alignments, stream, "clustal")
1
>>> print(stream.getvalue())
CLUSTAL 2.1 multiple sequence alignment


gi|6273285|gb|AF191659.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273284|gb|AF191658.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273287|gb|AF191661.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273286|gb|AF191660.1|AF191      TATACATAAAAGAAG...
```

Use an `Alignments` (plural) object (see Section *The Alignments class*) if you are creating alignments by hand, and would like to include metadata information in the output.

6.7.3 Stockholm

This is an example of a protein sequence alignment in the Stockholm file format used by PFAM:

```
# STOCKHOLM 1.0
#=GF ID   7kD_DNA_binding
#=GF AC   PF02294.20
#=GF DE   7kD DNA-binding domain
#=GF AU   Mian N;0000-0003-4284-4749
#=GF AU   Bateman A;0000-0002-6982-4660
#=GF SE   Pfam-B_8148 (release 5.2)
#=GF GA   25.00 25.00;
#=GF TC   26.60 46.20;
#=GF NC   23.20 19.20;
#=GF BM   hmmbuild HMM.ann SEED.ann
#=GF SM   hmmsearch -Z 57096847 -E 1000 --cpu 4 HMM pfamseq
```

(continues on next page)

(continued from previous page)

```
#=GF TP   Domain
#=GF CL   CL0049
#=GF RN   [1]
#=GF RM   3130377
#=GF RT   Microsequence analysis of DNA-binding proteins 7a, 7b, and 7e
#=GF RT   from the archaebacterium Sulfolobus acidocaldarius.
#=GF RA   Choli T, Wittmann-Liebold B, Reinhardt R;
#=GF RL   J Biol Chem 1988;263:7087-7093.
#=GF DR   INTERPRO; IPR003212;
#=GF DR   SCOP; 1sso; fa;
#=GF DR   SO; 0000417; polypeptide_domain;
#=GF CC   This family contains members of the hyper-thermophilic
#=GF CC   archaebacterium  7kD DNA-binding/endoribonuclease P2 family.
#=GF CC   There are five 7kD DNA-binding proteins, 7a-7e, found as
#=GF CC   monomers in the cell. Protein 7e shows the  tightest DNA-binding
#=GF CC   ability.
#=GF SQ   3
#=GS DN7_METS5/4-61   AC A4YEA2.1
#=GS DN7A_SACS2/3-61  AC P61991.2
#=GS DN7A_SACS2/3-61  DR PDB; 1SSO A; 2-60;
#=GS DN7A_SACS2/3-61  DR PDB; 1JIC A; 2-60;
#=GS DN7A_SACS2/3-61  DR PDB; 2CVR A; 2-60;
#=GS DN7A_SACS2/3-61  DR PDB; 1B4O A; 2-60;
#=GS DN7E_SULAC/3-60  AC P13125.2
DN7_METS5/4-61              KIKFKYKGQDLEVDISKVKKVWKVGKMVSFTYDD.NGKTGRGAVSEKDAPKELLNMIGK
DN7A_SACS2/3-61             TVKFKYKGEEKQVDISKIKKVWRVGKMISFTYDEGGGKTGRGAVSEKDAPKELLQMLEK
#=GR DN7A_SACS2/3-61  SS    EEEEESSSSEEEEETTTEEEEEESSSSEEEEEE-SSSSEEEEEEETTTS-CHHHHHHTT
DN7E_SULAC/3-60             KVRFKYKGEEKEVDTSKIKKVWRVGKMVSFTYDD.NGKTGRGAVSEKDAPKELMDMLAR
#=GC SS_cons                EEEEESSSSEEEEETTTEEEEEESSSSEEEEEE-SSSSEEEEEEETTTS-CHHHHHHTT
#=GC seq_cons               KVKFKYKGEEKEVDISKIKKVWRVGKMVSFTYDD.NGKTGRGAVSEKDAPKELLsMLuK
//
```

This is the seed alignment for the 7kD_DNA_binding (PF02294.20) PFAM entry, downloaded from the InterPro website (https://www.ebi.ac.uk/interpro/). This version of the PFAM entry is also available in the Biopython source distribution as the file `pfam2.seed.txt` in the subdirectory `Tests/Stockholm/`. We can load this file as follows:

```
>>> from Bio import Align
>>> alignment = Align.read("pfam2.seed.txt", "stockholm")
>>> alignment
<Alignment object (3 rows x 59 columns) at ...>
```

We can print out a summary of the alignment:

```
>>> print(alignment)
DN7_METS5         0 KIKFKYKGQDLEVDISKVKKVWKVGKMVSFTYDD-NGKTGRGAVSEKDAPKELLNMIGK
DN7A_SACS         0 TVKFKYKGEEKQVDISKIKKVWRVGKMISFTYDEGGGKTGRGAVSEKDAPKELLQMLEK
DN7E_SULA         0 KVRFKYKGEEKEVDTSKIKKVWRVGKMVSFTYDD-NGKTGRGAVSEKDAPKELMDMLAR

DN7_METS5        58
DN7A_SACS        59
DN7E_SULA        58
```

You could also call Python's built-in `format` function on the alignment object to show it in a particular file format (see

section *Printing alignments* for details), for example in the Stockholm format to regenerate the file:

```
>>> print(format(alignment, "stockholm"))
# STOCKHOLM 1.0
#=GF ID   7kD_DNA_binding
#=GF AC   PF02294.20
#=GF DE   7kD DNA-binding domain
#=GF AU   Mian N;0000-0003-4284-4749
#=GF AU   Bateman A;0000-0002-6982-4660
#=GF SE   Pfam-B_8148 (release 5.2)
#=GF GA   25.00 25.00;
#=GF TC   26.60 46.20;
#=GF NC   23.20 19.20;
#=GF BM   hmmbuild HMM.ann SEED.ann
#=GF SM   hmmsearch -Z 57096847 -E 1000 --cpu 4 HMM pfamseq
#=GF TP   Domain
#=GF CL   CL0049
#=GF RN   [1]
#=GF RM   3130377
#=GF RT   Microsequence analysis of DNA-binding proteins 7a, 7b, and 7e from
#=GF RT   the archaebacterium Sulfolobus acidocaldarius.
#=GF RA   Choli T, Wittmann-Liebold B, Reinhardt R;
#=GF RL   J Biol Chem 1988;263:7087-7093.
#=GF DR   INTERPRO; IPR003212;
#=GF DR   SCOP; 1sso; fa;
#=GF DR   SO; 0000417; polypeptide_domain;
#=GF CC   This family contains members of the hyper-thermophilic
#=GF CC   archaebacterium  7kD DNA-binding/endoribonuclease P2 family. There
#=GF CC   are five 7kD DNA-binding proteins, 7a-7e, found as monomers in the
#=GF CC   cell. Protein 7e shows the  tightest DNA-binding ability.
#=GF SQ   3
#=GS DN7_METS5/4-61   AC A4YEA2.1
#=GS DN7A_SACS2/3-61  AC P61991.2
#=GS DN7A_SACS2/3-61  DR PDB; 1SSO A; 2-60;
#=GS DN7A_SACS2/3-61  DR PDB; 1JIC A; 2-60;
#=GS DN7A_SACS2/3-61  DR PDB; 2CVR A; 2-60;
#=GS DN7A_SACS2/3-61  DR PDB; 1B4O A; 2-60;
#=GS DN7E_SULAC/3-60  AC P13125.2
DN7_METS5/4-61                  KIKFKYKGQDLEVDISKVKKVWKVGKMVSFTYDD.
↪NGKTGRGAVSEKDAPKELLNMIGK
DN7A_SACS2/3-61                ␣
↪TVKFKYKGEEKQVDISKIKKVWRVGKMISFTYDEGGGKTGRGAVSEKDAPKELLQMLEK
#=GR DN7A_SACS2/3-61  SS        EEEEESSSSEEEEETTTEEEEEESSSSEEEEEE-SSSSEEEEEEETTTS-
↪CHHHHHHTT
DN7E_SULAC/3-60                 KVRFKYKGEEKEVDTSKIKKVWRVGKMVSFTYDD.
↪NGKTGRGAVSEKDAPKELMDMLAR
#=GC SS_cons                    EEEEESSSSEEEEETTTEEEEEESSSSEEEEEE-SSSSEEEEEEETTTS-
↪CHHHHHHTT
#=GC seq_cons                   KVKFKYKGEEKEVDISKIKKVWRVGKMVSFTYDD.
↪NGKTGRGAVSEKDAPKELLsMLuK
//
```

or alternatively as aligned FASTA (see section *Aligned FASTA*):

```
>>> print(format(alignment, "fasta"))
>DN7_METS5/4-61
KIKFKYKGQDLEVDISKVKKVWKVGKMVSFTYDD-NGKTGRGAVSEKDAPKELLNMIGK
>DN7A_SACS2/3-61
TVKFKYKGEEKQVDISKIKKVWRVGKMISFTYDEGGGKTGRGAVSEKDAPKELLQMLEK
>DN7E_SULAC/3-60
KVRFKYKGEEKEVDTSKIKKVWRVGKMVSFTYDD-NGKTGRGAVSEKDAPKELMDMLAR
```

or in the PHYLIP format (see section *PHYLIP output files*):

```
>>> print(format(alignment, "phylip"))
3 59
DN7_METS5/KIKFKYKGQDLEVDISKVKKVWKVGKMVSFTYDD-NGKTGRGAVSEKDAPKELLNMIGK
DN7A_SACS2TVKFKYKGEEKQVDISKIKKVWRVGKMISFTYDEGGGKTGRGAVSEKDAPKELLQMLEK
DN7E_SULACKVRFKYKGEEKEVDTSKIKKVWRVGKMVSFTYDD-NGKTGRGAVSEKDAPKELMDMLAR
```

General information of the alignment is stored under the `annotations` attribute of the `Alignment` object, for example

```
>>> alignment.annotations["identifier"]
'7kD_DNA_binding'
>>> alignment.annotations["clan"]
'CL0049'
>>> alignment.annotations["database references"]
[{'reference': 'INTERPRO; IPR003212;'}, {'reference': 'SCOP; 1sso; fa;'}, {'reference':
↪'SO; 0000417; polypeptide_domain;'}]
```

The individual sequences in this alignment are stored under `alignment.sequences` as `SeqRecord`s, including any annotations associated with each sequence record:

```
>>> for record in alignment.sequences:
...     print("%s %s %s" % (record.id, record.annotations["accession"], record.dbxrefs))
...
DN7_METS5/4-61 A4YEA2.1 []
DN7A_SACS2/3-61 P61991.2 ['PDB; 1SSO A; 2-60;', 'PDB; 1JIC A; 2-60;', 'PDB; 2CVR A; 2-60;
↪', 'PDB; 1B4O A; 2-60;']
DN7E_SULAC/3-60 P13125.2 []
```

The secondary structure of the second sequence (`DN7A_SACS2/3-61`) is stored in the `letter_annotations` attribute of the `SeqRecord`:

```
>>> alignment.sequences[0].letter_annotations
{}
>>> alignment.sequences[1].letter_annotations
{'secondary structure': 'EEEEESSSSEEEEETTTEEEEESSSSEEEEEE-SSSSEEEEEEETTTS-CHHHHHHTT'}
>>> alignment.sequences[2].letter_annotations
{}
```

The consensus sequence and secondary structure are associated with the sequence alignment as a whole, and are therefore stored in the `column_annotations` attribute of the `Alignment` object:

```
>>> alignment.column_annotations
{'consensus secondary structure': 'EEEEESSSSEEEEETTTEEEEESSSSEEEEEE-SSSSEEEEEEETTTS-
↪CHHHHHHTT',
 'consensus sequence': 'KVKFKYKGEEKEVDISKIKKVWRVGKMVSFTYDD.NGKTGRGAVSEKDAPKELLsMLuK'}
```

6.7.4 PHYLIP output files

The PHYLIP format for sequence alignments is derived from the PHYLogeny Interference Package from Joe Felsenstein. Files in the PHYLIP format start with two numbers for the number of rows and columns in the printed alignment. The sequence alignment itself can be in sequential format or in interleaved format. An example of the former is the `sequential.phy` file (provided in `Tests/Phylip/` in the Biopython source distribution):

```
 3 384
CYS1_DICDI   -----MKVIL LFVLAVFTVF VSS------- --------RG IPPEEQ---- --------SQ
             FLEFQDKFNK KY-SHEEYLE RFEIFKSNLG KIEELNLIAI NHKADTKFGV NKFADLSSDE
             FKNYYLNNKE AIFTDDLPVA DYLDDEFINS IPTAFDWRTR G-AVTPVKNQ GQCGSCWSFS
             TTGNVEGQHF ISQNKLVSLS EQNLVDCDHE CMEYEGEEAC DEGCNGGLQP NAYNYIIKNG
             GIQTESSYPY TAETGTQCNF NSANIGAKIS NFTMIP-KNE TVMAGYIVST GPLAIAADAV
             E-WQFYIGGV F-DIPCN--P NSLDHGILIV GYSAKNTIFR KNMPYWIVKN SWGADWGEQG
             YIYLRRGKNT CGVSNFVSTS II--
ALEU_HORVU   MAHARVLLLA LAVLATAAVA VASSSSFADS NPIRPVTDRA ASTLESAVLG ALGRTRHALR
             FARFAVRYGK SYESAAEVRR RFRIFSESLE EVRSTN---- RKGLPYRLGI NRFSDMSWEE
             FQATRL-GAA QTCSATLAGN HLMRDA--AA LPETKDWRED G-IVSPVKNQ AHCGSCWTFS
             TTGALEAAYT QATGKNISLS EQQLVDCAGG FNNF------ --GCNGGLPS QAFEYIKYNG
             GIDTEESYPY KGVNGV-CHY KAENAAVQVL DSVNITLNAE DELKNAVGLV RPVSVAFQVI
             DGFRQYKSGV YTSDHCGTTP DDVNHAVLAV GYGVENGV-- ---PYWLIKN SWGADWGDNG
             YFKMEMGKNM CAIATCASYP VVAA
CATH_HUMAN   ------MWAT LPLLCAGAWL LGV------- -PVCGAAELS VNSLEK---- --------FH
             FKSWMSKHRK TY-STEEYHH RLQTFASNWR KINAHN---- NGNHTFKMAL NQFSDMSFAE
             IKHKYLWSEP QNCSAT--KS NYLRGT--GP YPPSVDWRKK GNFVSPVKNQ GACGSCWTFS
             TTGALESAIA IATGKMLSLA EQQLVDCAQD FNNY------ --GCQGGLPS QAFEYILYNK
             GIMGEDTYPY QGKDGY-CKF QPGKAIGFVK DVANITIYDE EAMVEAVALY NPVSFAFEVT
             QDFMMYRTGI YSSTSCHKTP DKVNHAVLAV GYGEKNGI-- ---PYWIVKN SWGPQWGMNG
             YFLIERGKNM CGLAACASYP IPLV
```

In the sequential format, the complete alignment for one sequence is shown before proceeding to the next sequence. In the interleaved format, the alignments for different sequences are next to each other, for example in the file `interlaced.phy` (provided in `Tests/Phylip/` in the Biopython source distribution):

```
 3 384
CYS1_DICDI   -----MKVIL LFVLAVFTVF VSS------- --------RG IPPEEQ---- --------SQ
ALEU_HORVU   MAHARVLLLA LAVLATAAVA VASSSSFADS NPIRPVTDRA ASTLESAVLG ALGRTRHALR
CATH_HUMAN   ------MWAT LPLLCAGAWL LGV------- -PVCGAAELS VNSLEK---- --------FH

             FLEFQDKFNK KY-SHEEYLE RFEIFKSNLG KIEELNLIAI NHKADTKFGV NKFADLSSDE
             FARFAVRYGK SYESAAEVRR RFRIFSESLE EVRSTN---- RKGLPYRLGI NRFSDMSWEE
             FKSWMSKHRK TY-STEEYHH RLQTFASNWR KINAHN---- NGNHTFKMAL NQFSDMSFAE

             FKNYYLNNKE AIFTDDLPVA DYLDDEFINS IPTAFDWRTR G-AVTPVKNQ GQCGSCWSFS
             FQATRL-GAA QTCSATLAGN HLMRDA--AA LPETKDWRED G-IVSPVKNQ AHCGSCWTFS
             IKHKYLWSEP QNCSAT--KS NYLRGT--GP YPPSVDWRKK GNFVSPVKNQ GACGSCWTFS

             TTGNVEGQHF ISQNKLVSLS EQNLVDCDHE CMEYEGEEAC DEGCNGGLQP NAYNYIIKNG
             TTGALEAAYT QATGKNISLS EQQLVDCAGG FNNF------ --GCNGGLPS QAFEYIKYNG
             TTGALESAIA IATGKMLSLA EQQLVDCAQD FNNY------ --GCQGGLPS QAFEYILYNK

             GIQTESSYPY TAETGTQCNF NSANIGAKIS NFTMIP-KNE TVMAGYIVST GPLAIAADAV
             GIDTEESYPY KGVNGV-CHY KAENAAVQVL DSVNITLNAE DELKNAVGLV RPVSVAFQVI
             GIMGEDTYPY QGKDGY-CKF QPGKAIGFVK DVANITIYDE EAMVEAVALY NPVSFAFEVT
```

(continues on next page)

(continued from previous page)

```
            E-WQFYIGGV F-DIPCN--P NSLDHGILIV GYSAKNTIFR KNMPYWIVKN SWGADWGEQG
            DGFRQYKSGV YTSDHCGTTP DDVNHAVLAV GYGVENGV-- ---PYWLIKN SWGADWGDNG
            QDFMMYRTGI YSSTSCHKTP DKVNHAVLAV GYGEKNGI-- ---PYWIVKN SWGPQWGMNG

            YIYLRRGKNT CGVSNFVSTS II--
            YFKMEMGKNM CAIATCASYP VVAA
            YFLIERGKNM CGLAACASYP IPLV
```

The parser in `Bio.Align` detects from the file contents if it is in the sequential or in the interleaved format, and then parses it appropriately.

```
>>> from Bio import Align
>>> alignment = Align.read("sequential.phy", "phylip")
>>> alignment
<Alignment object (3 rows x 384 columns) at ...>
>>> alignment2 = Align.read("interlaced.phy", "phylip")
>>> alignment2
<Alignment object (3 rows x 384 columns) at ...>
>>> alignment == alignment2
True
```

Here, two alignments are considered to be equal if they have the same sequence contents and the same alignment coordinates.

```
>>> alignment.shape
(3, 384)
>>> print(alignment)
CYS1_DICD         0 -----MKVILLFVLAVFTVFVSS---------------RGIPPEEQ------------SQ
ALEU_HORV         0 MAHARVLLLALAVLATAAVAVASSSSFADSNPIRPVTDRAASTLESAVLGALGRTRHALR
CATH_HUMA         0 ------MWATLPLLCAGAWLLGV--------PVCGAAELSVNSLEK------------FH

CYS1_DICD        28 FLEFQDKFNKKY-SHEEYLERFEIFKSNLGKIEELNLIAINHKADTKFGVNKFADLSSDE
ALEU_HORV        60 FARFAVRYGKSYESAAEVRRRFRIFSESLEEVRSTN----RKGLPYRLGINRFSDMSWEE
CATH_HUMA        34 FKSWMSKHRKTY-STEEYHHRLQTFASNWRKINAHN----NGNHTFKMALNQFSDMSFAE

CYS1_DICD        87 FKNYYLNNKEAIFTDDLPVADYLDDEFINSIPTAFDWRTRG-AVTPVKNQGQCGSCWSFS
ALEU_HORV       116 FQATRL-GAAQTCSATLAGNHLMRDA--AALPETKDWREDG-IVSPVKNQAHCGSCWTFS
CATH_HUMA        89 IKHKYLWSEPQNCSAT--KSNYLRGT--GPYPPSVDWRKKGNFVSPVKNQGACGSCWTFS

CYS1_DICD       146 TTGNVEGQHFISQNKLVSLSEQNLVDCDHECMEYEGEEACDEGCNGGLQPNAYNYIIKNG
ALEU_HORV       172 TTGALEAAYTQATGKNISLSEQQLVDCAGGFNNF--------GCNGGLPSQAFEYIKYNG
CATH_HUMA       145 TTGALESAIAIATGKMLSLAEQQLVDCAQDFNNY--------GCQGGLPSQAFEYILYNK

CYS1_DICD       206 GIQTESSYPYTAETGTQCNFNSANIGAKISNFTMIP-KNETVMAGYIVSTGPLAIAADAV
ALEU_HORV       224 GIDTEESYPYKGVNGV-CHYKAENAAVQVLDSVNITLNAEDELKNAVGLVRPVSVAFQVI
CATH_HUMA       197 GIMGEDTYPYQGKDGY-CKFQPGKAIGFVKDVANITIYDEEAMVEAVALYNPVSFAFEVT

CYS1_DICD       265 E-WQFYIGGVF-DIPCN--PNSLDHGILIVGYSAKNTIFRKNMPYWIVKNSWGADWGEQG
ALEU_HORV       283 DGFRQYKSGVYTSDHCGTTPDDVNHAVLAVGYGVENGV-----PYWLIKNSWGADWGDNG
CATH_HUMA       256 QDFMMYRTGIYSSTSCHKTPDKVNHAVLAVGYGEKNGI-----PYWIVKNSWGPQWGMNG
```

(continues on next page)

(continued from previous page)

```
CYS1_DICD       321 YIYLRRGKNTCGVSNFVSTSII-- 343
ALEU_HORV       338 YFKMEMGKNMCAIATCASYPVVAA 362
CATH_HUMA       311 YFLIERGKNMCGLAACASYPIPLV 335
```

When outputting the alignment in PHYLIP format, `Bio.Align` writes each of the aligned sequences on one line:

```
>>> print(format(alignment, "phylip"))
3 384
CYS1_DICDI-----MKVILLFVLAVFTVFVSS---------------RGIPPEEQ------------SQFLEFQDKFNKKY-
↪SHEEYLERFEIFKSNLGKIEELNLIAINHKADTKFGVNKFADLSSDEFKNYYLNNKEAIFTDDLPVADYLDDEFINSIPTAFDWRTRG-
↪AVTPVKNQGQCGSCWSFSTTGNVEGQHFISQNKLVSLSEQNLVDCDHECMEYEGEEACDEGCNGGLQPNAYNYIIKNGGIQTESSYPYTAETGTQCNFNSA
↪KNETVMAGYIVSTGPLAIAADAVE-WQFYIGGVF-DIPCN--
↪PNSLDHGILIVGYSAKNTIFRKNMPYWIVKNSWGADWGEQGYIYLRRGKNTCGVSNFVSTSII--
ALEU_
↪HORVUMAHARVLLLALAVLATAAVAVASSSSFADSNPIRPVTDRAASTLESAVLGALGRTRHALRFARFAVRYGKSYESAAEVRRRFRIFSESLEEVRSTN
↪---RKGLPYRLGINRFSDMSWEEFQATRL-GAAQTCSATLAGNHLMRDA--AALPETKDWREDG-
↪IVSPVKNQAHCGSCWTFSTTGALEAAYTQATGKNISLSEQQLVDCAGGFNNF--------
↪GCNGGLPSQAFEYIKYNGGIDTEESYPYKGVNGV-
↪CHYKAENAAVQVLDSVNITLNAEDELKNAVGLVRPVSVAFQVIDGFRQYKSGVYTSDHCGTTPDDVNHAVLAVGYGVENGV-----
↪PYWLIKNSWGADWGDNGYFKMEMGKNMCAIATCASYPVVAA
CATH_HUMAN------MWATLPLLCAGAWLLGV--------PVCGAAELSVNSLEK------------FHFKSWMSKHRKTY-
↪STEEYHHRLQTFASNWRKINAHN----NGNHTFKMALNQFSDMSFAEIKHKYLWSEPQNCSAT--KSNYLRGT--
↪GPYPPSVDWRKKGNFVSPVKNQGACGSCWTFSTTGALESAIAIATGKMLSLAEQQLVDCAQDFNNY--------
↪GCQGGLPSQAFEYILYNKGIMGEDTYPYQGKDGY-
↪CKFQPGKAIGFVKDVANITIYDEEAMVEAVALYNPVSFAFEVTQDFMMYRTGIYSSTSCHKTPDKVNHAVLAVGYGEKNGI-----
↪PYWIVKNSWGPQWGMNGYFLIERGKNMCGLAACASYPIPLV
```

We can write the alignment in PHYLIP format, parse the result, and confirm it is the same as the original alignment object:

```
>>> from io import StringIO
>>> stream = StringIO()
>>> Align.write(alignment, stream, "phylip")
1
>>> stream.seek(0)
0
>>> alignment3 = Align.read(stream, "phylip")
>>> alignment == alignment3
True
>>> [record.id for record in alignment.sequences]
['CYS1_DICDI', 'ALEU_HORVU', 'CATH_HUMAN']
>>> [record.id for record in alignment3.sequences]
['CYS1_DICDI', 'ALEU_HORVU', 'CATH_HUMAN']
```

6.7.5 EMBOSS

EMBOSS (European Molecular Biology Open Software Suite) is a set of open-source software tools for molecular biology and bioinformatics [Rice2000]. It includes software such as `needle` and `water` for pairwise sequence alignment. This is an example of output generated by the `water` program for Smith-Waterman local pairwise sequence alignment (available as `water.txt` in the `Tests/Emboss` directory of the Biopython distribution):

```
########################################
# Program:  water
# Rundate:  Wed Jan 16 17:23:19 2002
# Report_file: stdout
########################################
#=======================================
#
# Aligned_sequences: 2
# 1: IXI_234
# 2: IXI_235
# Matrix: EBLOSUM62
# Gap_penalty: 10.0
# Extend_penalty: 0.5
#
# Length: 131
# Identity:     112/131 (85.5%)
# Similarity:   112/131 (85.5%)
# Gaps:          19/131 (14.5%)
# Score: 591.5
#
#
#=======================================

IXI_234            1 TSPASIRPPAGPSSRPAMVSSRRTRPSPPGPRRPTGRPCCSAAPRRPQAT     50
                     |||||||||||||||         ||||||||||||||||||||||||||
IXI_235            1 TSPASIRPPAGPSSR---------RPSPPGPRRPTGRPCCSAAPRRPQAT     41

IXI_234           51 GGWKTCSGTCTTSTSTRHRGRSGWSARTTTAACLRASRKSMRAACSRSAG    100
                     |||||||||||||||||||||||||          ||||||||||||||||
IXI_235           42 GGWKTCSGTCTTSTSTRHRGRSGW----------RASRKSMRAACSRSAG     81

IXI_234          101 SRPNRFAPTLMSSCITSTTGPPAWAGDRSHE    131
                     ||||||||||||||||||||||||||||||||
IXI_235           82 SRPNRFAPTLMSSCITSTTGPPAWAGDRSHE    112


#---------------------------------------
#---------------------------------------
```

As this output file contains only one alignment, we can use `Align.read` to extract it directly. Here, instead we will use `Align.parse` so we can see the metadata of this `water` run:

```
>>> from Bio import Align
>>> alignments = Align.parse("water.txt", "emboss")
```

The `metadata` attribute of `alignments` stores the information shown in the header of the file, including the program used to generate the output, the date and time the program was run, the output file name, and the specific alignment

file format that was used (assumed to be `srspair` by default):

```
>>> alignments.metadata
{'Align_format': 'srspair', 'Program': 'water', 'Rundate': 'Wed Jan 16 17:23:19 2002',
↪'Report_file': 'stdout'}
```

To pull out the alignment, we use

```
>>> alignment = next(alignments)
>>> alignment
<Alignment object (2 rows x 131 columns) at ...>
>>> alignment.shape
(2, 131)
>>> print(alignment)
IXI_234           0 TSPASIRPPAGPSSRPAMVSSRRTRPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTC
                  0 |||||||||||||||---------|||||||||||||||||||||||||||||||||||
IXI_235           0 TSPASIRPPAGPSSR---------RPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTC

IXI_234          60 TTSTSTRHRGRSGWSARTTTAACLRASRKSMRAACSRSAGSRPNRFAPTLMSSCITSTTG
                  60 |||||||||||||||----------|||||||||||||||||||||||||||||||||||
IXI_235          51 TTSTSTRHRGRSGW----------RASRKSMRAACSRSAGSRPNRFAPTLMSSCITSTTG

IXI_234         120 PPAWAGDRSHE 131
                120 |||||||||||| 131
IXI_235         101 PPAWAGDRSHE 112

>>> print(alignment.coordinates)
[[  0  15  24  74  84 131]
 [  0  15  15  65  65 112]]
```

We can use indices to extract specific parts of the alignment:

```
>>> alignment[0]

↪'TSPASIRPPAGPSSRPAMVSSRRTRPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTCTTSTSTRHRGRSGWSARTTTAACLRASRKSMRAACSRSAG
↪'
>>> alignment[1]
'TSPASIRPPAGPSSR---------RPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTCTTSTSTRHRGRSGW----------
↪RASRKSMRAACSRSAGSRPNRFAPTLMSSCITSTTGPPAWAGDRSHE'
>>> alignment[1, 10:30]
'GPSSR---------RPSPPG'
```

The `annotations` attribute of the `alignment` stores the information associated with this alignment specifically:

```
>>> alignment.annotations
{'Matrix': 'EBLOSUM62', 'Gap_penalty': 10.0, 'Extend_penalty': 0.5, 'Identity': 112,
↪'Similarity': 112, 'Gaps': 19, 'Score': 591.5}
```

The number of gaps, identities, and mismatches can also be obtained by calling the `counts` method on the `alignment` object:

```
>>> alignment.counts()
AlignmentCounts(gaps=19, identities=112, mismatches=0)
```

where `AlignmentCounts` is a `namedtuple` in the `collections` module in Python's standard library.

The consensus line shown between the two sequences is stored in the `column_annotations` attribute:

```
>>> alignment.column_annotations
{'emboss_consensus': '||||||||||||||||        ␣
↪|||||||||||||||||||||||||||||||||||||||||||||||||        ␣
↪|||||||||||||||||||||||||||||||||||||||||||||||||'}
```

Use the `format` function (or the `format` method) to print the alignment in other formats, for example in the PHYLIP format (see section *PHYLIP output files*):

```
>>> print(format(alignment, "phylip"))
2 131
IXI_234  ␣
↪TSPASIRPPAGPSSRPAMVSSRRTRPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTCTTSTSTRHRGRSGWSARTTTAACLRASRKSMRAACSRSAGSI
IXI_235   TSPASIRPPAGPSSR---------RPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTCTTSTSTRHRGRSGW-----
↪-----RASRKSMRAACSRSAGSRPNRFAPTLMSSCITSTTGPPAWAGDRSHE
```

We can use `alignment.sequences` to get the individual sequences. However, as this is a pairwise alignment, we can also use `alignment.target` and `alignment.query` to get the target and query sequences:

```
>>> alignment.target
SeqRecord(seq=Seq('TSPASIRPPAGPSSRPAMVSSRRTRPSPPGPRRPTGRPCCSAAPRRPQATGGWK...SHE'), id=
↪'IXI_234', name='<unknown name>', description='<unknown description>', dbxrefs=[])
>>> alignment.query
SeqRecord(seq=Seq('TSPASIRPPAGPSSRRPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTCTTS...SHE'), id=
↪'IXI_235', name='<unknown name>', description='<unknown description>', dbxrefs=[])
```

Currently, Biopython does not support writing sequence alignments in the output formats defined by EMBOSS.

6.7.6 GCG Multiple Sequence Format (MSF)

The Multiple Sequence Format (MSF) was created to store multiple sequence alignments generated by the GCG (Genetics Computer Group) set of programs. The file `W_prot.msf` in the `Tests/msf` directory of the Biopython distribution is an example of a sequence alignment file in the MSF format This file shows an alignment of 11 protein sequences:

```
!!AA_MULTIPLE_ALIGNMENT

   MSF: 99  Type: P  Oct 18, 2017  11:35  Check: 0 ..

 Name: W*01:01:01:01    Len:    99  Check: 7236  Weight:  1.00
 Name: W*01:01:01:02    Len:    99  Check: 7236  Weight:  1.00
 Name: W*01:01:01:03    Len:    99  Check: 7236  Weight:  1.00
 Name: W*01:01:01:04    Len:    99  Check: 7236  Weight:  1.00
 Name: W*01:01:01:05    Len:    99  Check: 7236  Weight:  1.00
 Name: W*01:01:01:06    Len:    99  Check: 7236  Weight:  1.00
 Name: W*02:01          Len:    93  Check: 9483  Weight:  1.00
 Name: W*03:01:01:01    Len:    93  Check: 9974  Weight:  1.00
 Name: W*03:01:01:02    Len:    93  Check: 9974  Weight:  1.00
 Name: W*04:01          Len:    93  Check: 9169  Weight:  1.00
 Name: W*05:01          Len:    99  Check: 7331  Weight:  1.00
//

  W*01:01:01:01  GLTPFNGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
```

(continues on next page)

(continued from previous page)

```
  W*01:01:01:02  GLTPFNGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
  W*01:01:01:03  GLTPFNGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
  W*01:01:01:04  GLTPFNGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
  W*01:01:01:05  GLTPFNGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
  W*01:01:01:06  GLTPFNGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
        W*02:01  GLTPSNGYTA ATWTRTAASS VGMNIPYDGA SYLVRNQELR SWTAADKAAQ
  W*03:01:01:01  GLTPSSGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
  W*03:01:01:02  GLTPSSGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
        W*04:01  GLTPSNGYTA ATWTRTAASS VGMNIPYDGA SYLVRNQELR SWTAADKAAQ
        W*05:01  GLTPSSGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ

  W*01:01:01:01  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
  W*01:01:01:02  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
  W*01:01:01:03  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
  W*01:01:01:04  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
  W*01:01:01:05  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
  W*01:01:01:06  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
        W*02:01  MPWRRNMQSC SKPTCREGGR SGSAKSLRMG RRRCTAQNPK RLT
  W*03:01:01:01  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK RLT
  W*03:01:01:02  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK RLT
        W*04:01  MPWRRNMQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK RLT
        W*05:01  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
```

To parse this file with Biopython, use

```
>>> from Bio import Align
>>> alignment = Align.read("W_prot.msf", "msf")
```

The parser skips all lines up to and including the line starting with "`MSF:`". The following lines (until the "`//`" demarcation) are read by the parser to verify the length of each sequence. The alignment section (after the "`//`" demarcation) is read by the parser and stored as an `Alignment` object:

```
>>> alignment
<Alignment object (11 rows x 99 columns) at ...>
>>> print(alignment)
W*01:01:0         0 GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*01:01:0         0 GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*01:01:0         0 GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*01:01:0         0 GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*01:01:0         0 GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*01:01:0         0 GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*02:01           0 GLTPSNGYTAATWTRTAASSVGMNIPYDGASYLVRNQELRSWTAADKAAQMPWRRNMQSC
W*03:01:0         0 GLTPSSGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*03:01:0         0 GLTPSSGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*04:01           0 GLTPSNGYTAATWTRTAASSVGMNIPYDGASYLVRNQELRSWTAADKAAQMPWRRNMQSC
W*05:01           0 GLTPSSGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC

W*01:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
W*01:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
W*01:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
W*01:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
W*01:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
```

(continues on next page)

(continued from previous page)

```
W*01:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
W*02:01          60 SKPTCREGGRSGSAKSLRMGRRRCTAQNPKRLT------ 93
W*03:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKRLT------ 93
W*03:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKRLT------ 93
W*04:01          60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKRLT------ 93
W*05:01          60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
```

The sequences and their names are stored in the `alignment.sequences` attribute:

```
>>> len(alignment.sequences)
11
>>> alignment.sequences[0].id
'W*01:01:01:01'
>>> alignment.sequences[0].seq
Seq('GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWR...PHL')
```

The alignment coordinates are stored in the `alignment.coordinates` attribute:

```
>>> print(alignment.coordinates)
[[ 0 93 99]
 [ 0 93 99]
 [ 0 93 99]
 [ 0 93 99]
 [ 0 93 99]
 [ 0 93 99]
 [ 0 93 93]
 [ 0 93 93]
 [ 0 93 93]
 [ 0 93 93]
 [ 0 93 99]]
```

Currently, Biopython does not support writing sequence alignments in the MSF format.

6.7.7 Exonerate

Exonerate is a generic program for pairwise sequence alignments [Slater2005]. The sequence alignments found by Exonerate can be output in a human-readable form, in the "cigar" (Compact Idiosyncratic Gapped Alignment Report) format, or in the "vulgar" (Verbose Useful Labelled Gapped Alignment Report) format. The user can request to include one or more of these formats in the output. The parser in `Bio.Align` can only parse alignments in the cigar or vulgar formats, and will not parse output that includes alignments in human-readable format.

The file `exn_22_m_cdna2genome_vulgar.exn` in the Biopython test suite is an example of an Exonerate output file showing the alignments in vulgar format:

```
Command line: [exonerate -m cdna2genome ../scer_cad1.fa /media/Waterloo/Downloads/
↪genomes/scer_s288c/scer_s288c.fa --bestn 3 --showalignment no --showcigar no --
↪showvulgar yes]
Hostname: [blackbriar]
vulgar: gi|296143771|ref|NM_001180731.1| 0 1230 + gi|330443520|ref|NC_001136.10| 1319275␣
↪1318045 - 6146 M 1 1 C 3 3 M 1226 1226
vulgar: gi|296143771|ref|NM_001180731.1| 1230 0 - gi|330443520|ref|NC_001136.10| 1318045␣
↪1319275 + 6146 M 129 129 C 3 3 M 1098 1098
```

(continues on next page)

(continued from previous page)

```
vulgar: gi|296143771|ref|NM_001180731.1| 0 516 + gi|330443688|ref|NC_001145.3| 85010␣
↪667216 + 518 M 11 11 G 1 0 M 15 15 G 2 0 M 4 4 G 1 0 M 1 1 G 1 0 M 8 8 G 4 0 M 17 17 5␣
↪0 2 I 0 168904 3 0 2 M 4 4 G 0 1 M 8 8 G 2 0 M 3 3 G 1 0 M 33 33 G 0 2 M 7 7 G 0 1 M␣
↪102 102 5 0 2 I 0 96820 3 0 2 M 14 14 G 0 2 M 10 10 G 2 0 M 5 5 G 0 2 M 10 10 G 2 0 M␣
↪4 4 G 0 1 M 20 20 G 1 0 M 15 15 G 0 1 M 5 5 G 3 0 M 4 4 5 0 2 I 0 122114 3 0 2 M 20 20␣
↪G 0 5 M 6 6 5 0 2 I 0 193835 3 0 2 M 12 12 G 0 2 M 5 5 G 1 0 M 7 7 G 0 2 M 1 1 G 0 1 M␣
↪12 12 C 75 75 M 6 6 G 1 0 M 4 4 G 0 1 M 2 2 G 0 1 M 3 3 G 0 1 M 41 41
-- completed exonerate analysis
```

This file includes three alignments. To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("exn_22_m_cdna2genome_vulgar.exn", "exonerate")
```

The dictionary `alignments.metadata` stores general information about these alignments, shown at the top of the output file:

```
>>> alignments.metadata
{'Program': 'exonerate',
 'Command line': 'exonerate -m cdna2genome ../scer_cad1.fa /media/Waterloo/Downloads/
↪genomes/scer_s288c/scer_s288c.fa --bestn 3 --showalignment no --showcigar no --
↪showvulgar yes',
 'Hostname': 'blackbriar'}
```

Now we can iterate over the alignments. The first alignment, with alignment score 6146.0, has no gaps:

```
>>> alignment = next(alignments)
>>> alignment.score
6146.0
>>> print(alignment.coordinates)
[[1319275 1319274 1319271 1318045]
 [      0       1       4    1230]]
>>> print(alignment)
gi|330443   1319275 ??????????????????????????????????????????????????????????
                  0 ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
gi|296143         0 ??????????????????????????????????????????????????????????
...
gi|330443   1318075 ????????????????????????????????? 1318045
               1200 |||||||||||||||||||||||||||||||||    1230
gi|296143      1200 ?????????????????????????????????    1230
```

Note that the target (the first sequence) in the printed alignment is on the reverse strand while the query (the second sequence) is on the forward strand, with the target coordinate decreasing and the query coordinate increasing. Printing this alignment in `exonerate` format using Python's built-in `format` function writes a vulgar line:

```
>>> print(format(alignment, "exonerate"))
vulgar: gi|296143771|ref|NM_001180731.1| 0 1230 + gi|330443520|ref|NC_001136.10| 1319275␣
↪1318045 - 6146 M 1 1 C 3 3 M 1226 1226
```

Using the `format` method allows us to request either a vulgar line (default) or a cigar line:

```
>>> print(alignment.format("exonerate", "vulgar"))
vulgar: gi|296143771|ref|NM_001180731.1| 0 1230 + gi|330443520|ref|NC_001136.10| 1319275␣
```

(continues on next page)

(continued from previous page)

```
↪1318045 - 6146 M 1 1 C 3 3 M 1226 1226

>>> print(alignment.format("exonerate", "cigar"))
cigar: gi|296143771|ref|NM_001180731.1| 0 1230 + gi|330443520|ref|NC_001136.10| 1319275␣
↪1318045 - 6146 M 1 M 3 M 1226
```

The vulgar line contains information about the alignment (in the section `M 1 1 C 3 3 M 1226`) that is missing from the cigar line `M 1 M 3 M 1226`. The vulgar line specifies that the alignment starts with a single aligned nucleotides, followed by three aligned nucleotides that form a codon (`C`), followed by 1226 aligned nucleotides. In the cigar line, we see a single aligned nucleotide, followed by three aligned nucleotides, followed by 1226 aligned nucleotides; it does not specify that the three aligned nucleotides form a codon. This information from the vulgar line is stored in the `operations` attribute:

```
>>> alignment.operations
bytearray(b'MCM')
```

See the Exonerate documentation for the definition of other operation codes.

Similarly, the `"vulgar"` or `"cigar"` argument can be used when calling `Bio.Align.write` to write a file with vulgar or cigar alignment lines.

We can print the alignment in BED and PSL format:

```
>>> print(format(alignment, "bed"))
gi|330443520|ref|NC_001136.10|  1318045 1319275 gi|296143771|ref|NM_001180731.1| 6146   -
↪   1318045 1319275 0   3   1226,3,1,   0,1226,1229,

>>> print(format(alignment, "psl"))
1230    0   0   0   0   0   0   0   -   gi|296143771|ref|NM_001180731.1|    1230    0  ␣
↪1230    gi|330443520|ref|NC_001136.10|  1319275 1318045 1319275 3   1226,3,1,   0,1226,
↪1229,    1318045,1319271,1319274,
```

The SAM format parser defines its own (optional) `operations` attribute (section *Sequence Alignment/Map (SAM)*), which is not quite consistent with the `operations` attribute defined in the Exonerate format parser. As the `operations` attribute is optional, we delete it before printing the alignment in SAM format:

```
>>> del alignment.operations
>>> print(format(alignment, "sam"))
gi|296143771|ref|NM_001180731.1|    16  gi|330443520|ref|NC_001136.10|  1318046 255␣
↪1226M3M1M   *   0   0   *   *   AS:i:6146
```

The third alignment contains four long gaps:

```
>>> alignment = next(alignments)  # second alignment
>>> alignment = next(alignments)  # third alignment
>>> print(alignment)
gi|330443     85010 ???????????-???????????????--????-?-????????----???????????
                  0 |||||||||||-|||||||||||||||--||||-|-||||||||----|||||||||||
gi|296143          0 ??????????????????????????????????????????????????????????

gi|330443     85061 ??????????????????????????????????????????????????????????
                 60 |||||----------------------------------------------------------
gi|296143         60 ?????----------------------------------------------------------
...
```

(continues on next page)

(continued from previous page)

```
gi|330443    666990 ??????????????????????????????????????????????????????????
             582000 ----------------------------------------------------|||||||||||
gi|296143       346 ----------------------------------------------------??????????

gi|330443    667050 ?????????-??????????????????????????????????????????????????
             582060 ||--|||||-|||||||--|-||||||||||||||||||||||||||||||||||||||||||
gi|296143       356 ??--?????????????--?-??????????????????????????????????????

gi|330443    667109 ??????????????????????????????????????????????????-?????
             582120 ||||||||||||||||||||||||||||||||||||||||||||||||||||||-||||-
gi|296143       411 ??????????????????????????????????????????????????????????-

gi|330443    667168 ?????????????????????????????????????????????????? 667216
             582180 ||-|||-|||||||||||||||||||||||||||||||||||||||||||| 582228
gi|296143       470 ??-???-??????????????????????????????????????????    516

>>> print(format(alignment, "exonerate"))
vulgar: gi|296143771|ref|NM_001180731.1| 0 516 + gi|330443688|ref|NC_001145.3|
85010 667216 + 518 M 11 11 G 1 0 M 15 15 G 2 0 M 4 4 G 1 0 M 1 1 G 1 0 M 8 8
 G 4 0 M 17 17 5 0 2 I 0 168904 3 0 2 M 4 4 G 0 1 M 8 8 G 2 0 M 3 3 G 1 0
 M 33 33 G 0 2 M 7 7 G 0 1 M 102 102 5 0 2 I 0 96820 3 0 2 M 14 14 G 0 2 M 10 10
 G 2 0 M 5 5 G 0 2 M 10 10 G 2 0 M 4 4 G 0 1 M 20 20 G 1 0 M 15 15 G 0 1 M 5 5
 G 3 0 M 4 4 5 0 2 I 0 122114 3 0 2 M 20 20 G 0 5 M 6 6 5 0 2 I 0 193835 3 0 2
 M 12 12 G 0 2 M 5 5 G 1 0 M 7 7 G 0 2 M 1 1 G 0 1 M 12 12 C 75 75 M 6 6 G 1 0
 M 4 4 G 0 1 M 2 2 G 0 1 M 3 3 G 0 1 M 41 41
```

6.7.8 NEXUS

The NEXUS file format [Maddison1997] is used by several programs to store phylogenetic information. This is an example of a file in the NEXUS format (available as `codonposset.nex` in the `Tests/Nexus` subdirectory in the Biopython distribution):

```
#NEXUS
[MacClade 4.05 registered to Computational Biologist, University]


BEGIN DATA;
       DIMENSIONS  NTAX=2 NCHAR=22;
       FORMAT DATATYPE=DNA  MISSING=? GAP=- ;
MATRIX
[                          10        20 ]
[                          .         .  ]

Aegotheles         AAAAAGGCATTGTGGTGGGAAT   [22]
Aerodramus         ?????????TTGTGGTGGGAAT   [13]
;
END;


BEGIN CODONS;
       CODONPOSSET * CodonPositions =
```

(continues on next page)

(continued from previous page)

```
               N: 1-10,
               1: 11-22\3,
               2: 12-22\3,
               3: 13-22\3;
       CODESET  * UNTITLED = Universal: all ;
END;
```

In general, files in the NEXUS format can be much more complex. `Bio.Align` relies heavily on NEXUS parser in `Bio.Nexus` (see Chapter *Phylogenetics with Bio.Phylo*) to extract `Alignment` objects from NEXUS files.

To read the alignment in this NEXUS file, use

```
>>> from Bio import Align
>>> alignment = Align.read("codonposset.nex", "nexus")
>>> print(alignment)
Aegothele         0 AAAAAGGCATTGTGGTGGGAAT 22
                  0 .........|||||||||||||| 22
Aerodramu         0 ??????????TTGTGGTGGGAAT 22

>>> alignment.shape
(2, 22)
```

The sequences are stored under the `sequences` attribute:

```
>>> alignment.sequences[0].id
'Aegotheles'
>>> alignment.sequences[0].seq
Seq('AAAAAGGCATTGTGGTGGGAAT')
>>> alignment.sequences[0].annotations
{'molecule_type': 'DNA'}
>>> alignment.sequences[1].id
'Aerodramus'
>>> alignment.sequences[1].seq
Seq('??????????TTGTGGTGGGAAT')
>>> alignment.sequences[1].annotations
{'molecule_type': 'DNA'}
```

To print this alignment in the NEXUS format, use

```
>>> print(format(alignment, "nexus"))
#NEXUS
begin data;
dimensions ntax=2 nchar=22;
format datatype=dna missing=? gap=-;
matrix
Aegotheles AAAAAGGCATTGTGGTGGGAAT
Aerodramus ??????????TTGTGGTGGGAAT
;
end;
```

Similarly, you can use `Align.write(alignment, "myfilename.nex", "nexus")` to write the alignment in the NEXUS format to the file `myfilename.nex`.

6.7.9 Tabular output from BLAST or FASTA

Alignment output in tabular output is generated by the FASTA aligner [Pearson1988] run with the `-m 8CB` or `-m 8CC` argument, or by BLAST [Altschul1990] run with the `-outfmt 7` argument.

The file `nucleotide_m8CC.txt` in the `Tests/Fasta` subdirectory of the Biopython source distribution is an example of an output file generated by FASTA with the `-m 8CC` argument:

```
# fasta36 -m 8CC seq/mgstm1.nt seq/gst.nlib
# FASTA 36.3.8h May, 2020
# Query: pGT875  - 657 nt
# Database: seq/gst.nlib
# Fields: query id, subject id, % identity, alignment length, mismatches, gap opens, q.␣
↪start, q. end, s. start, s. end, evalue, bit score, aln_code
# 12 hits found
pGT875  pGT875  100.00  657 0   0   1   657 38  694 4.6e-191    655.6   657M
pGT875  RABGLTR 79.10   646 135 0   1   646 34  679 1.6e-116    408.0   646M
pGT875  BTGST   59.56   413 167 21  176 594 228 655 1.9e-07 45.7   ␣
↪149M1D7M1I17M3D60M5I6M1I13M2I13M4I30M2I6M2D112M
pGT875  RABGSTB 66.93   127 42  8   159 289 157 287 3.2e-07 45.0   ␣
↪15M2I17M2D11M1I58M1I11M1D7M1D8M
pGT875  OCDHPR  91.30   23  2   1   266 289 2303    2325    0.012   29.7    17M1D6M
...
# FASTA processed 1 queries
```

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("nucleotide_m8CC.txt", "tabular")
```

Information shown in the file header is stored in the `metadata` attribute of `alignments`:

```
>>> alignments.metadata
{'Command line': 'fasta36 -m 8CC seq/mgstm1.nt seq/gst.nlib',
 'Program': 'FASTA',
 'Version': '36.3.8h May, 2020',
 'Database': 'seq/gst.nlib'}
```

Extract a specific alignment by iterating over the `alignments`. As an example, let's go to the fourth alignment:

```
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> print(alignment)
RABGSTB         156 ????????????????????????????????--??????????????????????????
                  0 |||||||||||||||--||||||||||||||||--||||||||||-||||||||||||
pGT875          158 ??????????????--????????????????????????????-???????????

RABGSTB         214 ??????????????????????????????????????????????????????????-?
                 60 |||||||||||||||||||||||||||||||||||||||||||||-||||||||||||-|
pGT875          215 ??????????????????????????????????????????????-????????????

RABGSTB         273 ??????-???????? 287
```

(continues on next page)

(continued from previous page)

```
               120 ||||||-|||||||| 135
pGT875         274 ???????????????? 289

>>> print(alignment.coordinates)
[[156 171 173 190 190 201 202 260 261 272 272 279 279 287]
 [158 173 173 190 192 203 203 261 261 272 273 280 281 289]]
>>> alignment.aligned
array([[[156, 171],
        [173, 190],
        [190, 201],
        [202, 260],
        [261, 272],
        [272, 279],
        [279, 287]],

       [[158, 173],
        [173, 190],
        [192, 203],
        [203, 261],
        [261, 272],
        [273, 280],
        [281, 289]]])
```

The sequence information of the target and query sequences is stored in the `target` and `query` attributes (as well as under `alignment.sequences`):

```
>>> alignment.target
SeqRecord(seq=Seq(None, length=287), id='RABGSTB', name='<unknown name>', description='
↪<unknown description>', dbxrefs=[])
>>> alignment.query
SeqRecord(seq=Seq(None, length=657), id='pGT875', name='<unknown name>', description='
↪<unknown description>', dbxrefs=[])
```

Information of the alignment is stored under the `annotations` attribute of the `alignment`:

```
>>> alignment.annotations
{'% identity': 66.93,
 'mismatches': 42,
 'gap opens': 8,
 'evalue': 3.2e-07,
 'bit score': 45.0}
```

BLAST in particular offers many options to customize tabular output by including or excluding specific columns; see the BLAST documentation for details. This information is stored in the dictionaries `alignment.annotations`, `alignment.target.annotations`, or `alignment.query.annotations`, as appropriate.

6.7.10 HH-suite output files

Alignment files in the `hhr` format are generated by `hhsearch` or `hhblits` in HH-suite [Steinegger2019]. As an example, see the file `2uvo_hhblits.hhr` in Biopython's test suite:

```
Query         2UVO:A|PDBID|CHAIN|SEQUENCE
Match_columns 171
No_of_seqs    1560 out of 4005
Neff          8.3
Searched_HMMs 34
Date          Fri Feb 15 16:34:13 2019
Command       hhblits -i 2uvoAh.fasta -d /pdb70

 No Hit                             Prob E-value P-value  Score    SS Cols Query HMM ␣
↪Template HMM
  1 2uvo_A Agglutinin isolectin 1; 100.0 3.7E-34 4.8E-38  210.3   0.0  171    1-171    ␣
↪1-171 (171)
  2 2wga   ; lectin (agglutinin);   99.9 1.1E-30 1.4E-34  190.4   0.0  162    2-169    ␣
↪2-163 (164)
  3 1ulk_A Lectin-C; chitin-bindin  99.8 5.2E-24 6.6E-28  148.2   0.0  120    1-124    ␣
↪2-121 (126)
...
 31 4z8i_A BBTPGRP3, peptidoglycan  79.6    0.12 1.5E-05   36.1   0.0   37    1-37     ␣
↪9-54  (236)
 32 1wga   ; lectin (agglutinin);   40.4     2.6 0.00029   25.9   0.0  106   54-163    ␣
↪11-116 (164)

No 1
>2uvo_A Agglutinin isolectin 1; carbohydrate-binding protein, hevein domain, chitin-
↪binding, GERM agglutinin, chitin-binding protein; HET: NDG NAG GOL; 1.40A {Triticum␣
↪aestivum} PDB: 1wgc_A* 2cwg_A* 2x3t_A* 4aml_A* 7wga_A 9wga_A 2wgc_A 1wgt_A 1k7t_A*␣
↪1k7v_A* 1k7u_A 2x52_A* 1t0w_A*
Probab=99.95  E-value=3.7e-34  Score=210.31  Aligned_cols=171  Identities=100% ␣
↪Similarity=2.050  Sum_probs=166.9

Q 2UVO:A|PDBID|C    1␣
↪ERCGEQGSNMECPNNLCCSQYGYCGMGGDYCGKGCQNGACWTSKRCGSQAGGATCTNNQCCSQYGYCGFGAEYCGAGCQG   80␣
↪(171)
Q Consensus         1 ~~cg~~~~~~~c~~~~CCs~~g~CG~~~~~c~~~c~~~~c~~~~~Cg~~~~~~~c~~~~CCs~~g~
↪CG~~~~~c~~~c~~   80 (171)
                      ||||++.++..||++.|||+|+|||.+.+||+++||.+.|++..+|+++++.++|....|||.
↪++||+.+.+||+.+||.
T Consensus         1 ~~cg~~~~~~~c~~~~CCS~~g~Cg~~~~~Cg~gC~~~~c~~~~~cg~~~~~~~c~~~~CCs~~g~
↪Cg~~~~~c~~~c~~   80 (171)
T 2uvo_A            1␣
↪ERCGEQGSNMECPNNLCCSQYGYCGMGGDYCGKGCQNGACWTSKRCGSQAGGATCTNNQCCSQYGYCGFGAEYCGAGCQG   80␣
↪(171)
T ss_dssp           ␣
↪CBCBGGGTTBBCGGGCEECTTSBEEBSHHHHSTTCCBSSCSSCCBCBGGGTTBCCSTTCEECTTSBEEBSHHHHSTTCCB
T ss_pred           ␣
↪CCCCCCCCCcCCCCCCeeCCCCeECCCccccccCCccccccccccccccCcccCCcccCCccccCCCceeCCCccccCCCccc
Confidence          ␣
↪799999999999999999999999999999999999999999999999999999999999999999999999999999999
```

(continues on next page)

(continued from previous page)

```
Q 2UVO:A|PDBID|C   81␣
↪GPCRADIKCGSQAGGKLCPNNLCCSQWGFCGLGSEFCGGGCQSGACSTDKPCGKDAGGRVCTNNYCCSKWGSCGIGPGYC  160␣
↪(171)
Q Consensus        81 ~~~~~~~~Cg~~~~~~~c~~~~CCS~~G~CG~~~~~C~~~Cq~~~c~~~~~Cg~~~~~~~c~~~~
↪CCS~~G~CG~~~~~C  160 (171)
                      +++++|+.|+...+++.||++.|||.|||||...+||+.+||+++|++|.+|++.+++++|..+.
↪|||+++-||+...||
T Consensus        81 ~~~~~~~~cg~~~~~~~c~~~~CCs~~g~CG~~~~~C~~gCq~~~c~~~~~cg~~~~~~~c~~~~
↪ccs~~g~Cg~~~~~C  160 (171)
T 2uvo_A           81␣
↪GPCRADIKCGSQAGGKLCPNNLCCSQWGFCGLGSEFCGGGCQSGACSTDKPCGKDAGGRVCTNNYCCSKWGSCGIGPGYC  160␣
↪(171)
T ss_dssp             ␣
↪SSCSSCCBCBGGGTTBCCGGGCEECTTSBEEBSHHHHSTTCCBSSCSSCCCCBTTTTTBCCSTTCEECTTSCEEBSHHHH
T ss_pred             ␣
↪ccccccccccccccCCCCCCCcccCCCCccCCCccccccCCCcCCccccccccccccccccccCCCCCCcCCCCEecCchhhc
Confidence            ␣
↪99999999999889999999999999999999999999999999999999999999999999999999999999999999


Q 2UVO:A|PDBID|C  161 GAGCQSGGCDG  171 (171)
Q Consensus       161 ~~gCq~~~c~~  171 (171)
                      +++||++.|||
T Consensus       161 ~~~cq~~~~~~  171 (171)
T 2uvo_A          161 GAGCQSGGCDG  171 (171)
T ss_dssp             STTCCBSSCC-
T ss_pred             ccccccCCCCC
Confidence            99999999986


No 2
...


No 32
>1wga   ; lectin (agglutinin); NMR {}
Probab=40.43  E-value=2.6  Score=25.90  Aligned_cols=106  Identities=20%  Similarity=0.
↪652  Sum_probs=54.7

Q 2UVO:A|PDBID|C   54␣
↪TCTNNQCCSQYGYCGFGAEYCGAGCQGGPCRADIKCGSQAGGKLCPNNLCCSQWGFCGLGSEFCGGGCQSGACSTDKPCG  133␣
↪(171)
Q Consensus        54 ~c~~~~CCs~~g~CG~~~~~c~~~c~~~~~~~~~~Cg~~~~~~~c~~~~CCS~~G~CG~~~~~C~~~
↪Cq~~~c~~~~~Cg  133 (171)
                      .|....||.....|......|...|....|.....|...  ...|....||.....|......|...
↪|....+.....|.
T Consensus        11 ~c~~~~cc~~~~~c~~~~~~c~~~c~~~~c~~~~~c~~~--~~~c~~~~cc~~~~~c~~~~~~c~~~
↪c~~~~c~~~~~c~   88 (164)
T 1wga             11 XCXXXXCCXXXXXCXXXXXXCXXXCXXXXCXXXXXCXXX--
↪XXXCXXXXCCXXXXXCXXXXXXCXXXCXXXXCXXXXXCX   88 (164)
```

(continues on next page)

(continued from previous page)

```
T ss_pred             ccccccccccccccccccccccccccccccccccccccc--
↪ccccccccccccccccccccccccccccccccccccccc
Confidence            34455666666666666566555543333223333321 ␣
↪2346666777777777776666665554433222333 3


Q 2UVO:A|PDBID|C  134 KDAGGRVCTNNYCCSKWGSCGIGPGYCGAG  163 (171)
Q Consensus       134 ~~~~~~~c~~~~CCS~~G~CG~~~~~C~~g  163 (171)
                      ..  ...|....||.....|......|...
T Consensus        89 ~~--~~~c~~~~cc~~~~~c~~~~~~c~~~  116 (164)
T 1wga             89 XX--XXXCXXXXCCXXXXXCXXXXXXCXXX  116 (164)
T ss_pred             cc--ccccccccccccccccccccccccc
Confidence            22  233444555555555555555544433


Done!
```

The file contains three sections:

- A header with general information about the alignments;
- A summary with one line for each of the alignments obtained;
- The alignments shown consecutively in detail.

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("2uvo_hhblits.hhr", "hhr")
```

Most of the header information is stored in the `metadata` attribute of `alignments`:

```
>>> alignments.metadata
{'Match_columns': 171,
 'No_of_seqs': (1560, 4005),
 'Neff': 8.3,
 'Searched_HMMs': 34,
 'Rundate': 'Fri Feb 15 16:34:13 2019',
 'Command line': 'hhblits -i 2uvoAh.fasta -d /pdb70'}
```

except the query name, which is stored as an attribute:

```
>>> alignments.query_name
'2UVO:A|PDBID|CHAIN|SEQUENCE'
```

as it will reappear in each of the alignments.

Iterate over the alignments:

```
>>> for alignment in alignments:
...     print(alignment.target.id)
...
2uvo_A
2wga
1ulk_A
```

(continues on next page)

(continued from previous page)

```
...
4z8i_A
1wga
```

Let's look at the first alignment in more detail:

```
>>> alignments = iter(alignments)
>>> alignment = next(alignments)
>>> alignment
<Alignment object (2 rows x 171 columns) at ...>
>>> print(alignment)
2uvo_A            0 ERCGEQGSNMECPNNLCCSQYGYCGMGGDYCGKGCQNGACWTSKRCGSQAGGATCTNNQC
                  0 ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
2UVO:A|PD         0 ERCGEQGSNMECPNNLCCSQYGYCGMGGDYCGKGCQNGACWTSKRCGSQAGGATCTNNQC

2uvo_A           60 CSQYGYCGFGAEYCGAGCQGGPCRADIKCGSQAGGKLCPNNLCCSQWGFCGLGSEFCGGG
                  60 ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
2UVO:A|PD        60 CSQYGYCGFGAEYCGAGCQGGPCRADIKCGSQAGGKLCPNNLCCSQWGFCGLGSEFCGGG

2uvo_A          120 CQSGACSTDKPCGKDAGGRVCTNNYCCSKWGSCGIGPGYCGAGCQSGGCDG 171
                  120 ||||||||||||||||||||||||||||||||||||||||||||||||||| 171
2UVO:A|PD       120 CQSGACSTDKPCGKDAGGRVCTNNYCCSKWGSCGIGPGYCGAGCQSGGCDG 171
```

The target and query sequences are stored in `alignment.sequences`. As these are pairwise alignments, we can also access them through `alignment.target` and `alignment.query`:

```
>>> alignment.target is alignment.sequences[0]
True
>>> alignment.query is alignment.sequences[1]
True
```

The ID of the query is set from the `alignments.query_name` (note that the query ID printed in the alignment in the `hhr` file is abbreviated):

```
>>> alignment.query.id
'2UVO:A|PDBID|CHAIN|SEQUENCE'
```

The ID of the target is taken from the sequence alignment block (the line starting with `T 2uvo_A`):

```
>>> alignment.target.id
'2uvo_A'
```

The sequence contents of the target and query are filled in from the information available in this alignment:

```
>>> alignment.target.seq
Seq('ERCGEQGSNMECPNNLCCSQYGYCGMGGDYCGKGCQNGACWTSKRCGSQAGGAT...CDG')
>>> alignment.query.seq
Seq('ERCGEQGSNMECPNNLCCSQYGYCGMGGDYCGKGCQNGACWTSKRCGSQAGGAT...CDG')
```

The sequence contents will be incomplete (a partially defined sequence; see Section *Sequences with partially defined sequence contents*) if the alignment does not extend over the full sequence.

The second line of this alignment block, starting with ">", shows the name and description of the Hidden Markov Model from which the target sequence was taken. These are stored under the keys `"hmm_name"` and `"hmm_description"` in

the `alignment.target.annotations` dictionary:

```
>>> alignment.target.annotations
{'hmm_name': '2uvo_A',
 'hmm_description': 'Agglutinin isolectin 1; carbohydrate-binding protein, hevein domain,
↪ chitin-binding, GERM agglutinin, chitin-binding protein; HET: NDG NAG GOL; 1.40A
↪{Triticum aestivum} PDB: 1wgc_A* 2cwg_A* 2x3t_A* 4aml_A* 7wga_A 9wga_A 2wgc_A 1wgt_A␣
↪1k7t_A* 1k7v_A* 1k7u_A 2x52_A* 1t0w_A*'}
```

The dictionary `alignment.target.letter_annotations` stores the target alignent consensus sequence, the secondary structure as predicted by PSIPRED, and the target secondary structure as determined by DSSP:

```
>>> alignment.target.letter_annotations
{'Consensus': '~~cg~~~~~~~c~~~~CCS~~g~Cg~~~~~Cg~gC~~~~c~~~~~cg~~~~~~~c~~~~CCs~~g~Cg~~~~~
↪c~~~c~~~~~~~~~~cg~~~~~~~c~~~~CCs~~g~CG~~~~~C~~gCq~~~c~~~~~cg~~~~~~~c~~~~ccs~~g~Cg~~~~~
↪C~~~cq~~~~~~',
 'ss_pred':
↪'CCCCCCCCCcCCCCCCeeCCCCeECCCccccccCCccccccccccccccCcccCCcccCCccccCCCceeCCCccccCCCccccccccccccccccccCCCCCC
↪',
 'ss_dssp':
↪'CBCBGGGTTBBCGGGCEECTTSBEEBSHHHHSTTCCBSSCSSCCBCBGGGTTBCCSTTCEECTTSBEEBSHHHHSTTCCBSSCSSCCBCBGGGTTBCCGG
↪'}
```

In this example, for the query sequence only the consensus sequence is available:

```
>>> alignment.query.letter_annotations
{'Consensus': '~~cg~~~~~~~c~~~~CCs~~g~CG~~~~~c~~~c~~~~c~~~~~Cg~~~~~~~c~~~~CCs~~g~CG~~~~~
↪c~~~c~~~~~~~~~~Cg~~~~~~~c~~~~CCS~~G~CG~~~~~C~~~Cq~~~c~~~~~Cg~~~~~~~c~~~~CCS~~G~CG~~~~~
↪C~~gCq~~~c~~'}
```

The `alignment.annotations` dictionary stores information about the alignment shown on the third line of the alignment block:

```
>>> alignment.annotations
{'Probab': 99.95,
 'E-value': 3.7e-34,
 'Score': 210.31,
 'Identities': 100.0,
 'Similarity': 2.05,
 'Sum_probs': 166.9}
```

Confidence values for the pairwise alignment are stored under the `"Confidence"` key in the `alignment.column_annotations` dictionary. This dictionary also stores the score for each column, shown between the query and the target section of each alignment block:

```
>>> alignment.column_annotations
{'column score': '||||++.++..||++.|||+|+|||.+.+||+++||.+.|++..+|+++++.++|....|||.++||+.+.
↪+||+.+||.+++++|+.|+...+++.||++.|||.|||||...+||+.+||+++|++|.+|++.+++++|..+.|||+++-||+...
↪||+++||++.|||',
 'Confidence':
↪'799999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999988999999
↪'}
```

6.7.11 A2M

A2M files are alignment files created by `align2model` or `hmmscore` in the SAM Sequence Alignment and Modeling Software System [Krogh1994], [Hughey1996]. An A2M file contains one multiple alignment. The A2M file format is similar to aligned FASTA (see section *Aligned FASTA*). However, to distinguish insertions from deletions, A2M uses both dashes and periods to represent gaps, and both upper and lower case characters in the aligned sequences. Matches are represented by upper case letters and deletions by dashes in alignment columns containing matches or deletions only. Insertions are represented by lower case letters, with gaps aligned to the insertion shown as periods. Header lines start with ">" followed by the name of the sequence, and optionally a description.

The file `probcons.a2m` in Biopython's test suite is an example of an A2M file (see section *Aligned FASTA* for the same alignment in aligned FASTA format):

```
>plas_horvu
D.VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG.VD.VSKISQEEYLTAPGETFSVTLTV...
↪PGTYGFYCEPHAGAGMVGKVT
V
>plas_chlre
-.VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG.VN.ADAISRDDYLNAPGETYSVKLTA...
↪AGEYGYYCEPHQGAGMVGKII
V
>plas_anava
-.
↪VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKsADlAKSLSHKQLLMSPGQSTSTTFPAdapAGEYTFYCEPHRGAGMVGKIT
V
>plas_proho
VqIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG.ES.APALSNTKLRIAPGSFYSVTLGT...
↪PGTYSFYCTPHRGAGMVGTIT
V
>azup_achcy
VhMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG.AE.A-------FKSKINENYKVTFTA...
↪PGVYGVKCTPHYGMGMVGVVE
V
```

To parse this alignment, use

```
>>> from Bio import Align
>>> alignment = Align.read("probcons.a2m", "a2m")
>>> alignment
<Alignment object (5 rows x 101 columns) at ...>
>>> print(alignment)
plas_horv         0 D-VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG-VD-VSKISQE
plas_chlr         0 --VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG-VN-ADAISRD
plas_anav         0 --VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKSADLAKSLSHK
plas_proh         0 VQIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG-ES-APALSNT
azup_achc         0 VHMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG-AE-A------

plas_horv        57 EYLTAPGETFSVTLTV---PGTYGFYCEPHAGAGMVGKVTV 95
plas_chlr        56 DYLNAPGETYSVKLTA---AGEYGYYCEPHQGAGMVGKIIV 94
plas_anav        58 QLLMSPGQSTSTTFPADAPAGEYTFYCEPHRGAGMVGKITV 99
plas_proh        56 KLRIAPGSFYSVTLGT---PGTYSFYCTPHRGAGMVGTITV 94
azup_achc        51 -FKSKINENYKVTFTA---PGVYGVKCTPHYGMGMVGVVEV 88
```

The parser analyzes the pattern of dashes, periods, and lower and upper case letters in the A2M file to determine if a column is an match/mismatch/deletion ("`D`") or an insertion ("`I`"). This information is stored under the `match` key of

the `alignment.column_annotations` dictionary:

```
>>> alignment.column_annotations
{'state':
↪'DIDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDIDDIDDDDDDDDDDDDDDDDDDDDDDDDIIIDDDDDDDDDDDDDDDDDDDDDD
↪'}
```

As the state information is stored in the `alignment`, we can print the alignment in the A2M format:

```
>>> print(format(alignment, "a2m"))
>plas_horvu
D.VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG.VD.VSKISQEEYLTAPGETFSVTLTV...
↪PGTYGFYCEPHAGAGMVGKVTV
>plas_chlre
-.VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG.VN.ADAISRDDYLNAPGETYSVKLTA...
↪AGEYGYYCEPHQGAGMVGKIIV
>plas_anava
-.
↪VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKsADlAKSLSHKQLLMSPGQSTSTTFPAdapAGEYTFYCEPHRGAGMVGKITV
>plas_proho
VqIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG.ES.APALSNTKLRIAPGSFYSVTLGT...
↪PGTYSFYCTPHRGAGMVGTITV
>azup_achcy
VhMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG.AE.A-------FKSKINENYKVTFTA...
↪PGVYGVKCTPHYGMGMVGVVEV
```

Similarly, the alignment can be written in the A2M format to an output file using `Align.write` (see section *Writing alignments*).

6.7.12 Mauve eXtended Multi-FastA (xmfa) format

Mauve [Darling2004] is a software package for constructing multiple genome alignments. These alignments are stored in the eXtended Multi-FastA (xmfa) format. Depending on how exactly `progressiveMauve` (the aligner program in Mauve) was called, the xmfa format is slightly different.

If `progressiveMauve` is called with a single sequence input file, as in

```
progressiveMauve combined.fasta  --output=combined.xmfa ...
```

where `combined.fasta` contains the genome sequences:

```
>equCab1
GAAAAGGAAAGTACGGCCCGGCCACTCCGGGTGTGTGCTAGGAGGGCTTA
>mm9
GAAGAGGAAAAGTAGATCCCTGGCGTCCGGAGCTGGGACGT
>canFam2
CAAGCCCTGCGCGCTCAGCCGGAGTGTCCCGGGCCCTGCTTTCCTTTTC
```

then the output file `combined.xmfa` is as follows:

```
#FormatVersion Mauve1
#Sequence1File  combined.fa
#Sequence1Entry 1
#Sequence1Format    FastA
```

(continues on next page)

(continued from previous page)

```
#Sequence2File  combined.fa
#Sequence2Entry 2
#Sequence2Format    FastA
#Sequence3File  combined.fa
#Sequence3Entry 3
#Sequence3Format    FastA
#BackboneFile   combined.xmfa.bbcols
> 1:2-49 - combined.fa
AAGCCCTCCTAGCACACACCCGGAGTGG-CCGGGCCGTACTTTCCTTTT
> 2:0-0 + combined.fa
-------------------------------------------------
> 3:2-48 + combined.fa
AAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGCTTTCCTTTT
=
> 1:1-1 + combined.fa
G
=
> 1:50-50 + combined.fa
A
=
> 2:1-41 + combined.fa
GAAGAGGAAAAGTAGATCCCTGGCGTCCGGAGCTGGGACGT
=
> 3:1-1 + combined.fa
C
=
> 3:49-49 + combined.fa
C
=
```

with numbers (1, 2, 3) referring to the input genome sequences for horse (`equCab1`), mouse (`mm9`), and dog (`canFam2`), respectively. This xmfa file consists of six alignment blocks, separated by = characters. Use `Align.parse` to extract these alignments:

```
>>> from Bio import Align
>>> alignments = Align.parse("combined.xmfa", "mauve")
```

The file header data are stored in the `metadata` attribute:

```
>>> alignments.metadata
{'FormatVersion': 'Mauve1',
 'BackboneFile': 'combined.xmfa.bbcols',
 'File': 'combined.fa'}
```

The `identifiers` attribute stores the sequence identifiers for the three sequences, which in this case is the three numbers:

```
>>> alignments.identifiers
['0', '1', '2']
```

These identifiers are used in the individual alignments:

```
>>> for alignment in alignments:
...     print([record.id for record in alignment.sequences])
...     print(alignment)
...     print("*******")
...
['0', '1', '2']
0                49 AAGCCCTCCTAGCACACACCCGGAGTGG-CCGGGCCGTACTTTCCTTTT  1
1                 0 ---------------------------------------------  0
2                 1 AAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGCTTTCCTTTT 48

******
['0']
0                 0 G 1

******
['0']
0                49 A 50

******
['1']
1                 0 GAAGAGGAAAAGTAGATCCCTGGCGTCCGGAGCTGGGACGT 41

******
['2']
2                 0 C 1

******
['2']
2                48 C 49

******
```

Note that only the first block is a real alignment; the other blocks contain only a single sequence. By including these blocks, the xmfa file contains the full sequence that was provided in the `combined.fa` input file.

If `progressiveMauve` is called with a separate input file for each genome, as in

```
progressiveMauve equCab1.fa canFam2.fa mm9.fa --output=separate.xmfa ...
```

where each Fasta file contains the genome sequence for one species only, then the output file `separate.xmfa` is as follows:

```
#FormatVersion Mauve1
#Sequence1File  equCab1.fa
#Sequence1Format    FastA
#Sequence2File  canFam2.fa
#Sequence2Format    FastA
#Sequence3File  mm9.fa
#Sequence3Format    FastA
#BackboneFile   separate.xmfa.bbcols
> 1:1-50 - equCab1.fa
TAAGCCCTCCTAGCACACACCCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC
> 2:1-49 + canFam2.fa
CAAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC
```

(continues on next page)

(continued from previous page)

```
> 3:1-19 - mm9.fa
---------------------------------GGATCTACTTTTCCTCTTC
=
> 3:20-41 + mm9.fa
CTGGCGTCCGGAGCTGGGACGT
=
```

The identifiers `equCab1` for horse, `mm9` for mouse, and `canFam2` for dog are now shown explicitly in the output file. This xmfa file consists of two alignment blocks, separated by = characters. Use `Align.parse` to extract these alignments:

```
>>> from Bio import Align
>>> alignments = Align.parse("separate.xmfa", "mauve")
```

The file header data now does not include the input file name:

```
>>> alignments.metadata
{'FormatVersion': 'Mauve1',
 'BackboneFile': 'separate.xmfa.bbcols'}
```

The `identifiers` attribute stores the sequence identifiers for the three sequences:

```
>>> alignments.identifiers
['equCab1.fa', 'canFam2.fa', 'mm9.fa']
```

These identifiers are used in the individual alignments:

```
>>> for alignment in alignments:
...     print([record.id for record in alignment.sequences])
...     print(alignment)
...     print("*******")
...
['equCab1.fa', 'canFam2.fa', 'mm9.fa']
equCab1.f        50 TAAGCCCTCCTAGCACACACCCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC  0
canFam2.f         0 CAAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC 49
mm9.fa           19 ---------------------------------GGATCTACTTTTCCTCTTC  0

******
['mm9.fa']
mm9.fa           19 CTGGCGTCCGGAGCTGGGACGT 41

******
```

To output the alignments in Mauve format, use `Align.write`:

```
>>> from io import StringIO
>>> stream = StringIO()
>>> alignments = Align.parse("separate.xmfa", "mauve")
>>> Align.write(alignments, stream, "mauve")
2
>>> print(stream.getvalue())
#FormatVersion Mauve1
#Sequence1File  equCab1.fa
#Sequence1Format    FastA
```

(continues on next page)

(continued from previous page)

```
#Sequence2File  canFam2.fa
#Sequence2Format    FastA
#Sequence3File  mm9.fa
#Sequence3Format    FastA
#BackboneFile   separate.xmfa.bbcols
> 1:1-50 - equCab1.fa
TAAGCCCTCCTAGCACACACCCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC
> 2:1-49 + canFam2.fa
CAAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC
> 3:1-19 - mm9.fa
--------------------------------GGATCTACTTTTCCTCTTC
=
> 3:20-41 + mm9.fa
CTGGCGTCCGGAGCTGGGACGT
=
```

Here, the writer makes use of the information stored in `alignments.metadata` and `alignments.identifiers` to create this format. If your `alignments` object does not have these attributes, you can provide them as keyword arguments to `Align.write`:

```
>>> stream = StringIO()
>>> alignments = Align.parse("separate.xmfa", "mauve")
>>> metadata = alignments.metadata
>>> identifiers = alignments.identifiers
>>> alignments = list(alignments)  # this drops the attributes
>>> alignments.metadata
Traceback (most recent call last):
  ...
AttributeError: 'list' object has no attribute 'metadata'
>>> alignments.identifiers
Traceback (most recent call last):
  ...
AttributeError: 'list' object has no attribute 'identifiers'
>>> Align.write(alignments, stream, "mauve", metadata=metadata, identifiers=identifiers)
2
>>> print(stream.getvalue())
#FormatVersion Mauve1
#Sequence1File  equCab1.fa
#Sequence1Format    FastA
#Sequence2File  canFam2.fa
#Sequence2Format    FastA
#Sequence3File  mm9.fa
#Sequence3Format    FastA
#BackboneFile   separate.xmfa.bbcols
> 1:1-50 - equCab1.fa
TAAGCCCTCCTAGCACACACCCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC
> 2:1-49 + canFam2.fa
CAAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC
> 3:1-19 - mm9.fa
--------------------------------GGATCTACTTTTCCTCTTC
=
> 3:20-41 + mm9.fa
```

(continues on next page)

(continued from previous page)

```
CTGGCGTCCGGAGCTGGGACGT
=
```

Python does not allow you to add these attributes to the `alignments` object directly, as in this example it was converted to a plain list. However, you can construct an `Alignments` object and add attributes to it (see Section *The Alignments class*):

```
>>> alignments = Align.Alignments(alignments)
>>> alignments.metadata = metadata
>>> alignments.identifiers = identifiers
>>> stream = StringIO()
>>> Align.write(alignments, stream, "mauve", metadata=metadata, identifiers=identifiers)
2
>>> print(stream.getvalue())
#FormatVersion Mauve1
#Sequence1File  equCab1.fa
#Sequence1Format    FastA
#Sequence2File  canFam2.fa
#Sequence2Format    FastA
#Sequence3File  mm9.fa
#Sequence3Format    FastA
#BackboneFile   separate.xmfa.bbcols
> 1:1-50 - equCab1.fa
TAAGCCCTCCTAGCACACACCCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC
> 2:1-49 + canFam2.fa
CAAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC
> 3:1-19 - mm9.fa
---------------------------------GGATCTACTTTTCCTCTTC
=
> 3:20-41 + mm9.fa
CTGGCGTCCGGAGCTGGGACGT
=
```

When printing a single alignment in `Mauve` format, use keyword arguments to provide the metadata and identifiers:

```
>>> alignment = alignments[0]
>>> print(alignment.format("mauve", metadata=metadata, identifiers=identifiers))
> 1:1-50 - equCab1.fa
TAAGCCCTCCTAGCACACACCCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC
> 2:1-49 + canFam2.fa
CAAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC
> 3:1-19 - mm9.fa
---------------------------------GGATCTACTTTTCCTCTTC
=
```

6.7.13 Sequence Alignment/Map (SAM)

Files in the Sequence Alignment/Map (SAM) format [Li2009] store pairwise sequence alignments, usually of next-generation sequencing data against a reference genome. The file `ex1.sam` in Biopython's test suite is an example of a minimal file in the SAM format. Its first few lines are as follows:

```
EAS56_57:6:190:289:82   69      chr1    100     0       *       =       100     0       ␣
↪CTCAAGGTTGTTGCAAGGGGGTCTATGTGAACAAA     <<<7<<<;<<<<<<<<8;;<7;4<;<;;;;;94<;    ␣
↪MF:i:192
EAS56_57:6:190:289:82   137     chr1    100     73      35M     =       100     0       ␣
↪AGGGGTGCAGAGCCGAGTCACGGGGTTGCCAGCAC     <<<<<<;<<<<<<<<<<;<<;<<<<;8<6;9;;2;    ␣
↪MF:i:64 Aq:i:0  NM:i:0  UQ:i:0  H0:i:1  H1:i:0
EAS51_64:3:190:727:308  99      chr1    103     99      35M     =       263     195     ␣
↪GGTGCAGAGCCGAGTCACGGGGTTGCCAGCACAGG     <<<<<<<<<<<<<<<<<<<<<<<<<<<<<::<<<844    ␣
↪MF:i:18 Aq:i:73 NM:i:0  UQ:i:0  H0:i:1  H1:i:0
...
```

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("ex1.sam", "sam")
>>> alignment = next(alignments)
```

The `flag` of the first line is 69. According to the SAM/BAM file format specification, lines for which the flag contains the bitwise flag 4 are unmapped. As 69 has the bit corresponding to this position set to True, this sequence is unmapped and was not aligned to the genome (in spite of the first line showing `chr1`). The target of this alignment (or the first item in `alignment.sequences`) is therefore `None`:

```
>>> alignment.flag
69
>>> bin(69)
'0b1000101'
>>> bin(4)
'0b100'
>>> if alignment.flag & 4:
...     print("unmapped")
... else:
...     print("mapped")
...
unmapped
>>> alignment.sequences
[None, SeqRecord(seq=Seq('CTCAAGGTTGTTGCAAGGGGGTCTATGTGAACAAA'), id='EAS56_
↪57:6:190:289:82', name='<unknown name>', description='', dbxrefs=[])]
>>> alignment.target is None
True
```

The second line represents an alignment to chromosome 1:

```
>>> alignment = next(alignments)
>>> if alignment.flag & 4:
...     print("unmapped")
... else:
...     print("mapped")
...
```

(continues on next page)

(continued from previous page)

```
mapped
>>> alignment.target
SeqRecord(seq=None, id='chr1', name='<unknown name>', description='', dbxrefs=[])
```

As this SAM file does not store the genome sequence information for each alignment, we cannot print the alignment. However, we can print the alignment information in SAM format or any other format (such as BED, see section *Browser Extensible Data (BED)*) that does not require the target sequence information:

```
>>> format(alignment, "sam")
'EAS56_57:6:190:289:82\t137\tchr1\t100\t73\t35M\t=\t100\t0\
↪tAGGGGTGCAGAGCCGAGTCACGGGGTTGCCAGCAC\t<<<<<<;<<<<<<<<<<;<<;<<<<;8<6;9;;2;\tMF:i:64\
↪tAq:i:0\tNM:i:0\tUQ:i:0\tH0:i:1\tH1:i:0\n'
>>> format(alignment, "bed")
'chr1\t99\t134\tEAS56_57:6:190:289:82\t0\t+\t99\t134\t0\t1\t35,\t0,\n'
```

However, we cannot print the alignment in PSL format (see section *Pattern Space Layout (PSL)*) as that would require knowing the size of the target sequence chr1:

```
>>> format(alignment, "psl")
Traceback (most recent call last):
 ...
TypeError: ...
```

If you know the size of the target sequences, you can set them by hand:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> target = SeqRecord(Seq(None, length=1575), id="chr1")
>>> alignment.target = target
>>> format(alignment, "psl")
'35\t0\t0\t0\t0\t0\t0\t0\t+\tEAS56_57:6:190:289:82\t35\t0\t35\tchr1\t1575\t99\t134\t1\
↪t35,\t0,\t99,\n'
```

The file `ex1_header.sam` in Biopython's test suite contains the same alignments, but now also includes a header. Its first few lines are as follows:

```
@HD\tVN:1.3\tSO:coordinate
@SQ\tSN:chr1\tLN:1575
@SQ\tSN:chr2\tLN:1584
EAS56_57:6:190:289:82   69      chr1    100     0       *       =       100     0       ␣
↪CTCAAGGTTGTTGCAAGGGGGTCTATGTGAACAAA     <<<7<<<;<<<<<<<<8;;<7;4<;<;;;;;94<;    ␣
↪MF:i:192
...
```

The header stores general information about the alignments, including the size of the target chromosomes. The target information is stored in the `targets` attribute of the `alignments` object:

```
>>> from Bio import Align
>>> alignments = Align.parse("ex1_header.sam", "sam")
>>> len(alignments.targets)
2
>>> alignments.targets[0]
SeqRecord(seq=Seq(None, length=1575), id='chr1', name='<unknown name>', description='',␣
```

(continues on next page)

(continued from previous page)

```
↪dbxrefs=[])
>>> alignments.targets[1]
SeqRecord(seq=Seq(None, length=1584), id='chr2', name='<unknown name>', description='',␣
↪dbxrefs=[])
```

Other information provided in the header is stored in the `metadata` attribute:

```
>>> alignments.metadata
{'HD': {'VN': '1.3', 'SO': 'coordinate'}}
```

With the target information, we can now also print the alignment in PSL format:

```
>>> alignment = next(alignments)  # the unmapped sequence; skip it
>>> alignment = next(alignments)
>>> format(alignment, "psl")
'35\t0\t0\t0\t0\t0\t0\t0\t+\tEAS56_57:6:190:289:82\t35\t0\t35\tchr1\t1575\t99\t134\t1\
↪t35,\t0,\t99,\n'
```

We can now also print the alignment in human-readable form, but note that the target sequence contents is not available from this file:

```
>>> print(alignment)
chr1             99 ???????????????????????????????????? 134
                  0 ......................................  35
EAS56_57:         0 AGGGGTGCAGAGCCGAGTCACGGGGTTGCCAGCAC  35
```

Alignments in the file `sam1.sam` in the Biopython test suite contain an additional `MD` tag that shows how the query sequence differs from the target sequence:

```
@SQ     SN:1    LN:239940
@PG     ID:bwa  PN:bwa  VN:0.6.2-r126
HWI-1KL120:88:D0LRBACXX:1:1101:1780:2146        77      *       0       0       *       ␣
↪*       0       0       ␣
↪GATGGGAAACCCATGGCCGAGTGGGAAGAAACCAGCTGAGGTCACATCACCAGAGGAGGGAGAGTGTGGCCCCTGACTCAGTCCATCAGCTTGTGGAGCTG␣
↪  @=?DDDDBFFFF7A;E?GGEGE8BB?FF?F>G@F=GIIDEIBCFF<FEFEC@EEEE2?8B8/=@((-;?@2<B9@##########
↪################
...
HWI-1KL120:88:D0LRBACXX:1:1101:2852:2134        137     1       136186  25      101M    ␣
↪=       136186  0       ␣
↪TCACGGTGGCCTGTTGAGGCAGGGGCTCACGCTGACCTCTCTCGGCGTGGGAGGGGCCGGTGTGAGGCAAGGGCTCACGCTGACCTCTCTCGGCGTGGGAG␣
↪  @C@FFFDFHGHHHJJJIJJJJIJJJGEDHHGGHGBGIIGIIAB@GEE=BDBBCCDD@D@B7@;@DDD?<A?DD728:>8()009>
↪:>>C@>5??B######   XT:A:U  NM:i:5  SM:i:25 AM:i:0  X0:i:1  X1:i:0  XM:i:5  XO:i:0 ␣
↪XG:i:0  MD:Z:25G14G2C34A12A9
```

The parser reconstructs the local genome sequence from the `MD` tag, allowing us to see the target sequence explicitly when printing the alignment:

```
>>> from Bio import Align
>>> alignments = Align.parse("sam1.sam", "sam")
>>> for alignment in alignments:
...     if not alignment.flag & 4:  # Skip the unmapped lines
...         break
...
```

(continues on next page)

(continued from previous page)

```
>>> alignment
<Alignment object (2 rows x 101 columns) at ...>
>>> print(alignment)
1            136185 TCACGGTGGCCTGTTGAGGCAGGGGGTCACGCTGACCTCTGTCCGCGTGGGAGGGGCCGG
                 0 |||||||||||||||||||||||.|||||||||||||.||.|||||||||||||||
HWI-1KL12        0 TCACGGTGGCCTGTTGAGGCAGGGGCTCACGCTGACCTCTCTCGGCGTGGGAGGGGCCGG

1            136245 TGTGAGGCAAGGGCTCACACTGACCTCTCTCAGCGTGGGAG 136286
                60 ||||||||||||||||||.||||||||||.|||||||||    101
HWI-1KL12       60 TGTGAGGCAAGGGCTCACGCTGACCTCTCTCGGCGTGGGAG    101
```

SAM files may include additional information to distinguish simple sequence insertions and deletions from skipped regions of the genome (e.g. introns), hard and soft clipping, and padded sequence regions. As this information cannot be stored in the `coordinates` attribute of an `Alignment` object, and is stored in a dedicated `operations` attribute instead. Let's use the third alignment in this SAM file as an example:

```
>>> from Bio import Align
>>> alignments = Align.parse("dna_rna.sam", "sam")
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> print(format(alignment, "SAM"))
NR_111921.1 0   chr3    48663768    0   46M1827N82M3376N76M12H  *   0   0  ␣
↪CACGAGAGGAGCGGAGGCGAGGGGTGAACGCGGAGCACTCCAATCGCTCCCAACTAGAGGTCCACCCAGGACCCAGAGACCTGGATTTGAGGCTGCTGGGC
↪   *   AS:i:1000   NM:i:0

>>> print(alignment.coordinates)
[[48663767 48663813 48665640 48665722 48669098 48669174]
 [       0       46       46      128      128      204]]
>>> alignment.operations
bytearray(b'MNMNM')
>>> alignment.query.annotations["hard_clip_right"]
12
```

In this alignment, the cigar string `63M1062N75M468N43M` defines 46 aligned nucleotides, an intron of 1827 nucleotides, 82 aligned nucleotides, an intron of 3376 nucleotides, 76 aligned nucleotides, and 12 hard-clipped nucleotides. These operations are shown in the `operations` attribute, except for hard-clipping, which is stored in `alignment.query.annotations["hard_clip_right"]` (or `alignment.query.annotations["hard_clip_left"]`, if applicable) instead.

To write a SAM file with alignments created from scratch, use an `Alignments` (plural) object (see Section *The Alignments class*) to store the alignments as well as the metadata and targets:

```
>>> from io import StringIO
>>> import numpy as np

>>> from Bio import Align
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord

>>> alignments = Align.Alignments()

>>> seq1 = Seq(None, length=10000)
```

(continues on next page)

(continued from previous page)

```
>>> target1 = SeqRecord(seq1, id="chr1")
>>> seq2 = Seq(None, length=15000)
>>> target2 = SeqRecord(seq2, id="chr2")
>>> alignments.targets = [target1, target2]
>>> alignments.metadata = {"HD": {"VN": "1.3", "SO": "coordinate"}}

>>> seqA = Seq(None, length=20)
>>> queryA = SeqRecord(seqA, id="readA")
>>> sequences = [target1, queryA]
>>> coordinates = np.array([[4300, 4320], [0, 20]])
>>> alignment = Align.Alignment(sequences, coordinates)
>>> alignments.append(alignment)

>>> seqB = Seq(None, length=25)
>>> queryB = SeqRecord(seqB, id="readB")
>>> sequences = [target1, queryB]
>>> coordinates = np.array([[5900, 5925], [25, 0]])
>>> alignment = Align.Alignment(sequences, coordinates)
>>> alignments.append(alignment)

>>> seqC = Seq(None, length=40)
>>> queryC = SeqRecord(seqC, id="readC")
>>> sequences = [target2, queryC]
>>> coordinates = np.array([[12300, 12318], [0, 18]])
>>> alignment = Align.Alignment(sequences, coordinates)
>>> alignments.append(alignment)

>>> stream = StringIO()
>>> Align.write(alignments, stream, "sam")
3
>>> print(stream.getvalue())
@HD VN:1.3  SO:coordinate
@SQ SN:chr1 LN:10000
@SQ SN:chr2 LN:15000
readA   0   chr1    4301    255 20M *   0   0   *   *
readB   16  chr1    5901    255 25M *   0   0   *   *
readC   0   chr2    12301   255 18M22S  *   0   0   *       *
```

6.7.14 Browser Extensible Data (BED)

BED (Browser Extensible Data) files are typically used to store the alignments of gene transcripts to the genome. See the description from UCSC for a full explanation of the BED format.

BED files have three required fields and nine optional fields. The file `bed12.bed` in subdirectory `Tests/Blat` is an example of a BED file with 12 fields:

```
chr22   1000    5000    mRNA1   960 +   1200    4900    255,0,0 2   567,488,    0,3512,
chr22   2000    6000    mRNA2   900 -   2300    5960    0,255,0 2   433,399,    0,3601,
```

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("bed12.bed", "bed")
>>> len(alignments)
2
>>> for alignment in alignments:
...     print(alignment.coordinates)
...
[[1000 1567 4512 5000]
 [   0  567  567 1055]]
[[2000 2433 5601 6000]
 [ 832  399  399    0]]
```

Note that the first sequence ("mRNA1") was mapped to the forward strand, while the second sequence ("mRNA2") was mapped to the reverse strand.

As a BED file does not store the length of each chromosome, the length of the target sequence is set to its maximum:

```
>>> alignment.target
SeqRecord(seq=Seq(None, length=9223372036854775807), id='chr22', name='<unknown name>',␣
↪description='', dbxrefs=[])
```

The length of the query sequence can be inferred from its alignment information:

```
>>> alignment.query
SeqRecord(seq=Seq(None, length=832), id='mRNA2', name='<unknown name>', description='',␣
↪dbxrefs=[])
```

The alignment score (field 5) and information stored in fields 7-9 (referred to as `thickStart`, `thickEnd`, and `itemRgb` in the BED format specification) are stored as attributes on the `alignment` object:

```
>>> alignment.score
900.0
>>> alignment.thickStart
2300
>>> alignment.thickEnd
5960
>>> alignment.itemRgb
'0,255,0'
```

To print an alignment in the BED format, you can use Python's built-in `format` function:

```
>>> print(format(alignment, "bed"))
chr22   2000    6000    mRNA2   900 -   2300    5960    0,255,0 2   433,399,    0,3601,
```

or you can use the `format` method of the `alignment` object. This allows you to specify the number of fields to be written as the `bedN` keyword argument:

```
>>> print(alignment.format("bed"))
chr22   2000    6000    mRNA2   900 -   2300    5960    0,255,0 2   433,399,    0,3601,

>>> print(alignment.format("bed", 3))
chr22   2000    6000

>>> print(alignment.format("bed", 6))
chr22   2000    6000    mRNA2   900 -
```

The same keyword argument can be used with `Align.write`:

```
>>> Align.write(alignments, "mybed3file.bed", "bed", bedN=3)
2
>>> Align.write(alignments, "mybed6file.bed", "bed", bedN=6)
2
>>> Align.write(alignments, "mybed12file.bed", "bed")
2
```

6.7.15 bigBed

The bigBed file format is an indexed binary version of a BED file *Browser Extensible Data (BED)*. To create a bigBed file, you can either use the `bedToBigBed` program from UCSC (`) <https://genome.ucsc.edu/goldenPath/help/bigBed.html>`__. or you can use Biopython for it by calling the `Bio.Align.write` function with `fmt="bigbed"`. While the two methods should result in identical bigBed files, using `bedToBigBed` is much faster and may be more reliable, as it is the gold standard. As bigBed files come with a built-in index, it allows you to quickly search a specific genomic region.

As an example, let's parse the bigBed file `dna_rna.bb`, available in the `Tests/Blat` subdirectory in the Biopython distribution:

```
>>> from Bio import Align
>>> alignments = Align.parse("dna_rna.bb", "bigbed")
>>> len(alignments)
4
>>> print(alignments.declaration)
table bed
"Browser Extensible Data"
(
   string          chrom;          "Reference sequence chromosome or scaffold"
   uint            chromStart;     "Start position in chromosome"
   uint            chromEnd;       "End position in chromosome"
   string          name;           "Name of item."
   uint            score;          "Score (0-1000)"
   char[1]         strand;         "+ or - for strand"
   uint            thickStart;     "Start of where display should be thick (start codon)"
   uint            thickEnd;       "End of where display should be thick (stop codon)"
   uint            reserved;       "Used as itemRgb as of 2004-11-22"
   int             blockCount;     "Number of blocks"
   int[blockCount] blockSizes;     "Comma separated list of block sizes"
   int[blockCount] chromStarts;    "Start positions relative to chromStart"
)
```

The `declaration` contains the specification of the columns, in AutoSql format, that was used to create the bigBed file. Target sequences (typically, the chromosomes against which the sequences were aligned) are stored in the `targets` attribute. In the bigBed format, only the identifier and the size of each target is stored. In this example, there is only a single chromosome:

```
>>> alignments.targets
[SeqRecord(seq=Seq(None, length=198295559), id='chr3', name='<unknown name>',␣
↪description='<unknown description>', dbxrefs=[])]
```

Let's look at the individual alignments. The alignment information is stored in the same way as for a BED file (see section *Browser Extensible Data (BED)*):

```
>>> alignment = next(alignments)
>>> alignment.target.id
'chr3'
>>> alignment.query.id
'NR_046654.1'
>>> alignment.coordinates
array([[42530895, 42530958, 42532020, 42532095, 42532563, 42532606],
       [     181,      118,      118,       43,       43,        0]])
>>> alignment.thickStart
42530895
>>> alignment.thickEnd
42532606
>>> print(alignment)
chr3       42530895 ??????????????????????????????????????????????????????????
               0 ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
NR_046654      181 ??????????????????????????????????????????????????????????

chr3       42530955 ??????????????????????????????????????????????????????????
              60 |||-----------------------------------------------------------
NR_046654      121 ???-----------------------------------------------------------
...
chr3       42532515 ??????????????????????????????????????????????????????????
            1620 ------------------------------------------------------||||||||||||
NR_046654       43 ------------------------------------------------------???????????

chr3       42532575 ????????????????????????????????? 42532606
            1680 |||||||||||||||||||||||||||||||||     1711
NR_046654       31 ?????????????????????????????????        0
```

The default bigBed format does not store the sequence contents of the target and query. If these are available elsewhere (for example, a Fasta file), you can set `alignment.target.seq` and `alignment.query.seq` to show the sequence contents when printing the alignment, or to write the alignment in formats that require the sequence contents (such as Clustal, see section *ClustalW*). The test script `test_Align_bigbed.py` in the `Tests` subdirectory in the Biopython distribution gives some examples on how to do that.

Now let's see how to search for a sequence region. These are the sequences stored in the bigBed file, printed in BED format (see section *Browser Extensible Data (BED)*):

```
>>> for alignment in alignments:
...     print(format(alignment, "bed"))
...
chr3    42530895    42532606    NR_046654.1 1000    -   42530895    42532606    0   3  ␣
↪63,75,43,   0,1125,1668,

chr3    42530895    42532606    NR_046654.1_modified    978 -   42530895    42532606   ␣
↪0   5   27,36,17,56,43, 0,27,1125,1144,1668,

chr3    48663767    48669174    NR_111921.1 1000    +   48663767    48669174    0   3  ␣
↪46,82,76,   0,1873,5331,

chr3    48663767    48669174    NR_111921.1_modified    972 +   48663767    48669174   ␣
↪0   5   28,17,76,6,76,  0,29,1873,1949,5331,
```

Use the `search` method on the `alignments` object to find regions on chr3 between positions 48000000 and 49000000.

This method returns an iterator:

```
>>> selected_alignments = alignments.search("chr3", 48000000, 49000000)
>>> for alignment in selected_alignments:
...     print(alignment.query.id)
...
NR_111921.1
NR_111921.1_modified
```

The chromosome name may be `None` to include all chromosomes, and the start and end positions may be `None` to start searching from position 0 or to continue searching until the end of the chromosome, respectively.

Writing alignments in the bigBed format is as easy as calling `Bio.Align.write`:

```
>>> Align.write(alignments, "output.bb", "bigbed")
```

You can specify the number of BED fields to be included in the bigBed file. For example, to write a BED6 file, use

```
>>> Align.write(alignments, "output.bb", "bigbed", bedN=6)
```

Same as for `bedToBigBed`, you can include additional columns in the bigBed output. Suppose the file `bedExample2.as` (available in the `Tests/Blat` subdirectory of the Biopython distribution) stores the declaration of the included BED fields in AutoSql format. We can read this declaration as follows:

```
>>> from Bio.Align import bigbed
>>> with open("bedExample2.as") as stream:
...     autosql_data = stream.read()
...
>>> declaration = bigbed.AutoSQLTable.from_string(autosql_data)
>>> type(declaration)
<class 'Bio.Align.bigbed.AutoSQLTable'>
>>> print(declaration)
table hg18KGchr7
"UCSC Genes for chr7 with color plus GeneSymbol and SwissProtID"
(
   string  chrom;         "Reference sequence chromosome or scaffold"
   uint    chromStart;    "Start position of feature on chromosome"
   uint    chromEnd;      "End position of feature on chromosome"
   string  name;          "Name of gene"
   uint    score;         "Score"
   char[1] strand;        "+ or - for strand"
   uint    thickStart;    "Coding region start"
   uint    thickEnd;      "Coding region end"
   uint    reserved;      "Green on + strand, Red on - strand"
   string  geneSymbol;    "Gene Symbol"
   string  spID;          "SWISS-PROT protein Accession number"
)
```

Now we can write a bigBed file with the 9 BED fields plus the additional fields `geneSymbol` and `spID` by calling

```
>>> Align.write(
...     alignments,
...     "output.bb",
...     "bigbed",
...     bedN=9,
```

(continues on next page)

(continued from previous page)

```
...     declaration=declaration,
...     extraIndex=["name", "geneSymbol"],
... )
```

Here, we also requested to include additional indices on the `name` and `geneSymbol` in the bigBed file. `Align.write` expects to find the keys `geneSymbol` and `spID` in the `alignment.annotations` dictionary. Please refer to the test script `test_Align_bigbed.py` in the `Tests` subdirectory in the Biopython distribution for more examples of writing alignment files in the bigBed format.

Optional arguments are `compress` (default value is `True`), `blockSize` (default value is 256), and `itemsPerSlot` (default value is 512). See the documentation of UCSC's `bedToBigBed` program for a description of these arguments. Searching a `bigBed` file can be faster by using `compress=False` and `itemsPerSlot=1` when creating the bigBed file.

6.7.16 Pattern Space Layout (PSL)

PSL (Pattern Space Layout) files are are generated by the BLAST-Like Alignment Tool BLAT [Kent2002]. Like BED files (see section *Browser Extensible Data (BED)*), PSL files are typically used to store alignments of transcripts to genomes. This is an example of a short BLAT file (available as `dna_rna.psl` in the `Tests/Blat` subdirectory of the Biopython distribution), with the standard PSL header consisting of 5 lines:

```
psLayout version 3

match   mis-    rep.    N's Q gap   Q gap   T gap   T gap   strand  Q           Q      ␣
↪Q       Q   T           T       T       T   block   blockSizes  qStarts  tStarts
        match   match       count   bases   count   bases           name        size   ␣
↪start   end name        size    start   end count
-------------------------------------------------------------------------------------
↪-----------------------------------------------------------------------
165 0   39  0   0   0   2   5203    +   NR_111921.1 216 0   204 chr3    198295559  ␣
↪48663767    48669174    3   46,82,76,   0,46,128,   48663767,48665640,48669098,
175 0   6   0   0   0   2   1530    -   NR_046654.1 181 0   181 chr3    198295559  ␣
↪42530895    42532606    3   63,75,43,   0,63,138,   42530895,42532020,42532563,
162 2   39  0   1   2   3   5204    +   NR_111921.1_modified    220 3   208 chr3   ␣
↪198295559   48663767    48669174    5   28,17,76,6,76,  3,31,48,126,132,    48663767,
↪48663796,48665640,48665716,48669098,
172 1   6   0   1   3   3   1532    -   NR_046654.1_modified    190 3   185 chr3   ␣
↪198295559   42530895    42532606    5   27,36,17,56,43, 5,35,71,88,144, 42530895,
↪42530922,42532020,42532039,42532563,
```

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("dna_rna.psl", "psl")
>>> alignments.metadata
{'psLayout version': '3'}
```

Iterate over the alignments to get one `Alignment` object for each line:

```
>>> for alignment in alignments:
...     print(alignment.target.id, alignment.query.id)
...
chr3 NR_046654.1
```

(continues on next page)

(continued from previous page)

```
chr3 NR_046654.1_modified
chr3 NR_111921.1
chr3 NR_111921.1_modified
```

Let's look at the last alignment in more detail. The first four columns in the PSL file show the number of matches, the number of mismatches, the number of nucleotides aligned to repeat regions, and the number of nucleotides aligned to N (unknown) characters. These values are stored as attributes to the `Alignment` object:

```
>>> alignment.matches
162
>>> alignment.misMatches
2
>>> alignment.repMatches
39
>>> alignment.nCount
0
```

As the sequence data of the target and query are not stored explicitly in the PSL file, the sequence content of `alignment.target` and `alignment.query` is undefined. However, their sequence lengths are known:

```
>>> alignment.target
SeqRecord(seq=Seq(None, length=198295559), id='chr3', ...)
>>> alignment.query
SeqRecord(seq=Seq(None, length=220), id='NR_111921.1_modified', ...)
```

We can print the alignment in BED or PSL format:

```
>>> print(format(alignment, "bed"))
chr3    48663767    48669174    NR_111921.1_modified    0   +   48663767    48669174    ␣
↪0   5   28,17,76,6,76,  0,29,1873,1949,5331,

>>> print(format(alignment, "psl"))
162 2   39  0   1   2   3   5204    +   NR_111921.1_modified    220 3   208 chr3    ␣
↪198295559   48663767    48669174    5   28,17,76,6,76,  3,31,48,126,132,    48663767,
↪48663796,48665640,48665716,48669098,
```

Here, the number of matches, mismatches, repeat region matches, and matches to unknown nucleotides were taken from the corresponding attributes of the `Alignment` object. If these attributes are not available, for example if the alignment did not come from a PSL file, then these numbers are calculated using the sequence contents, if available. Repeat regions in the target sequence are indicated by masking the sequence as lower-case or upper-case characters, as defined by the following values for the `mask` keyword argument:

- `False` (default): Do not count matches to masked sequences separately;
- `"lower"`: Count and report matches to lower-case characters as matches to repeat regions;
- `"upper"`: Count and report matches to upper-case characters as matches to repeat regions;

The character used for unknown nucleotides is defined by the `wildcard` argument. For consistency with BLAT, the wildcard character is `"N"` by default. Use `wildcard=None` if you don't want to count matches to any unknown nucleotides separately.

```
>>> import numpy
>>> from Bio import Align
>>> query = "GGTGGGGG"
```

(continues on next page)

(continued from previous page)

```
>>> target = "AAAAAAAggggGGNGAAAAA"
>>> coordinates = numpy.array([[0, 7, 15, 20], [0, 0, 8, 8]])
>>> alignment = Align.Alignment([target, query], coordinates)
>>> print(alignment)
target            0 AAAAAAAggggGGNGAAAAA 20
                  0 -------....||.|----- 20
query             0 -------GGTGGGGG-----  8

>>> line = alignment.format("psl")
>>> print(line)
6   1   0   1   0   0   0   0   +   query   8   0   8   target   20   7   15   1   8,  ␣
↪0,   7,
>>> line = alignment.format("psl", mask="lower")
>>> print(line)
3   1   3   1   0   0   0   0   +   query   8   0   8   target   20   7   15   1   8,  ␣
↪0,   7,
>>> line = alignment.format("psl", mask="lower", wildcard=None)
>>> print(line)
3   2   3   0   0   0   0   0   +   query   8   0   8   target   20   7   15   1   8,  ␣
↪0,   7,
```

The same arguments can be used when writing alignments to an output file in PSL format using `Bio.Align.write`. This function has an additional keyword `header` (`True` by default) specifying if the PSL header should be written.

In addition to the `format` method, you can use Python's built-in `format` function:

```
>>> print(format(alignment, "psl"))
6   1   0   1   0   0   0   0   +   query   8   0   8   target   20   7   15   1   8,  ␣
↪0,   7,
```

allowing `Alignment` objects to be used in formatted (f-) strings in Python:

```
>>> line = f"The alignment in PSL format is '{alignment:psl}'."
>>> print(line)
The alignment in PSL format is '6   1   0   1   0   0   0   0   +   query   8   0   8  ␣
↪target   20   7   15   1   8,   0,   7,
'
```

Note that optional keyword arguments cannot be used with the `format` function or with formatted strings.

6.7.17 bigPsl

A bigPsl file is a bigBed file with a BED12+13 format consisting of the 12 predefined BED fields and 13 custom fields defined in the AutoSql file bigPsl.as provided by UCSC, creating an indexed binary version of a PSL file (see section *Pattern Space Layout (PSL)*). To create a bigPsl file, you can either use the `pslToBigPsl` and `bedToBigBed` programs from UCSC. or you can use Biopython by calling the `Bio.Align.write` function with `fmt="bigpsl"`. While the two methods should result in identical bigPsl files, the UCSC tools are much faster and may be more reliable, as it is the gold standard. As bigPsl files are bigBed files, they come with a built-in index, allowing you to quickly search a specific genomic region.

As an example, let's parse the bigBed file `dna_rna.psl.bb`, available in the `Tests/Blat` subdirectory in the Biopython distribution. This file is the bigPsl equivalent of the bigBed file `dna_rna.bb` (see section *bigBed*) and of the PSL file `dna_rna.psl` (see section *Pattern Space Layout (PSL)*).

```
>>> from Bio import Align
>>> alignments = Align.parse("dna_rna.psl.bb", "bigpsl")
>>> len(alignments)
4
>>> print(alignments.declaration)
table bigPsl
"bigPsl pairwise alignment"
(
   string          chrom;           "Reference sequence chromosome or scaffold"
   uint            chromStart;      "Start position in chromosome"
   uint            chromEnd;        "End position in chromosome"
   string          name;            "Name or ID of item, ideally both human readable and␣
↪unique"
   uint            score;           "Score (0-1000)"
   char[1]         strand;          "+ or - indicates whether the query aligns to the +␣
↪or - strand on the reference"
   uint            thickStart;      "Start of where display should be thick (start codon)
↪"
   uint            thickEnd;        "End of where display should be thick (stop codon)"
   uint            reserved;        "RGB value (use R,G,B string in input file)"
   int             blockCount;      "Number of blocks"
   int[blockCount] blockSizes;      "Comma separated list of block sizes"
   int[blockCount] chromStarts;     "Start positions relative to chromStart"
   uint            oChromStart;     "Start position in other chromosome"
   uint            oChromEnd;       "End position in other chromosome"
   char[1]         oStrand;         "+ or -, - means that psl was reversed into BED-
↪compatible coordinates"
   uint            oChromSize;      "Size of other chromosome."
   int[blockCount] oChromStarts;    "Start positions relative to oChromStart or from␣
↪oChromStart+oChromSize depending on strand"
   lstring         oSequence;       "Sequence on other chrom (or edit list, or empty)"
   string          oCDS;            "CDS in NCBI format"
   uint            chromSize;       "Size of target chromosome"
   uint            match;           "Number of bases matched."
   uint            misMatch;        "Number of bases that don't match"
   uint            repMatch;        "Number of bases that match but are part of repeats"
   uint            nCount;          "Number of 'N' bases"
   uint            seqType;         "0=empty, 1=nucleotide, 2=amino_acid"
)
```

The declaration contains the specification of the columns as defined by the `bigPsl.as` AutoSql file from UCSC. Target sequences (typically, the chromosomes against which the sequences were aligned) are stored in the `targets` attribute. In the bigBed format, only the identifier and the size of each target is stored. In this example, there is only a single chromosome:

```
>>> alignments.targets
[SeqRecord(seq=Seq(None, length=198295559), id='chr3', name='<unknown name>',␣
↪description='<unknown description>', dbxrefs=[])]
```

Iterating over the alignments gives one Alignment object for each line:

```
>>> for alignment in alignments:
...     print(alignment.target.id, alignment.query.id)
```

(continues on next page)

(continued from previous page)

```
...
chr3 NR_046654.1
chr3 NR_046654.1_modified
chr3 NR_111921.1
chr3 NR_111921.1_modified
```

Let's look at the individual alignments. The alignment information is stored in the same way as for the corresponding PSL file (see section *Pattern Space Layout (PSL)*):

```
>>> alignment.coordinates
array([[48663767, 48663795, 48663796, 48663813, 48665640, 48665716,
        48665716, 48665722, 48669098, 48669174],
       [       3,       31,       31,       48,       48,      124,
             126,      132,      132,      208]])
>>> alignment.thickStart
48663767
>>> alignment.thickEnd
48669174
>>> alignment.matches
162
>>> alignment.misMatches
2
>>> alignment.repMatches
39
>>> alignment.nCount
0
```

We can print the alignment in BED or PSL format:

```
>>> print(format(alignment, "bed"))
chr3    48663767    48669174    NR_111921.1_modified    1000    +    48663767    48669174␣
↪    0    5    28,17,76,6,76,  0,29,1873,1949,5331,

>>> print(format(alignment, "psl"))
162 2    39  0    1    2    3    5204    +    NR_111921.1_modified    220 3    208 chr3    ␣
↪198295559    48663767    48669174    5    28,17,76,6,76,  3,31,48,126,132,    48663767,
↪48663796,48665640,48665716,48669098,
```

As a bigPsl file is a special case of a bigBed file, you can use the `search` method on the alignments object to find alignments to specific genomic regions. For example, we can look for regions on chr3 between positions 48000000 and 49000000:

```
>>> selected_alignments = alignments.search("chr3", 48000000, 49000000)
>>> for alignment in selected_alignments:
...     print(alignment.query.id)
...
NR_111921.1
NR_111921.1_modified
```

The chromosome name may be `None` to include all chromosomes, and the start and end positions may be `None` to start searching from position 0 or to continue searching until the end of the chromosome, respectively.

To write a bigPsl file with Biopython, use `Bio.Align.write(alignments, "myfilename.bb", fmt="bigpsl")`, where `myfilename.bb` is the name of the output bigPsl file. Alternatively, you can use a (binary) stream for output.

Additional options are

- `compress`: If `True` (default), compress data using zlib; if `False`, do not compress data.
- `extraIndex`: List of strings with the names of extra columns to be indexed.
- `cds`: If `True`, look for a query feature of type CDS and write it in NCBI style in the PSL file (default: `False`).
- `fa`: If `True`, include the query sequence in the PSL file (default: `False`).
- `mask`: Specify if repeat regions in the target sequence are masked and should be reported in the `repMatches` field instead of in the `matches` field. Acceptable values are
  - `None`: no masking (default);
  - `"lower"`: masking by lower-case characters;
  - `"upper"`: masking by upper-case characters.
- `wildcard`: Report alignments to the wildcard character (representing unknown nucleotides) in the target or query sequence in the `nCount` field instead of in the `matches`, `misMatches`, or `repMatches` fields. Default value is `"N"`.

See section *Pattern Space Layout (PSL)* for an explanation on how the number of matches, mismatches, repeat region matches, and matches to unknown nucleotides are obtained.

Further optional arguments are `blockSize` (default value is 256), and `itemsPerSlot` (default value is 512). See the documentation of UCSC's `bedToBigBed` program for a description of these arguments. Searching a `bigPsl` file can be faster by using `compress=False` and `itemsPerSlot=1` when creating the bigPsl file.

6.7.18 Multiple Alignment Format (MAF)

MAF (Multiple Alignment Format) files store a series of multiple sequence alignments in a human-readable format. MAF files are typically used to store alignment s of genomes to each other. The file `ucsc_test.maf` in the `Tests/MAF` subdirectory of the Biopython distribution is an example of a simple MAF file:

```
track name=euArc visibility=pack mafDot=off frames="multiz28wayFrames" speciesOrder=
↪"hg16 panTro1 baboon mm4 rn3" description="A sample alignment"
##maf version=1 scoring=tba.v8
# tba.v8 (((human chimp) baboon) (mouse rat))
# multiz.v7
# maf_project.v5 _tba_right.maf3 mouse _tba_C
# single_cov2.v4 single_cov2 /dev/stdin

a score=23262.0
s hg16.chr7    27578828 38 + 158545518 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s panTro1.chr6 28741140 38 + 161576975 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s baboon         116834 38 +   4622798 AAA-GGGAATGTTAACCAAATGA---GTTGTCTCTTATGGTG
s mm4.chr6     53215344 38 + 151104725 -AATGGGAATGTTAAGCAAACGA---ATTGTCTCTCAGTGTG
s rn3.chr4     81344243 40 + 187371129 -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCAATGTG

a score=5062.0
s hg16.chr7    27699739 6 + 158545518 TAAAGA
s panTro1.chr6 28862317 6 + 161576975 TAAAGA
s baboon         241163 6 +   4622798 TAAAGA
s mm4.chr6     53303881 6 + 151104725 TAAAGA
s rn3.chr4     81444246 6 + 187371129 taagga
```

(continues on next page)

(continued from previous page)

```
a score=6636.0
s hg16.chr7    27707221 13 + 158545518 gcagctgaaaaca
s panTro1.chr6 28869787 13 + 161576975 gcagctgaaaaca
s baboon         249182 13 +   4622798 gcagctgaaaaca
s mm4.chr6     53310102 13 + 151104725 ACAGCTGAAAATA
```

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("ucsc_test.maf", "maf")
```

Information shown in the file header (the track line and subsequent lines starting with "#")) is stored in the `metadata` attribute of the `alignments` object:

```
>>> alignments.metadata
{'name': 'euArc',
 'visibility': 'pack',
 'mafDot': 'off',
 'frames': 'multiz28wayFrames',
 'speciesOrder': ['hg16', 'panTro1', 'baboon', 'mm4', 'rn3'],
 'description': 'A sample alignment',
 'MAF Version': '1',
 'Scoring': 'tba.v8',
 'Comments': ['tba.v8 (((human chimp) baboon) (mouse rat))',
              'multiz.v7',
              'maf_project.v5 _tba_right.maf3 mouse _tba_C',
              'single_cov2.v4 single_cov2 /dev/stdin']}
```

By iterating over the `alignments` we obtain one `Alignment` object for each alignment block in the MAF file:

```
>>> alignment = next(alignments)
>>> alignment.score
23262.0
>>> {seq.id: len(seq) for seq in alignment.sequences}
{'hg16.chr7': 158545518,
 'panTro1.chr6': 161576975,
 'baboon': 4622798,
 'mm4.chr6': 151104725,
 'rn3.chr4': 187371129}
>>> print(alignment.coordinates)
[[27578828 27578829 27578831 27578831 27578850 27578850 27578866]
 [28741140 28741141 28741143 28741143 28741162 28741162 28741178]
 [  116834   116835   116837   116837   116856   116856   116872]
 [53215344 53215344 53215346 53215347 53215366 53215366 53215382]
 [81344243 81344243 81344245 81344245 81344264 81344267 81344283]]
>>> print(alignment)
hg16.chr7  27578828 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG 27578866
panTro1.c  28741140 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG 28741178
baboon       116834 AAA-GGGAATGTTAACCAAATGA---GTTGTCTCTTATGGTG   116872
mm4.chr6   53215344 -AATGGGAATGTTAAGCAAACGA---ATTGTCTCTCAGTGTG 53215382
rn3.chr4   81344243 -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCAATGTG 81344283

>>> print(format(alignment, "phylip"))
```

(continues on next page)

(continued from previous page)

```
5 42
hg16.chr7 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
panTro1.chAAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
baboon    AAA-GGGAATGTTAACCAAATGA---GTTGTCTCTTATGGTG
mm4.chr6  -AATGGGAATGTTAAGCAAACGA---ATTGTCTCTCAGTGTG
rn3.chr4  -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCAATGTG
```

In addition to the "a" (alignment block) and "s" (sequence) lines, MAF files may contain "i" lines with information about the genome sequence before and after this block, "e" lines with information about empty parts of the alignment, and "q" lines showing the quality of each aligned base. This is an example of an alignment block including such lines:

```
a score=19159.000000
s mm9.chr10                          3014644 45 + 129993255 CCTGTACC---
↪CTTTGGTGAGAATTTTTGTTTCAGTGTTAAAAGTTTG
s hg18.chr6                         15870786 46 - 170899992 CCTATACCTTTCTTTTATGAGAA-
↪TTTTGTTTTAATCCTAAAC-TTTT
i hg18.chr6                         I 9085 C 0
s panTro2.chr6                      16389355 46 - 173908612 CCTATACCTTTCTTTTATGAGAA-
↪TTTTGTTTTAATCCTAAAC-TTTT
q panTro2.chr6                                              99999999999999999999-
↪9999999999999999999-9999
i panTro2.chr6                      I 9106 C 0
s calJac1.Contig6394                    6182 46 +    133105 CCTATACCTTTCTTTCATGAGAA-
↪TTTTGTTTGAATCCTAAAC-TTTT
i calJac1.Contig6394                N 0 C 0
s loxAfr1.scaffold_75566                1167 34 -     10574 ------------TTTGGTTAGAA-
↪TTATGCTTTAATTCAAAAC-TTCC
q loxAfr1.scaffold_75566                                    ------------99999699899-
↪9999999999999869998-9997
i loxAfr1.scaffold_75566            N 0 C 0
e tupBel1.scaffold_114895.1-498454   167376 4145 -    498454 I
e echTel1.scaffold_288249             87661 7564 +    100002 I
e otoGar1.scaffold_334.1-359464      181217 2931 -    359464 I
e ponAbe2.chr6                      16161448 8044 - 174210431 I
```

This is the 10th alignment block in the file `ucsc_mm9_chr10.maf` (available in the `Tests/MAF` subdirectory of the Biopython distribution):

```
>>> from Bio import Align
>>> alignments = Align.parse("ucsc_mm9_chr10.maf", "maf")
>>> for i in range(10):
...     alignment = next(alignments)
...
>>> alignment.score
19159.0
>>> print(alignment)
mm9.chr10   3014644 CCTGTACC---CTTTGGTGAGAATTTTTGTTTCAGTGTTAAAAGTTTG   3014689
hg18.chr6 155029206 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAAC-TTTT 155029160
panTro2.c 157519257 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAAC-TTTT 157519211
calJac1.C      6182 CCTATACCTTTCTTTCATGAGAA-TTTTGTTTGAATCCTAAAC-TTTT      6228
loxAfr1.s      9407 ------------TTTGGTTAGAA-TTATGCTTTAATTCAAAAC-TTCC      9373
```

The "i" lines show the relationship between the sequence in the current alignment block to the ones in the preceding and

subsequent alignment block. This information is stored in the `annotations` attribute of the corresponding sequence:

```
>>> alignment.sequences[0].annotations
{}
>>> alignment.sequences[1].annotations
{'leftStatus': 'I', 'leftCount': 9085, 'rightStatus': 'C', 'rightCount': 0}
```

showing that there are 9085 bases inserted ("I") between this block and the preceding one, while the block is contiguous ("C") with the subsequent one. See the UCSC documentation for the full description of these fields and status characters.

The "q" lines show the sequence quality, which is stored under the "quality" dictionary key of the`annotations` attribute of the corresponding sequence:

```
>>> alignment.sequences[2].annotations["quality"]
'99999999999999999999999999999999999999999999999'
>>> alignment.sequences[4].annotations["quality"]
'99999698989999999999999998699989997'
```

The "e" lines show information about species with a contiguous sequence before and after this alignment bloack, but with no aligning nucleotides in this alignment block. This is stored under the "empty" key of the `alignment.annotations` dictionary:

```
>>> alignment.annotations["empty"]
[(SeqRecord(seq=Seq(None, length=498454), id='tupBel1.scaffold_114895.1-498454', name='',
↪ description='', dbxrefs=[]), (331078, 326933), 'I'),
 (SeqRecord(seq=Seq(None, length=100002), id='echTel1.scaffold_288249', name='',␣
↪description='', dbxrefs=[]), (87661, 95225), 'I'),
 (SeqRecord(seq=Seq(None, length=359464), id='otoGar1.scaffold_334.1-359464', name='',␣
↪description='', dbxrefs=[]), (178247, 175316), 'I'),
 (SeqRecord(seq=Seq(None, length=174210431), id='ponAbe2.chr6', name='', description='',␣
↪dbxrefs=[]), (158048983, 158040939), 'I')]
```

This shows for example that there were non-aligning bases inserted ("I") from position 158040939 to 158048983 on the opposite strand of the `ponAbe2.chr6` genomic sequence. Again, see the UCSC documentation for the full definition of "e" lines.

To print an alignment in MAF format, you can use Python's built-in `format` function:

```
>>> print(format(alignment, "MAF"))
a score=19159.000000
s mm9.chr10                        3014644   45 + 129993255 CCTGTACC---
↪CTTTGGTGAGAATTTTTGTTTCAGTGTTAAAAGTTTG
s hg18.chr6                       15870786   46 - 170899992 CCTATACCTTTCTTTTATGAGAA-
↪TTTTGTTTTAATCCTAAAC-TTTT
i hg18.chr6                       I 9085 C 0
s panTro2.chr6                    16389355   46 - 173908612 CCTATACCTTTCTTTTATGAGAA-
↪TTTTGTTTTAATCCTAAAC-TTTT
q panTro2.chr6                                              99999999999999999999-
↪9999999999999999999-9999
i panTro2.chr6                    I 9106 C 0
s calJac1.Contig6394                 6182   46 +    133105 CCTATACCTTTCTTTCATGAGAA-
↪TTTTGTTTGAATCCTAAAC-TTTT
i calJac1.Contig6394              N 0 C 0
s loxAfr1.scaffold_75566             1167   34 -     10574 ------------TTTGGTTAGAA-
↪TTATGCTTTAATTCAAAAC-TTCC
```

(continues on next page)

(continued from previous page)

```
q loxAfr1.scaffold_75566                                    ------------99999699899-
↪9999999999999869998-9997
i loxAfr1.scaffold_75566           N 0 C 0
e tupBel1.scaffold_114895.1-498454   167376 4145 -    498454 I
e echTel1.scaffold_288249             87661 7564 +    100002 I
e otoGar1.scaffold_334.1-359464      181217 2931 -    359464 I
e ponAbe2.chr6                     16161448 8044 - 174210431 I
```

To write a complete MAF file, use `Bio.Align.write(alignments, "myfilename.maf", fmt="maf")`, where `myfilename.maf` is the name of the output MAF file. Alternatively, you can use a (text) stream for output. File header information will be taken from the `metadata` attribute of the `alignments` object. If you are creating the alignments from scratch, you can use the `Alignments` (plural) class to create a list-like `alignments` object (see Section *The Alignments class*) and give it a `metadata` attribute.

6.7.19 bigMaf

A bigMaf file is a bigBed file with a BED3+1 format consisting of the 3 required BED fields plus a custom field that stores a MAF alignment block as a string, creating an indexed binary version of a MAF file (see section *Multiple Alignment Format (MAF)*). The associated AutoSql file bigMaf.as is provided by UCSC. To create a bigMaf file, you can either use the `mafToBigMaf` and `bedToBigBed` programs from UCSC. or you can use Biopython by calling the Bio.Align.write function with `fmt="bigmaf"`. While the two methods should result in identical bigMaf files, the UCSC tools are much faster and may be more reliable, as it is the gold standard. As bigMaf files are bigBed files, they come with a built-in index, allowing you to quickly search a specific region of the reference genome.

The file `ucsc_test.bb` in the `Tests/MAF` subdirectory of the Biopython distribution is an example of a bigMaf file. This file is equivalent to the MAF file `ucsc_test.maf` (see section *Multiple Alignment Format (MAF)*). To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("ucsc_test.bb", "bigmaf")
>>> len(alignments)
3
>>> print(alignments.declaration)
table bedMaf
"Bed3 with MAF block"
(
   string  chrom;         "Reference sequence chromosome or scaffold"
   uint    chromStart;    "Start position in chromosome"
   uint    chromEnd;      "End position in chromosome"
   lstring mafBlock;      "MAF block"
)
```

The declaration contains the specification of the columns as defined by the bigMaf.as AutoSql file from UCSC.

The bigMaf file does not store the header information found in the MAF file, but it does define a reference genome. The corresponding `SeqRecord` is stored in the `targets` attribute of the `alignments` object:

```
>>> alignments.reference
'hg16'
>>> alignments.targets
[SeqRecord(seq=Seq(None, length=158545518), id='hg16.chr7', ...)]
```

By iterating over the `alignments` we obtain one `Alignment` object for each alignment block in the bigMaf file:

```
>>> alignment = next(alignments)
>>> alignment.score
23262.0
>>> {seq.id: len(seq) for seq in alignment.sequences}
{'hg16.chr7': 158545518,
 'panTro1.chr6': 161576975,
 'baboon': 4622798,
 'mm4.chr6': 151104725,
 'rn3.chr4': 187371129}
>>> print(alignment.coordinates)
[[27578828 27578829 27578831 27578831 27578850 27578850 27578866]
 [28741140 28741141 28741143 28741143 28741162 28741162 28741178]
 [  116834   116835   116837   116837   116856   116856   116872]
 [53215344 53215344 53215346 53215347 53215366 53215366 53215382]
 [81344243 81344243 81344245 81344245 81344264 81344267 81344283]]
>>> print(alignment)
hg16.chr7  27578828 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG 27578866
panTro1.c  28741140 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG 28741178
baboon       116834 AAA-GGGAATGTTAACCAAATGA---GTTGTCTCTTATGGTG   116872
mm4.chr6   53215344 -AATGGGAATGTTAAGCAAACGA---ATTGTCTCTCAGTGTG 53215382
rn3.chr4   81344243 -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCAATGTG 81344283

>>> print(format(alignment, "phylip"))
5 42
hg16.chr7 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
panTro1.chAAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
baboon    AAA-GGGAATGTTAACCAAATGA---GTTGTCTCTTATGGTG
mm4.chr6  -AATGGGAATGTTAAGCAAACGA---ATTGTCTCTCAGTGTG
rn3.chr4  -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCAATGTG
```

Information in the "i", "e", and "q" lines is stored in the same way as in the corresponding MAF file (see section *Multiple Alignment Format (MAF)*):

```
>>> from Bio import Align
>>> alignments = Align.parse("ucsc_mm9_chr10.bb", "bigmaf")
>>> for i in range(10):
...     alignment = next(alignments)
...
>>> alignment.score
19159.0
>>> print(alignment)
mm9.chr10   3014644 CCTGTACC---CTTTGGTGAGAATTTTTGTTTCAGTGTTAAAAGTTTG   3014689
hg18.chr6 155029206 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAAC-TTTT 155029160
panTro2.c 157519257 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAAC-TTTT 157519211
calJac1.C      6182 CCTATACCTTTCTTTCATGAGAA-TTTTGTTTGAATCCTAAAC-TTTT      6228
loxAfr1.s      9407 ------------TTTGGTTAGAA-TTATGCTTTAATTCAAAAC-TTCC      9373

>>> print(format(alignment, "MAF"))
a score=19159.000000
s mm9.chr10                        3014644   45 + 129993255 CCTGTACC---
↪CTTTGGTGAGAATTTTTGTTTCAGTGTTAAAAGTTTG
s hg18.chr6                       15870786   46 - 170899992 CCTATACCTTTCTTTTATGAGAA-
```

(continues on next page)

(continued from previous page)

```
↪TTTTGTTTTAATCCTAAAC-TTTT
i hg18.chr6                          I 9085 C 0
s panTro2.chr6                       16389355   46 - 173908612 CCTATACCTTTCTTTTATGAGAA-
↪TTTTGTTTTAATCCTAAAC-TTTT
q panTro2.chr6                                                 99999999999999999999-
↪9999999999999999999-9999
i panTro2.chr6                       I 9106 C 0
s calJac1.Contig6394                     6182   46 +    133105 CCTATACCTTTCTTTCATGAGAA-
↪TTTTGTTTGAATCCTAAAC-TTTT
i calJac1.Contig6394                 N 0 C 0
s loxAfr1.scaffold_75566                 1167   34 -     10574 ------------TTTGGTTAGAA-
↪TTATGCTTTAATTCAAAAC-TTCC
q loxAfr1.scaffold_75566                                       ------------99999699899-
↪9999999999999869998-9997
i loxAfr1.scaffold_75566             N 0 C 0
e tupBel1.scaffold_114895.1-498454   167376 4145 -    498454 I
e echTel1.scaffold_288249             87661 7564 +    100002 I
e otoGar1.scaffold_334.1-359464      181217 2931 -    359464 I
e ponAbe2.chr6                       16161448 8044 - 174210431 I


>>> alignment.sequences[1].annotations
{'leftStatus': 'I', 'leftCount': 9085, 'rightStatus': 'C', 'rightCount': 0}
>>> alignment.sequences[2].annotations["quality"]
'99999999999999999999999999999999999999999999999'
>>> alignment.sequences[4].annotations["quality"]
'99999698999999999999999698999989997'
>>> alignment.annotations["empty"]
[(SeqRecord(seq=Seq(None, length=498454), id='tupBel1.scaffold_114895.1-498454', name='',
↪ description='', dbxrefs=[]), (331078, 326933), 'I'),
 (SeqRecord(seq=Seq(None, length=100002), id='echTel1.scaffold_288249', name='',␣
↪description='', dbxrefs=[]), (87661, 95225), 'I'),
 (SeqRecord(seq=Seq(None, length=359464), id='otoGar1.scaffold_334.1-359464', name='',␣
↪description='', dbxrefs=[]), (178247, 175316), 'I'),
 (SeqRecord(seq=Seq(None, length=174210431), id='ponAbe2.chr6', name='', description='',␣
↪dbxrefs=[]), (158048983, 158040939), 'I')]
```

To write a complete bigMaf file, use `Bio.Align.write(alignments, "myfilename.bb", fmt="bigMaf")`, where `myfilename.bb` is the name of the output bigMaf file. Alternatively, you can use a (binary) stream for output. If you are creating the alignments from scratch, you can use the `Alignments` (plural) class to create a list-like `alignments` object (see Section *The Alignments class*) and give it a `targets` attribute. The latter must be a list of `SeqRecord` objects for the chromosomes for the reference species in the order in which they appear in the alignments. Alternatively, you can use the `targets` keyword argument when calling `Bio.Align.write`. The `id` of each `SeqRecord` must be of the form `reference.chromosome`, where `reference` refers to the reference species. `Bio.Align.write` has the additional keyword argument `compress` (`True` by default) specifying whether the data should be compressed using zlib. Further optional arguments are `blockSize` (default value is 256), and `itemsPerSlot` (default value is 512). See the documentation of UCSC's `bedToBigBed` program for a description of these arguments.

As a bigMaf file is a special case of a bigBed file, you can use the `search` method on the `alignments` object to find alignments to specific regions of the reference species. For example, we can look for regions on chr10 between positions 3018000 and 3019000 on chromosome 10:

```
>>> selected_alignments = alignments.search("mm9.chr10", 3018000, 3019000)
>>> for alignment in selected_alignments:
...     start, end = alignment.coordinates[0, 0], alignment.coordinates[0, -1]
...     print(start, end)
...
3017743 3018161
3018161 3018230
3018230 3018359
3018359 3018482
3018482 3018644
3018644 3018822
3018822 3018932
3018932 3019271
```

The chromosome name may be `None` to include all chromosomes, and the start and end positions may be `None` to start searching from position 0 or to continue searching until the end of the chromosome, respectively. Note that we can search on genomic position for the reference species only.

Searching a `bigMaf` file can be faster by using `compress=False` and `itemsPerSlot=1` when creating the bigMaf file.

6.7.20 UCSC chain file format

Chain files describe a pairwise alignment between two nucleotide sequences, allowing gaps in both sequences. Only the length of each aligned subsequences and the gap lengths are stored in a chain file; the sequences themselves are not stored. Chain files are typically used to store alignments between two genome assembly versions, allowing alignments to one genome assembly version to be lifted over to the other genome assembly. This is an example of a chain file (available as `psl_34_001.chain` in the `Tests/Blat` subdirectory of the Biopython distribution):

```
chain 16 chr4 191154276 + 61646095 61646111 hg18_dna 33 + 11 27 1
16
chain 33 chr1 249250621 + 10271783 10271816 hg18_dna 33 + 0 33 2
33
chain 17 chr2 243199373 + 53575980 53575997 hg18_dna 33 - 8 25 3
17
chain 35 chr9 141213431 + 85737865 85737906 hg19_dna 50 + 9 50 4
41
chain 41 chr8 146364022 + 95160479 95160520 hg19_dna 50 + 8 49 5
41
chain 30 chr22 51304566 + 42144400 42144436 hg19_dna 50 + 11 47 6
36
chain 41 chr2 243199373 + 183925984 183926028 hg19_dna 50 + 1 49 7
6       0       4
38
chain 31 chr19 59128983 + 35483340 35483510 hg19_dna 50 + 10 46 8
25      134     0
11
chain 39 chr18 78077248 + 23891310 23891349 hg19_dna 50 + 10 49 9
39
...
```

This file was generated by running UCSC's `pslToChain` program on the PSL file `psl_34_001.psl`. According to the chain file format specification, there should be a blank line after each chain block, but some tools (including

pslToChain) apparently do not follow this rule.

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("psl_34_001.chain", "chain")
```

Iterate over alignments to get one Alignment object for each chain:

```
>>> for alignment in alignments:
...     print(alignment.target.id, alignment.query.id)
...
chr4 hg18_dna
chr1 hg18_dna
chr2 hg18_dna
chr9 hg19_dna
chr8 hg19_dna
chr22 hg19_dna
chr2 hg19_dna
...
chr1 hg19_dna
```

Iterate from the start until we reach the seventh alignment:

```
>>> alignments = iter(alignments)
>>> for i in range(7):
...     alignment = next(alignments)
...
```

Check the alignment score and chain ID (the first and last number, respectively, in the header line of each chain block) to confirm that we got the seventh alignment:

```
>>> alignment.score
41.0
>>> alignment.annotations["id"]
'7'
```

We can print the alignment in the chain file format. The alignment coordinates are consistent with the information in the chain block, with an aligned section of 6 nucleotides, a gap of 4 nucleotides, and an aligned section of 38 nucleotides:

```
>>> print(format(alignment, "chain"))
chain 41 chr2 243199373 + 183925984 183926028 hg19_dna 50 + 1 49 7
6   0   4
38


>>> alignment.coordinates
array([[183925984, 183925990, 183925990, 183926028],
       [        1,         7,        11,        49]])
>>> print(alignment)
chr2      183925984 ??????----????????????????????????????????????? 183926028
                  0 ||||||----|||||||||||||||||||||||||||||||||||||||        48
hg19_dna          1 ?????????????????????????????????????????????????        49
```

We can also print the alignment in a few other alignment fite formats:

```
>>> print(format(alignment, "BED"))
chr2    183925984   183926028   hg19_dna    41  +   183925984   183926028   0   2   6,38,
↪   0,6,

>>> print(format(alignment, "PSL"))
44  0   0   0   1   4   0   0   +   hg19_dna    50  1   49  chr2    243199373  ␣
↪183925984   183926028   2   6,38,   1,11,   183925984,183925990,

>>> print(format(alignment, "exonerate"))
vulgar: hg19_dna 1 49 + chr2 183925984 183926028 + 41 M 6 6 G 4 0 M 38 38

>>> print(alignment.format("exonerate", "cigar"))
cigar: hg19_dna 1 49 + chr2 183925984 183926028 + 41 M 6 I 4 M 38

>>> print(format(alignment, "sam"))
hg19_dna    0   chr2    183925985   255 1S6M4I38M1S *   0   0   *   *   AS:i:41 id:A:7
```

CHAPTER

SEVEN

PAIRWISE SEQUENCE ALIGNMENT

Pairwise sequence alignment is the process of aligning two sequences to each other by optimizing the similarity score between them. The `Bio.Align` module contains the `PairwiseAligner` class for global and local alignments using the Needleman-Wunsch, Smith-Waterman, Gotoh (three-state), and Waterman-Smith-Beyer global and local pairwise alignment algorithms, with numerous options to change the alignment parameters. We refer to Durbin *et al.* [Durbin1998] for in-depth information on sequence alignment algorithms.

7.1 Basic usage

To generate pairwise alignments, first create a `PairwiseAligner` object:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
```

The `PairwiseAligner` object `aligner` (see Section *The pairwise aligner object*) stores the alignment parameters to be used for the pairwise alignments. These attributes can be set in the constructor of the object:

```
>>> aligner = Align.PairwiseAligner(match_score=1.0)
```

or after the object is made:

```
>>> aligner.match_score = 1.0
```

Use the `aligner.score` method to calculate the alignment score between two sequences:

```
>>> target = "GAACT"
>>> query = "GAT"
>>> score = aligner.score(target, query)
>>> score
3.0
```

The `aligner.align` method returns `Alignment` objects, each representing one alignment between the two sequences:

```
>>> alignments = aligner.align(target, query)
>>> alignment = alignments[0]
>>> alignment
<Alignment object (2 rows x 5 columns) at ...>
```

Iterate over the `Alignment` objects and print them to see the alignments:

```
>>> for alignment in alignments:
...     print(alignment)
...
target            0 GAACT 5
                  0 ||--| 5
query             0 GA--T 3

target            0 GAACT 5
                  0 |-|-| 5
query             0 G-A-T 3
```

Each alignment stores the alignment score:

```
>>> alignment.score
3.0
```

as well as pointers to the sequences that were aligned:

```
>>> alignment.target
'GAACT'
>>> alignment.query
'GAT'
```

Internally, the alignment is stored in terms of the sequence coordinates:

```
>>> alignment = alignments[0]
>>> alignment.coordinates
array([[0, 2, 4, 5],
       [0, 2, 2, 3]])
```

Here, the two rows refer to the target and query sequence. These coordinates show that the alignment consists of the following three blocks:

- `target[0:2]` aligned to `query[0:2]`;
- `target[2:4]` aligned to a gap, since `query[2:2]` is an empty string;
- `target[4:5]` aligned to `query[2:3]`.

The number of aligned sequences is always 2 for a pairwise alignment:

```
>>> len(alignment)
2
```

The alignment length is defined as the number of columns in the alignment as printed. This is equal to the sum of the number of matches, number of mismatches, and the total length of gaps in the target and query:

```
>>> alignment.length
5
```

The `aligned` property, which returns the start and end indices of aligned subsequences, returns two tuples of length 2 for the first alignment:

```
>>> alignment.aligned
array([[[0, 2],
        [4, 5]],
```

(continues on next page)

(continued from previous page)

```
       [[0, 2],
        [2, 3]]])
```

while for the alternative alignment, two tuples of length 3 are returned:

```
>>> alignment = alignments[1]
>>> print(alignment)
target            0 GAACT 5
                  0 |-|-| 5
query             0 G-A-T 3

>>> alignment.aligned
array([[[0, 1],
        [2, 3],
        [4, 5]],

       [[0, 1],
        [1, 2],
        [2, 3]]])
```

Note that different alignments may have the same subsequences aligned to each other. In particular, this may occur if alignments differ from each other in terms of their gap placement only:

```
>>> aligner.mode = "global"
>>> aligner.mismatch_score = -10
>>> alignments = aligner.align("AAACAAA", "AAAGAAA")
>>> len(alignments)
2
>>> print(alignments[0])
target            0 AAAC-AAA 7
                  0 |||--||| 8
query             0 AAA-GAAA 7

>>> alignments[0].aligned
array([[[0, 3],
        [4, 7]],

       [[0, 3],
        [4, 7]]])
>>> print(alignments[1])
target            0 AAA-CAAA 7
                  0 |||--||| 8
query             0 AAAG-AAA 7

>>> alignments[1].aligned
array([[[0, 3],
        [4, 7]],

       [[0, 3],
        [4, 7]]])
```

The `map` method can be applied on a pairwise alignment `alignment1` to find the pairwise alignment of the query

of alignment2 to the target of alignment1, where the target of alignment2 and the query of alignment1 are identical. A typical example is where alignment1 is the pairwise alignment between a chromosome and a transcript, alignment2 is the pairwise alignment between the transcript and a sequence (e.g., an RNA-seq read), and we want to find the alignment of the sequence to the chromosome:

```
>>> aligner.mode = "local"
>>> aligner.open_gap_score = -1
>>> aligner.extend_gap_score = 0
>>> chromosome = "AAAAAAAACCCCCCCAAAAAAAAAAAGGGGGGAAAAAAAA"
>>> transcript = "CCCCCCCGGGGGG"
>>> alignments1 = aligner.align(chromosome, transcript)
>>> len(alignments1)
1
>>> alignment1 = alignments1[0]
>>> print(alignment1)
target            8 CCCCCCCAAAAAAAAAAAGGGGGG 32
                  0 |||||||-----------|||||| 24
query             0 CCCCCCC-----------GGGGGG 13

>>> sequence = "CCCCGGGG"
>>> alignments2 = aligner.align(transcript, sequence)
>>> len(alignments2)
1
>>> alignment2 = alignments2[0]
>>> print(alignment2)
target            3 CCCCGGGG 11
                  0 ||||||||  8
query             0 CCCCGGGG  8

>>> mapped_alignment = alignment1.map(alignment2)
>>> print(mapped_alignment)
target           11 CCCCAAAAAAAAAAGGGG 30
                  0 ||||-----------|||| 19
query             0 CCCC-----------GGGG  8

>>> format(mapped_alignment, "psl")
'8\t0\t0\t0\t0\t0\t1\t11\t+\tquery\t8\t0\t8\ttarget\t40\t11\t30\t2\t4,4,\t0,4,\t11,26,\n'
```

Mapping the alignment does not depend on the sequence contents. If we delete the sequence contents, the same alignment is found in PSL format (though we obviously lose the ability to print the sequence alignment):

```
>>> from Bio.Seq import Seq
>>> alignment1.target = Seq(None, len(alignment1.target))
>>> alignment1.query = Seq(None, len(alignment1.query))
>>> alignment2.target = Seq(None, len(alignment2.target))
>>> alignment2.query = Seq(None, len(alignment2.query))
>>> mapped_alignment = alignment1.map(alignment2)
>>> format(mapped_alignment, "psl")
'8\t0\t0\t0\t0\t0\t1\t11\t+\tquery\t8\t0\t8\ttarget\t40\t11\t30\t2\t4,4,\t0,4,\t11,26,\n'
```

By default, a global pairwise alignment is performed, which finds the optimal alignment over the whole length of target and query. Instead, a local alignment will find the subsequence of target and query with the highest alignment score. Local alignments can be generated by setting aligner.mode to "local":

```
>>> aligner.mode = "local"
>>> target = "AGAACTC"
>>> query = "GAACT"
>>> score = aligner.score(target, query)
>>> score
5.0
>>> alignments = aligner.align(target, query)
>>> for alignment in alignments:
...     print(alignment)
...
target            1 GAACT 6
                  0 ||||| 5
query             0 GAACT 5
```

Note that there is some ambiguity in the definition of the best local alignments if segments with a score 0 can be added to the alignment. We follow the suggestion by Waterman & Eggert [Waterman1987] and disallow such extensions.

7.2 The pairwise aligner object

The `PairwiseAligner` object stores all alignment parameters to be used for the pairwise alignments. To see an overview of the values for all parameters, use

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner(match_score=1.0, mode="local")
>>> print(aligner)
Pairwise sequence aligner with parameters
  wildcard: None
  match_score: 1.000000
  mismatch_score: 0.000000
  target_internal_open_gap_score: 0.000000
  target_internal_extend_gap_score: 0.000000
  target_left_open_gap_score: 0.000000
  target_left_extend_gap_score: 0.000000
  target_right_open_gap_score: 0.000000
  target_right_extend_gap_score: 0.000000
  query_internal_open_gap_score: 0.000000
  query_internal_extend_gap_score: 0.000000
  query_left_open_gap_score: 0.000000
  query_left_extend_gap_score: 0.000000
  query_right_open_gap_score: 0.000000
  query_right_extend_gap_score: 0.000000
  mode: local
```

See Sections *Substitution scores*, *Affine gap scores*, and *General gap scores* below for the definition of these parameters. The attribute `mode` (described above in Section *Basic usage*) can be set equal to `"global"` or `"local"` to specify global or local pairwise alignment, respectively.

Depending on the gap scoring parameters (see Sections *Affine gap scores* and *General gap scores*) and mode, a `PairwiseAligner` object automatically chooses the appropriate algorithm to use for pairwise sequence alignment. To verify the selected algorithm, use

```
>>> aligner.algorithm
'Smith-Waterman'
```

This attribute is read-only.

A `PairwiseAligner` object also stores the precision ϵ to be used during alignment. The value of ϵ is stored in the attribute `aligner.epsilon`, and by default is equal to 10^{-6}:

```
>>> aligner.epsilon
1e-06
```

Two scores will be considered equal to each other for the purpose of the alignment if the absolute difference between them is less than ϵ.

7.3 Substitution scores

Substitution scores define the value to be added to the total score when two letters (nucleotides or amino acids) are aligned to each other. The substitution scores to be used by the `PairwiseAligner` can be specified in two ways:

- By specifying a match score for identical letters, and a mismatch scores for mismatched letters. Nucleotide sequence alignments are typically based on match and mismatch scores. For example, by default BLAST [Altschul1990] uses a match score of $+1$ and a mismatch score of -2 for nucleotide alignments by `megablast`, with a gap penalty of 2.5 (see section *Affine gap scores* for more information on gap scores). Match and mismatch scores can be specified by setting the `match` and `mismatch` attributes of the `PairwiseAligner` object:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> aligner.match_score
1.0
>>> aligner.mismatch_score
0.0
>>> score = aligner.score("ACGT", "ACAT")
>>> print(score)
3.0
>>> aligner.match_score = 1.0
>>> aligner.mismatch_score = -2.0
>>> aligner.gap_score = -2.5
>>> score = aligner.score("ACGT", "ACAT")
>>> print(score)
1.0
```

  When using match and mismatch scores, you can specify a wildcard character (`None` by default) for unknown letters. These will get a zero score in alignments, irrespective of the value of the match or mismatch score:

```
>>> aligner.wildcard = "?"
>>> score = aligner.score("ACGT", "AC?T")
>>> print(score)
3.0
```

- Alternatively, you can use the `substitution_matrix` attribute of the `PairwiseAligner` object to specify a substitution matrix. This allows you to apply different scores for different pairs of matched and mismatched letters. This is typically used for amino acid sequence alignments. For example, by default BLAST [Altschul1990]

uses the BLOSUM62 substitution matrix for protein alignments by `blastp`. This substitution matrix is available from Biopython:

```
>>> from Bio.Align import substitution_matrices
>>> substitution_matrices.load()
['BENNER22', 'BENNER6', 'BENNER74', 'BLASTN', 'BLASTP', 'BLOSUM45', 'BLOSUM50',
↪'BLOSUM62', ..., 'TRANS']
>>> matrix = substitution_matrices.load("BLOSUM62")
>>> print(matrix)
#  Matrix made by matblas from blosum62.iij
...
     A    R    N    D    C    Q ...
A  4.0 -1.0 -2.0 -2.0  0.0 -1.0 ...
R -1.0  5.0  0.0 -2.0 -3.0  1.0 ...
N -2.0  0.0  6.0  1.0 -3.0  0.0 ...
D -2.0 -2.0  1.0  6.0 -3.0  0.0 ...
C  0.0 -3.0 -3.0 -3.0  9.0 -3.0 ...
Q -1.0  1.0  0.0  0.0 -3.0  5.0 ...
...
>>> aligner.substitution_matrix = matrix
>>> score = aligner.score("ACDQ", "ACDQ")
>>> score
24.0
>>> score = aligner.score("ACDQ", "ACNQ")
>>> score
19.0
```

When using a substitution matrix, `X` is *not* interpreted as an unknown character. Instead, the score provided by the substitution matrix will be used:

```
>>> matrix["D", "X"]
-1.0
>>> score = aligner.score("ACDQ", "ACXQ")
>>> score
17.0
```

By default, `aligner.substitution_matrix` is `None`. The attributes `aligner.match_score` and `aligner.mismatch_score` are ignored if `aligner.substitution_matrix` is not `None`. Setting `aligner.match_score` or `aligner.mismatch_score` to valid values will reset `aligner.substitution_matrix` to `None`.

7.4 Affine gap scores

Affine gap scores are defined by a score to open a gap, and a score to extend an existing gap:

gap score = open gap score + $(n - 1) \times$ extend gap score,

where n is the length of the gap. Biopython's pairwise sequence aligner allows fine-grained control over the gap scoring scheme by specifying the following twelve attributes of a `PairwiseAligner` object:

Opening scores	Extending scores
query_left_open_gap_score	query_left_extend_gap_score
query_internal_open_gap_score	query_internal_extend_gap_score
query_right_open_gap_score	query_right_extend_gap_score
target_left_open_gap_score	target_left_extend_gap_score
target_internal_open_gap_score	target_internal_extend_gap_score
target_right_open_gap_score	target_right_extend_gap_score

These attributes allow for different gap scores for internal gaps and on either end of the sequence, as shown in this example:

target	query	score
A	-	query left open gap score
C	-	query left extend gap score
C	-	query left extend gap score
G	G	match score
G	T	mismatch score
G	-	query internal open gap score
A	-	query internal extend gap score
A	-	query internal extend gap score
T	T	match score
A	A	match score
G	-	query internal open gap score
C	C	match score
-	C	target internal open gap score
-	C	target internal extend gap score
C	C	match score
T	G	mismatch score
C	C	match score
-	C	target internal open gap score
A	A	match score
-	T	target right open gap score
-	A	target right extend gap score
-	A	target right extend gap score

For convenience, `PairwiseAligner` objects have additional attributes that refer to a number of these values collectively, as shown (hierarchically) in Table *Meta-attributes of the pairwise aligner objects.*.

Table 1: Meta-attributes of the pairwise aligner objects.

Meta-attribute	Attributes it maps to
`gap_score`	`target_gap_score`, `query_gap_score`
`open_gap_score`	`target_open_gap_score`, `query_open_gap_score`
`extend_gap_score`	`target_extend_gap_score`, `query_extend_gap_score`

cc

Table 1 – continued from previous page

Meta-attribute	Attributes it maps to
internal_gap_score	target_internal_gap_score, query_internal_gap_score
internal_open_gap_score	target_internal_open_gap_score, query_internal_open_gap_score
internal_extend_gap_score	target_internal_extend_gap_score, query_internal_extend_gap_score
end_gap_score	target_end_gap_score, query_end_gap_score
end_open_gap_score	target_end_open_gap_score, query_end_open_gap_score
end_extend_gap_score	target_end_extend_gap_score, query_end_extend_gap_score
left_gap_score	target_left_gap_score, query_left_gap_score
right_gap_score	target_right_gap_score, query_right_gap_score
left_open_gap_score	target_left_open_gap_score, query_left_open_gap_score
left_extend_gap_score	target_left_extend_gap_score, query_left_extend_gap_score
right_open_gap_score	target_right_open_gap_score, query_right_open_gap_score
right_extend_gap_score	target_right_extend_gap_score, query_right_extend_gap_score
target_open_gap_score	target_internal_open_gap_score, target_left_open_gap_score, target_right_op
target_extend_gap_score	target_internal_extend_gap_score, target_left_extend_gap_score, target_righ
target_gap_score	target_open_gap_score, target_extend_gap_score
query_open_gap_score	query_internal_open_gap_score, query_left_open_gap_score, query_right_open_
query_extend_gap_score	query_internal_extend_gap_score, query_left_extend_gap_score, query_right_e
query_gap_score	query_open_gap_score, query_extend_gap_score
target_internal_gap_score	target_internal_open_gap_score, target_internal_extend_gap_score
target_end_gap_score	target_end_open_gap_score, target_end_extend_gap_score
target_end_open_gap_score	target_left_open_gap_score, target_right_open_gap_score
target_end_extend_gap_score	target_left_extend_gap_score, target_right_extend_gap_score
target_left_gap_score	target_left_open_gap_score, target_left_extend_gap_score
target_right_gap_score	target_right_open_gap_score, target_right_extend_gap_score
query_end_gap_score	query_end_open_gap_score, query_end_extend_gap_score
query_end_open_gap_score	query_left_open_gap_score, query_right_open_gap_score
query_end_extend_gap_score	query_left_extend_gap_score, query_right_extend_gap_score
query_internal_gap_score	query_internal_open_gap_score, query_internal_extend_gap_score
query_left_gap_score	query_left_open_gap_score, query_left_extend_gap_score
query_right_gap_score	query_right_open_gap_score, query_right_extend_gap_score

7.5 General gap scores

For even more fine-grained control over the gap scores, you can specify a gap scoring function. For example, the gap scoring function below disallows a gap after two nucleotides in the query sequence:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> def my_gap_score_function(start, length):
...     if start == 2:
...         return -1000
...     else:
...         return -1 * length
...
>>> aligner.query_gap_score = my_gap_score_function
>>> alignments = aligner.align("AACTT", "AATT")
>>> for alignment in alignments:
...     print(alignment)
...
```

(continues on next page)

(continued from previous page)

```
target            0 AACTT 5
                  0 -|.|| 5
query             0 -AATT 4

target            0 AACTT 5
                  0 |-.|| 5
query             0 A-ATT 4

target            0 AACTT 5
                  0 ||.-| 5
query             0 AAT-T 4

target            0 AACTT 5
                  0 ||.|- 5
query             0 AATT- 4
```

7.6 Using a pre-defined substitution matrix and gap scores

By default, a `PairwiseAligner` object is initialized with a match score of +1.0, a mismatch score of 0.0, and all gap scores equal to 0.0, While this has the benefit of being a simple scoring scheme, in general it does not give the best performance. Instead, you can use the argument `scoring` to select a predefined scoring scheme when initializing a `PairwiseAligner` object. Currently, the provided scoring schemes are `blastn` and `megablast`, which are suitable for nucleotide alignments, and `blastp`, which is suitable for protein alignments. Selecting these scoring schemes will initialize the `PairwiseAligner` object to the default scoring parameters used by BLASTN, MegaBLAST, and BLASTP, respectively.

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner(scoring="blastn")
>>> print(aligner)
Pairwise sequence aligner with parameters
  substitution_matrix: <Array object at ...>
  target_internal_open_gap_score: -7.000000
  target_internal_extend_gap_score: -2.000000
  target_left_open_gap_score: -7.000000
  target_left_extend_gap_score: -2.000000
  target_right_open_gap_score: -7.000000
  target_right_extend_gap_score: -2.000000
  query_internal_open_gap_score: -7.000000
  query_internal_extend_gap_score: -2.000000
  query_left_open_gap_score: -7.000000
  query_left_extend_gap_score: -2.000000
  query_right_open_gap_score: -7.000000
  query_right_extend_gap_score: -2.000000
  mode: global

>>> print(aligner.substitution_matrix[:, :])
     A    T    G    C    S    W    R    Y    K    M    B    V    H    D    N
A  2.0 -3.0 -3.0 -3.0 -3.0 -1.0 -1.0 -3.0 -3.0 -1.0 -3.0 -1.0 -1.0 -1.0 -2.0
T -3.0  2.0 -3.0 -3.0 -3.0 -1.0 -3.0 -1.0 -1.0 -3.0 -1.0 -3.0 -1.0 -1.0 -2.0
G -3.0 -3.0  2.0 -3.0 -1.0 -3.0 -1.0 -3.0 -1.0 -3.0 -1.0 -1.0 -3.0 -1.0 -2.0
```

(continues on next page)

(continued from previous page)

```
C -3.0 -3.0 -3.0  2.0 -1.0 -3.0 -3.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -3.0 -2.0
S -3.0 -3.0 -1.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
W -1.0 -1.0 -3.0 -3.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
R -1.0 -3.0 -1.0 -3.0 -1.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
Y -3.0 -1.0 -3.0 -1.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
K -3.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -2.0
M -1.0 -3.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
B -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
V -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
H -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
D -1.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
N -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0
```

7.7 Iterating over alignments

The `alignments` returned by `aligner.align` are a kind of immutable iterable objects (similar to `range`). While they appear similar to a `tuple` or `list` of `Alignment` objects, they are different in the sense that each `Alignment` object is created dynamically when it is needed. This approach was chosen because the number of alignments can be extremely large, in particular for poor alignments (see Section *Examples* for an example).

You can perform the following operations on `alignments`:

- `len(alignments)` returns the number of alignments stored. This function returns quickly, even if the number of alignments is huge. If the number of alignments is extremely large (typically, larger than 9,223,372,036,854,775,807, which is the largest integer that can be stored as a `long int` on 64 bit machines), `len(alignments)` will raise an `OverflowError`. A large number of alignments suggests that the alignment quality is low.

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> alignments = aligner.align("AAA", "AA")
>>> len(alignments)
3
```

- You can extract a specific alignment by index:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> alignments = aligner.align("AAA", "AA")
>>> print(alignments[2])
target            0 AAA 3
                  0 -|| 3
query             0 -AA 2

>>> print(alignments[0])
target            0 AAA 3
                  0 ||- 3
query             0 AA- 2
```

- You can iterate over alignments, for example as in

```
>>> for alignment in alignments:
...     print(alignment)
...
```

The `alignments` iterator can be converted into a `list` or `tuple`:

```
>>> alignments = list(alignments)
```

It is wise to check the number of alignments by calling `len(alignments)` before attempting to call `list(alignments)` to save all alignments as a list.

- The alignment score (which has the same value for each alignment in `alignments`) is stored as an attribute. This allows you to check the alignment score before proceeding to extract individual alignments:

```
>>> print(alignments.score)
2.0
```

7.8 Aligning to the reverse strand

By default, the pairwise aligner aligns the forward strand of the query to the forward strand of the target. To calculate the alignment score for `query` to the reverse strand of `target`, use `strand="-"`:

```
>>> from Bio import Align
>>> from Bio.Seq import reverse_complement
>>> target = "AAAACCC"
>>> query = "AACC"
>>> aligner = Align.PairwiseAligner()
>>> aligner.mismatch_score = -1
>>> aligner.internal_gap_score = -1
>>> aligner.score(target, query)  # strand is "+" by default
4.0
>>> aligner.score(target, reverse_complement(query), strand="-")
4.0
>>> aligner.score(target, query, strand="-")
0.0
>>> aligner.score(target, reverse_complement(query))
0.0
```

The alignments against the reverse strand can be obtained by specifying `strand="-"` when calling `aligner.align`:

```
>>> alignments = aligner.align(target, query)
>>> len(alignments)
1
>>> print(alignments[0])
target            0 AAAACCC 7
                  0 --||||- 7
query             0 --AACC- 4

>>> print(alignments[0].format("bed"))
target   2   6   query   4   +   2   6   0   1   4,   0,

>>> alignments = aligner.align(target, reverse_complement(query), strand="-")
```

(continues on next page)

(continued from previous page)

```
>>> len(alignments)
1
>>> print(alignments[0])
target            0 AAAACCC 7
                  0 --||||- 7
query             4 --AACC- 0

>>> print(alignments[0].format("bed"))
target   2   6   query   4   -   2   6   0   1   4,   0,

>>> alignments = aligner.align(target, query, strand="-")
>>> len(alignments)
2
>>> print(alignments[0])
target            0 AAAACCC----  7
                  0 ----------- 11
query             4 -------GGTT  0

>>> print(alignments[1])
target            0 ----AAAACCC  7
                  0 ----------- 11
query             4 GGTT-------  0
```

Note that the score for aligning `query` to the reverse strand of `target` may be different from the score for aligning the reverse complement of `query` to the forward strand of `target` if the left and right gap scores are different:

```
>>> aligner.left_gap_score = -0.5
>>> aligner.right_gap_score = -0.2
>>> aligner.score(target, query)
2.8
>>> alignments = aligner.align(target, query)
>>> len(alignments)
1
>>> print(alignments[0])
target            0 AAAACCC 7
                  0 --||||- 7
query             0 --AACC- 4

>>> aligner.score(target, reverse_complement(query), strand="-")
3.1
>>> alignments = aligner.align(target, reverse_complement(query), strand="-")
>>> len(alignments)
1
>>> print(alignments[0])
target            0 AAAACCC 7
                  0 --||||- 7
query             4 --AACC- 0
```

7.9 Substitution matrices

Substitution matrices [Durbin1998] provide the scoring terms for classifying how likely two different residues are to substitute for each other. This is essential in doing sequence comparisons. Biopython provides a ton of common substitution matrices, including the famous PAM and BLOSUM series of matrices, and also provides functionality for creating your own substitution matrices.

7.9.1 Array objects

You can think of substitutions matrices as 2D arrays in which the indices are letters (nucleotides or amino acids) rather than integers. The `Array` class in `Bio.Align.substitution_matrices` is a subclass of numpy arrays that supports indexing both by integers and by specific strings. An `Array` instance can either be a one-dimensional array or a square two-dimensional arrays. A one-dimensional `Array` object can for example be used to store the nucleotide frequency of a DNA sequence, while a two-dimensional `Array` object can be used to represent a scoring matrix for sequence alignments.

To create a one-dimensional `Array`, only the alphabet of allowed letters needs to be specified:

```
>>> from Bio.Align.substitution_matrices import Array
>>> counts = Array("ACGT")
>>> print(counts)
A 0.0
C 0.0
G 0.0
T 0.0
```

The allowed letters are stored in the `alphabet` property:

```
>>> counts.alphabet
'ACGT'
```

This property is read-only; modifying the underlying `_alphabet` attribute may lead to unexpected results. Elements can be accessed both by letter and by integer index:

```
>>> counts["C"] = -3
>>> counts[2] = 7
>>> print(counts)
A  0.0
C -3.0
G  7.0
T  0.0

>>> counts[1]
-3.0
```

Using a letter that is not in the alphabet, or an index that is out of bounds, will cause a `IndexError`:

```
>>> counts["U"]
Traceback (most recent call last):
    ...
IndexError: 'U'
>>> counts["X"] = 6
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    ...
IndexError: 'X'
>>> counts[7]
Traceback (most recent call last):
    ...
IndexError: index 7 is out of bounds for axis 0 with size 4
```

A two-dimensional `Array` can be created by specifying `dims=2`:

```
>>> from Bio.Align.substitution_matrices import Array
>>> counts = Array("ACGT", dims=2)
>>> print(counts)
    A   C   G   T
A 0.0 0.0 0.0 0.0
C 0.0 0.0 0.0 0.0
G 0.0 0.0 0.0 0.0
T 0.0 0.0 0.0 0.0
```

Again, both letters and integers can be used for indexing, and specifying a letter that is not in the alphabet will cause an `IndexError`:

```
>>> counts["A", "C"] = 12.0
>>> counts[2, 1] = 5.0
>>> counts[3, "T"] = -2
>>> print(counts)
    A    C   G    T
A 0.0 12.0 0.0  0.0
C 0.0  0.0 0.0  0.0
G 0.0  5.0 0.0  0.0
T 0.0  0.0 0.0 -2.0

>>> counts["X", 1]
Traceback (most recent call last):
    ...
IndexError: 'X'
>>> counts["A", 5]
Traceback (most recent call last):
    ...
IndexError: index 5 is out of bounds for axis 1 with size 4
```

Selecting a row or column from the two-dimensional array will return a one-dimensional `Array`:

```
>>> counts = Array("ACGT", dims=2)
>>> counts["A", "C"] = 12.0
>>> counts[2, 1] = 5.0
>>> counts[3, "T"] = -2
```

```
>>> counts["G"]
Array([0., 5., 0., 0.],
      alphabet='ACGT')
>>> counts[:, "C"]
Array([12.,  0.,  5.,  0.],
      alphabet='ACGT')
```

`Array` objects can thus be used as an array and as a dictionary. They can be converted to plain numpy arrays or plain dictionary objects:

```
>>> import numpy as np
>>> x = Array("ACGT")
>>> x["C"] = 5
```

```
>>> x
Array([0., 5., 0., 0.],
      alphabet='ACGT')
>>> a = np.array(x)  # create a plain numpy array
>>> a
array([0., 5., 0., 0.])
>>> d = dict(x)  # create a plain dictionary
>>> d
{'A': 0.0, 'C': 5.0, 'G': 0.0, 'T': 0.0}
```

While the alphabet of an `Array` is usually a string, you may also use a tuple of (immutable) objects. This is used for example for a codon substitution matrix (as in the `substitution_matrices.load("SCHNEIDER")` example shown later), where the keys are not individual nucleotides or amino acids but instead three-nucleotide codons.

While the `alphabet` property of an `Array` is immutable, you can create a new `Array` object by selecting the letters you are interested in from the alphabet. For example,

```
>>> a = Array("ABCD", dims=2, data=np.arange(16).reshape(4, 4))
>>> print(a)
     A    B    C    D
A  0.0  1.0  2.0  3.0
B  4.0  5.0  6.0  7.0
C  8.0  9.0 10.0 11.0
D 12.0 13.0 14.0 15.0

>>> b = a.select("CAD")
>>> print(b)
     C    A    D
C 10.0  8.0 11.0
A  2.0  0.0  3.0
D 14.0 12.0 15.0
```

Note that this also allows you to reorder the alphabet.

Data for letters that are not found in the alphabet are set to zero:

```
>>> c = a.select("DEC")
>>> print(c)
     D   E    C
D 15.0 0.0 14.0
E  0.0 0.0  0.0
C 11.0 0.0 10.0
```

As the `Array` class is a subclass of numpy array, it can be used as such. A `ValueError` is triggered if the `Array` objects appearing in a mathematical operation have different alphabets, for example

```
>>> from Bio.Align.substitution_matrices import Array
>>> d = Array("ACGT")
```

(continues on next page)

(continued from previous page)

```
>>> r = Array("ACGU")
>>> d + r
Traceback (most recent call last):
    ...
ValueError: alphabets are inconsistent
```

7.9.2 Calculating a substitution matrix from a pairwise sequence alignment

As `Array` is a subclass of a numpy array, you can apply mathematical operations on an `Array` object in much the same way. Here, we illustrate this by calculating a scoring matrix from the alignment of the 16S ribosomal RNA gene sequences of *Escherichia coli* and *Bacillus subtilis*. First, we create a `PairwiseAligner` object (see Chapter *Pairwise sequence alignment*) and initialize it with the default scores used by `blastn`:

```
>>> from Bio.Align import PairwiseAligner
>>> aligner = PairwiseAligner(scoring="blastn")
>>> aligner.mode = "local"
```

Next, we read in the 16S ribosomal RNA gene sequence of *Escherichia coli* and *Bacillus subtilis* (provided in `Tests/Align/ecoli.fa` and `Tests/Align/bsubtilis.fa`), and align them to each other:

```
>>> from Bio import SeqIO
>>> sequence1 = SeqIO.read("ecoli.fa", "fasta")
>>> sequence2 = SeqIO.read("bsubtilis.fa", "fasta")
>>> alignments = aligner.align(sequence1, sequence2)
```

The number of alignments generated is very large:

```
>>> len(alignments)
1990656
```

However, as they only differ trivially from each other, we arbitrarily choose the first alignment, and count the number of each substitution:

```
>>> alignment = alignments[0]
>>> substitutions = alignment.substitutions
>>> print(substitutions)
      A     C     G     T
A 307.0  19.0  34.0  19.0
C  15.0 280.0  25.0  29.0
G  34.0  24.0 401.0  20.0
T  24.0  36.0  20.0 228.0
```

We normalize against the total number to find the probability of each substitution, and create a symmetric matrix of observed frequencies:

```
>>> observed_frequencies = substitutions / substitutions.sum()
>>> observed_frequencies = (observed_frequencies + observed_frequencies.transpose()) / 2.
↪0
>>> print(format(observed_frequencies, "%.4f"))
       A      C      G      T
A 0.2026 0.0112 0.0224 0.0142
C 0.0112 0.1848 0.0162 0.0215
```

(continues on next page)

(continued from previous page)

```
G 0.0224 0.0162 0.2647 0.0132
T 0.0142 0.0215 0.0132 0.1505
```

The background probability is the probability of finding an A, C, G, or T nucleotide in each sequence separately. This can be calculated as the sum of each row or column:

```
>>> background = observed_frequencies.sum(0)
>>> print(format(background, "%.4f"))
A 0.2505
C 0.2337
G 0.3165
T 0.1993
```

The number of substitutions expected at random is simply the product of the background distribution with itself:

```
>>> expected_frequencies = background[:, None].dot(background[None, :])
>>> print(format(expected_frequencies, "%.4f"))
       A      C      G      T
A 0.0627 0.0585 0.0793 0.0499
C 0.0585 0.0546 0.0740 0.0466
G 0.0793 0.0740 0.1002 0.0631
T 0.0499 0.0466 0.0631 0.0397
```

The scoring matrix can then be calculated as the logarithm of the odds-ratio of the observed and the expected probabilities:

```
>>> oddsratios = observed_frequencies / expected_frequencies
>>> import numpy as np
>>> scoring_matrix = np.log2(oddsratios)
>>> print(scoring_matrix)
     A    C    G    T
A  1.7 -2.4 -1.8 -1.8
C -2.4  1.8 -2.2 -1.1
G -1.8 -2.2  1.4 -2.3
T -1.8 -1.1 -2.3  1.9
```

The matrix can be used to set the substitution matrix for the pairwise aligner (see Chapter *Pairwise sequence alignment*):

```
>>> aligner.substitution_matrix = scoring_matrix
```

7.9.3 Calculating a substitution matrix from a multiple sequence alignment

In this example, we'll first read a protein sequence alignment from the Clustalw file protein.aln (also available online here)

```
>>> from Bio import Align
>>> filename = "protein.aln"
>>> alignment = Align.read(filename, "clustal")
```

Section *ClustalW* contains more information on doing this.

The `substitutions` property of the alignment stores the number of times different residues substitute for each other:

```
>>> substitutions = alignment.substitutions
```

To make the example more readable, we'll select only amino acids with polar charged side chains:

```
>>> substitutions = substitutions.select("DEHKR")
>>> print(substitutions)
       D      E      H      K      R
D 2360.0  270.0   15.0    1.0   48.0
E  241.0 3305.0   15.0   45.0    2.0
H    0.0   18.0 1235.0    8.0    0.0
K    0.0    9.0   24.0 3218.0  130.0
R    2.0    2.0   17.0  103.0 2079.0
```

Rows and columns for other amino acids were removed from the matrix.

Next, we normalize the matrix and make it symmetric.

```
>>> observed_frequencies = substitutions / substitutions.sum()
>>> observed_frequencies = (observed_frequencies + observed_frequencies.transpose()) / 2.
↪0
>>> print(format(observed_frequencies, "%.4f"))
       D      E      H      K      R
D 0.1795 0.0194 0.0006 0.0000 0.0019
E 0.0194 0.2514 0.0013 0.0021 0.0002
H 0.0006 0.0013 0.0939 0.0012 0.0006
K 0.0000 0.0021 0.0012 0.2448 0.0089
R 0.0019 0.0002 0.0006 0.0089 0.1581
```

Summing over rows or columns gives the relative frequency of occurrence of each residue:

```
>>> background = observed_frequencies.sum(0)
>>> print(format(background, "%.4f"))
D 0.2015
E 0.2743
H 0.0976
K 0.2569
R 0.1697

>>> sum(background) == 1.0
True
```

The expected frequency of residue pairs is then

```
>>> expected_frequencies = background[:, None].dot(background[None, :])
>>> print(format(expected_frequencies, "%.4f"))
       D      E      H      K      R
D 0.0406 0.0553 0.0197 0.0518 0.0342
E 0.0553 0.0752 0.0268 0.0705 0.0465
H 0.0197 0.0268 0.0095 0.0251 0.0166
K 0.0518 0.0705 0.0251 0.0660 0.0436
R 0.0342 0.0465 0.0166 0.0436 0.0288
```

Here, `background[:, None]` creates a 2D array consisting of a single column with the values of `expected_frequencies`, and `rxpected_frequencies[None, :]` a 2D array with these values as a single row. Taking their dot product (inner product) creates a matrix of expected frequencies where each entry consists of two

`expected_frequencies` values multiplied with each other. For example, `expected_frequencies['D', 'E']` is equal to `residue_frequencies['D'] * residue_frequencies['E']`.

We can now calculate the log-odds matrix by dividing the observed frequencies by the expected frequencies and taking the logarithm:

```
>>> import numpy as np
>>> scoring_matrix = np.log2(observed_frequencies / expected_frequencies)
>>> print(scoring_matrix)
      D    E    H     K    R
D   2.1 -1.5 -5.1 -10.4 -4.2
E  -1.5  1.7 -4.4  -5.1 -8.3
H  -5.1 -4.4  3.3  -4.4 -4.7
K -10.4 -5.1 -4.4   1.9 -2.3
R  -4.2 -8.3 -4.7  -2.3  2.5
```

This matrix can be used as the substitution matrix when performing alignments. For example,

```
>>> from Bio.Align import PairwiseAligner
>>> aligner = PairwiseAligner()
>>> aligner.substitution_matrix = scoring_matrix
>>> aligner.gap_score = -3.0
>>> alignments = aligner.align("DEHEK", "DHHKK")
>>> print(alignments[0])
target            0 DEHEK 5
                  0 |.|.| 5
query             0 DHHKK 5

>>> print("%.2f" % alignments.score)
-2.18
>>> score = (
...     scoring_matrix["D", "D"]
...     + scoring_matrix["E", "H"]
...     + scoring_matrix["H", "H"]
...     + scoring_matrix["E", "K"]
...     + scoring_matrix["K", "K"]
... )
>>> print("%.2f" % score)
-2.18
```

(see Chapter *Pairwise sequence alignment* for details).

7.9.4 Reading `Array` objects from file

`Bio.Align.substitution_matrices` includes a parser to read one- and two-dimensional `Array` objects from file. One-dimensional arrays are represented by a simple two-column format, with the first column containing the key and the second column the corresponding value. For example, the file `hg38.chrom.sizes` (obtained from UCSC), available in the `Tests/Align` subdirectory of the Biopython distribution, contains the size in nucleotides of each chromosome in human genome assembly hg38:

```
chr1    248956422
chr2    242193529
chr3    198295559
```

(continues on next page)

(continued from previous page)

```
chr4    190214555
...
chrUn_KI270385v1    990
chrUn_KI270423v1    981
chrUn_KI270392v1    971
chrUn_KI270394v1    970
```

To parse this file, use

```
>>> from Bio.Align import substitution_matrices
>>> with open("hg38.chrom.sizes") as handle:
...     table = substitution_matrices.read(handle)
...
>>> print(table)
chr1 248956422.0
chr2 242193529.0
chr3 198295559.0
chr4 190214555.0
...
chrUn_KI270423v1       981.0
chrUn_KI270392v1       971.0
chrUn_KI270394v1       970.0
```

Use `dtype=int` to read the values as integers:

```
>>> with open("hg38.chrom.sizes") as handle:
...     table = substitution_matrices.read(handle, int)
...
>>> print(table)
chr1 248956422
chr2 242193529
chr3 198295559
chr4 190214555
...
chrUn_KI270423v1       981
chrUn_KI270392v1       971
chrUn_KI270394v1       970
```

For two-dimensional arrays, we follow the file format of substitution matrices provided by NCBI. For example, the BLOSUM62 matrix, which is the default substitution matrix for NCBI's protein-protein BLAST [Altschul1990] program `blastp`, is stored as follows:

```
#  Matrix made by matblas from blosum62.iij
#  * column uses minimum score
#  BLOSUM Clustered Scoring Matrix in 1/2 Bit Units
#  Blocks Database = /data/blocks_5.0/blocks.dat
#  Cluster Percentage: >= 62
#  Entropy =   0.6979, Expected =  -0.5209
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1  0 -4
R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1  0 -1 -4
N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  3  0 -1 -4
D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3  4  1 -1 -4
```

(continues on next page)

(continued from previous page)

```
C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1 -3 -3 -2 -4
Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2  0  3 -1 -4
E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3 -1 -2 -1 -4
H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3  0  0 -1 -4
...
```

This file is included in the Biopython distribution under `Bio/Align/substitution_matrices/data`. To parse this file, use

```
>>> from Bio.Align import substitution_matrices
>>> with open("BLOSUM62") as handle:
...     matrix = substitution_matrices.read(handle)
...
>>> print(matrix.alphabet)
ARNDCQEGHILKMFPSTWYVBZX*
>>> print(matrix["A", "D"])
-2.0
```

The header lines starting with # are stored in the attribute `header`:

```
>>> matrix.header[0]
'Matrix made by matblas from blosum62.iij'
```

We can now use this matrix as the substitution matrix on an aligner object:

```
>>> from Bio.Align import PairwiseAligner
>>> aligner = PairwiseAligner()
>>> aligner.substitution_matrix = matrix
```

To save an Array object, create a string first:

```
>>> text = str(matrix)
>>> print(text)
#  Matrix made by matblas from blosum62.iij
#  * column uses minimum score
#  BLOSUM Clustered Scoring Matrix in 1/2 Bit Units
#  Blocks Database = /data/blocks_5.0/blocks.dat
#  Cluster Percentage: >= 62
#  Entropy =   0.6979, Expected =  -0.5209
     A    R    N    D    C    Q    E    G    H    I    L    K    M    F    P    S ...
A  4.0 -1.0 -2.0 -2.0  0.0 -1.0 -1.0  0.0 -2.0 -1.0 -1.0 -1.0 -1.0 -2.0 -1.0  1.0 ...
R -1.0  5.0  0.0 -2.0 -3.0  1.0  0.0 -2.0  0.0 -3.0 -2.0  2.0 -1.0 -3.0 -2.0 -1.0 ...
N -2.0  0.0  6.0  1.0 -3.0  0.0  0.0  0.0  1.0 -3.0 -3.0  0.0 -2.0 -3.0 -2.0  1.0 ...
D -2.0 -2.0  1.0  6.0 -3.0  0.0  2.0 -1.0 -1.0 -3.0 -4.0 -1.0 -3.0 -3.0 -1.0  0.0 ...
C  0.0 -3.0 -3.0 -3.0  9.0 -3.0 -4.0 -3.0 -3.0 -1.0 -1.0 -3.0 -1.0 -2.0 -3.0 -1.0 ...
...
```

and write the `text` to a file.

7.9.5 Loading predefined substitution matrices

Biopython contains a large set of substitution matrices defined in the literature, including BLOSUM (Blocks Substitution Matrix) [Henikoff1992] and PAM (Point Accepted Mutation) matrices [Dayhoff1978]. These matrices are available as flat files in the `Bio/Align/substitution_matrices/data` directory, and can be loaded into Python using the `load` function in the `substitution_matrices` submodule. For example, the BLOSUM62 matrix can be loaded by running

```
>>> from Bio.Align import substitution_matrices
>>> m = substitution_matrices.load("BLOSUM62")
```

This substitution matrix has an alphabet consisting of the 20 amino acids used in the genetic code, the three ambiguous amino acids B (asparagine or aspartic acid), Z (glutamine or glutamic acid), and X (representing any amino acid), and the stop codon represented by an asterisk:

```
>>> m.alphabet
'ARNDCQEGHILKMFPSTWYVBZX*'
```

To get a full list of available substitution matrices, use `load` without an argument:

```
>>> substitution_matrices.load()
['BENNER22', 'BENNER6', 'BENNER74', 'BLASTN', 'BLASTP', 'BLOSUM45', 'BLOSUM50', ...,
↪'TRANS']
```

Note that the substitution matrix provided by Schneider *et al.* [Schneider2005] uses an alphabet consisting of three-nucleotide codons:

```
>>> m = substitution_matrices.load("SCHNEIDER")
>>> m.alphabet
('AAA', 'AAC', 'AAG', 'AAT', 'ACA', 'ACC', 'ACG', 'ACT', ..., 'TTG', 'TTT')
```

7.10 Examples

Suppose you want to do a global pairwise alignment between the same two hemoglobin sequences from above (`HBA_HUMAN`, `HBB_HUMAN`) stored in `alpha.faa` and `beta.faa`:

```
>>> from Bio import Align
>>> from Bio import SeqIO
>>> seq1 = SeqIO.read("alpha.faa", "fasta")
>>> seq2 = SeqIO.read("beta.faa", "fasta")
>>> aligner = Align.PairwiseAligner()
>>> score = aligner.score(seq1.seq, seq2.seq)
>>> print(score)
72.0
```

showing an alignment score of 72.0. To see the individual alignments, do

```
>>> alignments = aligner.align(seq1.seq, seq2.seq)
```

In this example, the total number of optimal alignments is huge (more than 4×10^{37}), and calling `len(alignments)` will raise an `OverflowError`:

```
>>> len(alignments)
Traceback (most recent call last):
...
OverflowError: number of optimal alignments is larger than 9223372036854775807
```

Let's have a look at the first alignment:

```
>>> alignment = alignments[0]
```

The alignment object stores the alignment score, as well as the alignment itself:

```
>>> print(alignment.score)
72.0
>>> print(alignment)
target            0 MV-LS-PAD--KTN--VK-AA-WGKV-----GAHAGEYGAEALE-RMFLSF----P-TTK
                  0 ||-|--|----|----|--|--||||-----|---||--|--|--|--|------|-|--
query             0 MVHL-TP--EEK--SAV-TA-LWGKVNVDEVG---GE--A--L-GR--L--LVVYPWT--

target           41 TY--FPHF----DLSHGS---AQVK-G------HGKKV--A--DA-LTNAVAHV-DDMPN
                 60 ----|--|----|||------|-|--|------|||||--|--|--|--|--|--|---|
query            39 --QRF--FESFGDLS---TPDA-V-MGNPKVKAHGKKVLGAFSD-GL--A--H-LD---N

target           79 ALS----A-LSD-LHAH--KLR-VDPV-NFK-LLSHC---LLVT--LAAHLPA----EFT
                120 -|-----|-||--||----||--|||--||--||------|-|---||-|-------|||
query            81 -L-KGTFATLS-ELH--CDKL-HVDP-ENF-RLL---GNVL-V-CVLA-H---HFGKEFT

target          119 PA-VH-ASLDKFLAS---VSTV------LTS--KYR- 142
                180 |--|--|------|----|--|------|----||-- 217
query           124 P-PV-QA------A-YQKV--VAGVANAL--AHKY-H 147
```

Better alignments are usually obtained by penalizing gaps: higher costs for opening a gap and lower costs for extending an existing gap. For amino acid sequences match scores are usually encoded in matrices like `PAM` or `BLOSUM`. Thus, a more meaningful alignment for our example can be obtained by using the BLOSUM62 matrix, together with a gap open penalty of 10 and a gap extension penalty of 0.5:

```
>>> from Bio import Align
>>> from Bio import SeqIO
>>> from Bio.Align import substitution_matrices
>>> seq1 = SeqIO.read("alpha.faa", "fasta")
>>> seq2 = SeqIO.read("beta.faa", "fasta")
>>> aligner = Align.PairwiseAligner()
>>> aligner.open_gap_score = -10
>>> aligner.extend_gap_score = -0.5
>>> aligner.substitution_matrix = substitution_matrices.load("BLOSUM62")
>>> score = aligner.score(seq1.seq, seq2.seq)
>>> print(score)
292.5
>>> alignments = aligner.align(seq1.seq, seq2.seq)
>>> len(alignments)
2
>>> print(alignments[0].score)
292.5
>>> print(alignments[0])
```

(continues on next page)

(continued from previous page)

```
target             0 MV-LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF-DLS-----HGS
                   0 ||-|.|..|..|.|.||||--...|.|.|||.|.....|.|...|..|-|||-----.|.
query              0 MVHLTPEEKSAVTALWGKV--NVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGN

target            53 AQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAH
                  60 ..||.|||||..|.....||.|........||.||..||.|||.||.||...|...||.|
query             58 PKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLAHH

target           113 LPAEFTPAVHASLDKFLASVSTVLTSKYR 142
                 120 ...||||.|.|...|..|.|...|..||. 149
query            118 FGKEFTPPVQAAYQKVVAGVANALAHKYH 147
```

This alignment has the same score that we obtained earlier with EMBOSS needle using the same sequences and the same parameters.

To perform a local alignment, set `aligner.mode` to `'local'`:

```
>>> aligner.mode = "local"
>>> aligner.open_gap_score = -10
>>> aligner.extend_gap_score = -1
>>> alignments = aligner.align("LSPADKTNVKAA", "PEEKSAV")
>>> print(len(alignments))
1
>>> alignment = alignments[0]
>>> print(alignment)
target             2 PADKTNV 9
                   0 |..|..| 7
query              0 PEEKSAV 7

>>> print(alignment.score)
16.0
```

7.11 Generalized pairwise alignments

In most cases, `PairwiseAligner` is used to perform alignments of sequences (strings or `Seq` objects) consisting of single-letter nucleotides or amino acids. More generally, `PairwiseAligner` can also be applied to lists or tuples of arbitrary objects. This section will describe some examples of such generalized pairwise alignments.

7.11.1 Generalized pairwise alignments using a substitution matrix and alphabet

Schneider *et al.* [Schneider2005] created a substitution matrix for aligning three-nucleotide codons (see *below* in section *Substitution matrices* for more information). This substitution matrix is associated with an alphabet consisting of all three-letter codons:

```
>>> from Bio.Align import substitution_matrices
>>> m = substitution_matrices.load("SCHNEIDER")
>>> m.alphabet
('AAA', 'AAC', 'AAG', 'AAT', 'ACA', 'ACC', 'ACG', 'ACT', ..., 'TTG', 'TTT')
```

We can use this matrix to align codon sequences to each other:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> aligner.substitution_matrix = m
>>> aligner.gap_score = -1.0
>>> s1 = ("AAT", "CTG", "TTT", "TTT")
>>> s2 = ("AAT", "TTA", "TTT")
>>> alignments = aligner.align(s1, s2)
>>> len(alignments)
2
>>> print(alignments[0])
AAT CTG TTT TTT
|||  ...  |||  ---
AAT TTA TTT  ---

>>> print(alignments[1])
AAT CTG TTT TTT
|||  ...  ---  |||
AAT TTA ---  TTT
```

Note that aligning `TTT` to `TTA`, as in this example:

```
AAT CTG TTT TTT
|||  ---  ...  |||
AAT ---  TTA TTT
```

would get a much lower score:

```
>>> print(m["CTG", "TTA"])
7.6
>>> print(m["TTT", "TTA"])
-0.3
```

presumably because `CTG` and `TTA` both code for leucine, while `TTT` codes for phenylalanine. The three-letter codon substitution matrix also reveals a preference among codons representing the same amino acid. For example, `TTA` has a preference for `CTG` preferred compared to `CTC`, though all three code for leucine:

```
>>> s1 = ("AAT", "CTG", "CTC", "TTT")
>>> s2 = ("AAT", "TTA", "TTT")
>>> alignments = aligner.align(s1, s2)
>>> len(alignments)
1
>>> print(alignments[0])
AAT CTG CTC TTT
|||  ...  ---  |||
AAT TTA ---  TTT

>>> print(m["CTC", "TTA"])
6.5
```

7.11.2 Generalized pairwise alignments using match/mismatch scores and an alphabet

Using the three-letter amino acid symbols, the sequences above translate to

```
>>> s1 = ("Asn", "Leu", "Leu", "Phe")
>>> s2 = ("Asn", "Leu", "Phe")
```

We can align these sequences directly to each other by using a three-letter amino acid alphabet:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> aligner.alphabet = ['Ala', 'Arg', 'Asn', 'Asp', 'Cys',
...                     'Gln', 'Glu', 'Gly', 'His', 'Ile',
...                     'Leu', 'Lys', 'Met', 'Phe', 'Pro',
...                     'Ser', 'Thr', 'Trp', 'Tyr', 'Val']  # fmt: skip
...
```

We use +6/-1 match and mismatch scores as an approximation of the BLOSUM62 matrix, and align these sequences to each other:

```
>>> aligner.match = +6
>>> aligner.mismatch = -1
>>> alignments = aligner.align(s1, s2)
>>> print(len(alignments))
2
>>> print(alignments[0])
Asn Leu Leu Phe
|||  |||  ---  |||
Asn Leu  ---  Phe

>>> print(alignments[1])
Asn Leu Leu Phe
|||  ---  |||  |||
Asn  ---  Leu Phe

>>> print(alignments.score)
18.0
```

7.11.3 Generalized pairwise alignments using match/mismatch scores and integer sequences

Internally, the first step when performing an alignment is to replace the two sequences by integer arrays consisting of the indices of each letter in each sequence in the alphabet associated with the aligner. This step can be bypassed by passing integer arrays directly:

```
>>> import numpy as np
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> s1 = np.array([2, 10, 10, 13], np.int32)
>>> s2 = np.array([2, 10, 13], np.int32)
>>> aligner.match = +6
```

(continues on next page)

(continued from previous page)

```
>>> aligner.mismatch = -1
>>> alignments = aligner.align(s1, s2)
>>> print(len(alignments))
2
>>> print(alignments[0])
2 10 10 13
| || -- ||
2 10 -- 13

>>> print(alignments[1])
2 10 10 13
| -- || ||
2 -- 10 13

>>> print(alignments.score)
18.0
```

Note that the indices should consist of 32-bit integers, as specified in this example by `numpy.int32`.

Unknown letters can again be included by defining a wildcard character, and using the corresponding Unicode code point number as the index:

```
>>> aligner.wildcard = "?"
>>> ord(aligner.wildcard)
63
>>> s2 = np.array([2, 63, 13], np.int32)
>>> aligner.gap_score = -3
>>> alignments = aligner.align(s1, s2)
>>> print(len(alignments))
2
>>> print(alignments[0])
2 10 10 13
| .. -- ||
2 63 -- 13

>>> print(alignments[1])
2 10 10 13
| -- .. ||
2 -- 63 13

>>> print(alignments.score)
9.0
```

7.11.4 Generalized pairwise alignments using a substitution matrix and integer sequences

Integer sequences can also be aligned using a substitution matrix, in this case a numpy square array without an alphabet associated with it. In this case, all index values must be non-negative, and smaller than the size of the substitution matrix:

```
>>> from Bio import Align
>>> import numpy as np
>>> aligner = Align.PairwiseAligner()
>>> m = np.eye(5)
>>> m[0, 1:] = m[1:, 0] = -2
>>> m[2, 2] = 3
>>> print(m)
[[ 1. -2. -2. -2. -2.]
 [-2.  1.  0.  0.  0.]
 [-2.  0.  3.  0.  0.]
 [-2.  0.  0.  1.  0.]
 [-2.  0.  0.  0.  1.]]
>>> aligner.substitution_matrix = m
>>> aligner.gap_score = -1
>>> s1 = np.array([0, 2, 3, 4], np.int32)
>>> s2 = np.array([0, 3, 2, 1], np.int32)
>>> alignments = aligner.align(s1, s2)
>>> print(len(alignments))
2
>>> print(alignments[0])
0 - 2 3 4
| - | . -
0 3 2 1 -

>>> print(alignments[1])
0 - 2 3 4
| - | - .
0 3 2 - 1

>>> print(alignments.score)
2.0
```

7.12 Codon alignments

The `CodonAligner` class in the `Bio.Align` module implements a specialized aligner for aligning a nucleotide sequence to the amino acid sequence it encodes. Such alignments are non-trivial if frameshifts occur during translation.

7.12.1 Aligning a nucleotide sequence to an amino acid sequence

To align a nucleotide sequence to an amino acid sequence, first create a `CodonAligner` object:

```
>>> from Bio import Align
>>> aligner = Align.CodonAligner()
```

The `CodonAligner` object `aligner` stores the alignment parameters to be used for the alignments:

```
>>> print(aligner)
Codon aligner with parameters
  wildcard: 'X'
  match_score: 1.0
  mismatch_score: 0.0
  frameshift_minus_two_score: -3.0
  frameshift_minus_one_score: -3.0
  frameshift_plus_one_score: -3.0
  frameshift_plus_two_score: -3.0
```

The `wildcard`, `match_score`, and `mismatch_score` parameters are defined in the same was as for the `PairwiseAligner` class described above (see Section *The pairwise aligner object*). The values specified by the `frameshift_minus_two_score`, `frameshift_minus_one_score`, `frameshift_plus_one_score`, and `frameshift_plus_two_score` parameters are added to the alignment score whenever a -2, -1, +1, or +2 frame shift, respectively, occurs in the alignment. By default, the frame shift scores are set to -3.0. Similar to the `PairwiseAligner` class (Table *Meta-attributes of the pairwise aligner objects.*), the `CodonAligner` class defines additional attributes that refer to a number of these values collectively, as shown in Table *Meta-attributes of CodonAligner objects.*.

Table 2: Meta-attributes of CodonAligner objects.

Meta-attribute	Attributes it maps to
`frameshift_minus`	`frameshift_minus_two_score`, `frameshift_minus_one_score`
`frameshift_plus_`	`frameshift_plus_two_score`, `frameshift_plus_one_score`
`frameshift_two_s`	`frameshift_minus_two_score`, `frameshift_plus_two_score`
`frameshift_one_s`	`frameshift_minus_one_score`, `frameshift_plus_one_score`
`frameshift_score`	`frameshift_minus_two_score`, `frameshift_minus_one_score`, `frameshift_plus_one_score`, `frameshift_plus_two_score`

Now let's consider two nucleotide sequences and the amino acid sequences they encode:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> nuc1 = Seq("TCAGGGACTGCGAGAACCAAGCTACTGCTGCTGCTGGCTGCGCTCTGCGCCGCAGGTGGGGCGCTGGAG")
>>> rna1 = SeqRecord(nuc1, id="rna1")
>>> nuc2 = Seq("TCAGGGACTTCGAGAACCAAGCGCTCCTGCTGCTGGCTGCGCTCGGCGCCGCAGGTGGAGCACTGGAG")
>>> rna2 = SeqRecord(nuc2, id="rna2")
>>> aa1 = Seq("SGTARTKLLLLLAALCAAGGALE")
>>> aa2 = Seq("SGTSRTKRLLLLAALGAAGGALE")
>>> pro1 = SeqRecord(aa1, id="pro1")
>>> pro2 = SeqRecord(aa2, id="pro2")
```

While the two protein sequences both consist of 23 amino acids, the first nucleotide sequence consists of $3 \times 23 = 69$ nucleotides while the second nucleotide sequence tonsists of only 68 nucleotides:

```
>>> len(pro1)
23
>>> len(pro2)
23
>>> len(rna1)
69
>>> len(rna2)
68
```

This is due to a -1 frame shift event during translation of the second nucleotide sequence. Use `CodonAligner.align` to align `rna1` to `pro1`, and `rna2` to `pro2`, returning an iterator of `Alignment` objects:

```
>>> alignments1 = aligner.align(pro1, rna1)
>>> len(alignments1)
1
>>> alignment1 = next(alignments1)
>>> print(alignment1)
pro1              0 S  G  T  A  R  T  K  L  L  L  L  L  A  A  L  C  A  A  G  G
rna1              0 TCAGGGACTGCGAGAACCAAGCTACTGCTGCTGCTGGCTGCGCTCTGCGCCGCAGGTGGG

pro1             20 A  L  E   23
rna1             60 GCGCTGGAG 69

>>> alignment1.coordinates
array([[ 0, 23],
       [ 0, 69]])
>>> alignment1[0]
'SGTARTKLLLLLAALCAAGGALE'
>>> alignment1[1]
'TCAGGGACTGCGAGAACCAAGCTACTGCTGCTGCTGGCTGCGCTCTGCGCCGCAGGTGGGGCGCTGGAG'
>>> alignments2 = aligner.align(pro2, rna2)
>>> len(alignments2)
1
>>> alignment2 = next(alignments2)
>>> print(alignment2)
pro2              0 S  G  T  S  R  T  K  R   8
rna2              0 TCAGGGACTTCGAGAACCAAGCGC 24

pro2              8 L  L  L  L  A  A  L  G  A  A  G  G  A  L  E   23
rna2             23 CTCCTGCTGCTGGCTGCGCTCGGCGCCGCAGGTGGAGCACTGGAG 68

>>> alignment2[0]
'SGTSRTKRLLLLAALGAAGGALE'
>>> alignment2[1]
'TCAGGGACTTCGAGAACCAAGCGCCTCCTGCTGCTGGCTGCGCTCGGCGCCGCAGGTGGAGCACTGGAG'
>>> alignment2.coordinates
array([[ 0,  8,  8, 23],
       [ 0, 24, 23, 68]])
```

While `alignment1` is a continuous alignment of the 69 nucleotides to the 23 amino acids, in `alignment2` we find a -1 frame shift after 24 nucleotides. As `alignment2[1]` contains the nucleotide sequence after applying the -1 frame shift, it is one nucleotide longer than `nuc2` and can be translated directly, resulting in the amino acid sequence `aa2`:

```
>>> from Bio.Seq import translate
>>> len(nuc2)
68
>>> len(alignment2[1])
69
>>> translate(alignment2[1])
'SGTSRTKRLLLLAALGAAGGALE'
>>> _ == aa2
True
```

The alignment score is stored as an attribute on the `alignments1` and `alignments2` iterators, and on the individual alignments `alignment1` and `alignment2`:

```
>>> alignments1.score
23.0
>>> alignment1.score
23.0
>>> alignments2.score
20.0
>>> alignment2.score
20.0
```

where the score of the `rna1`-`pro1` alignment is equal to the number of aligned amino acids, and the score of the `rna2`-`pro2` alignment is 3 less due to the penalty for the frame shift. To calculate the alignment score without calculating the alignment itself, the `score` method can be used:

```
>>> score = aligner.score(pro1, rna1)
>>> print(score)
23.0
>>> score = aligner.score(pro2, rna2)
>>> print(score)
20.0
```

7.12.2 Generating a multiple sequence alignment of codon sequences

Suppose we have a third related amino acid sequence and its associated nucleotide sequence:

```
>>> aa3 = Seq("MGTALLLLAALCAAGGALE")
>>> pro3 = SeqRecord(aa3, id="pro3")
>>> nuc3 = Seq("ATGGGAACCGCGCTGCTTTTGCTACTGGCCGCGCTCTGCGCCGCAGGTGGGGCCCTGGAG")
>>> rna3 = SeqRecord(nuc3, id="rna3")
>>> nuc3.translate() == aa3
True
```

As above, we use the `CodonAligner` to align the nucleotide sequence to the amino acid sequence:

```
>>> alignments3 = aligner.align(pro3, rna3)
>>> len(alignments3)
1
>>> alignment3 = next(alignments3)
>>> print(alignment3)
pro3             0 M  G  T  A  L  L  L  L  L  A  A  L  C  A  A  G  G  A  L  E
```

(continues on next page)

(continued from previous page)

```
rna3              0 ATGGGAACCGCGCTGCTTTTGCTACTGGCCGCGCTCTGCGCCGCAGGTGGGGCCCTGGAG

pro3             20
rna3             60
```

The three amino acid sequences can be aligned to each other, for example using ClustalW. Here, we create the alignment by hand:

```
>>> import numpy as np
>>> from Bio.Align import Alignment
>>> sequences = [pro1, pro2, pro3]
>>> protein_alignment = Alignment(
...     sequences, coordinates=np.array([[0, 4, 7, 23], [0, 4, 7, 23], [0, 4, 4, 20]])
... )
>>> print(protein_alignment)
pro1              0 SGTARTKLLLLLAALCAAGGALE 23
pro2              0 SGTSRTKRLLLLAALGAAGGALE 23
pro3              0 MGTA---LLLLLAALCAAGGALE 20
```

Now we can use the `mapall` method on the protein alignment, with the nucleotide-to-protein pairwise alignments as the argument, to obtain the corresponding codon alignment:

```
>>> codon_alignment = protein_alignment.mapall([alignment1, alignment2, alignment3])
>>> print(codon_alignment)
rna1              0 TCAGGGACTGCGAGAACCAAGCTA 24
rna2              0 TCAGGGACTTCGAGAACCAAGCGC 24
rna3              0 ATGGGAACCGCG---------CTG 15

rna1             24 CTGCTGCTGCTGGCTGCGCTCTGCGCCGCAGGTGGGGCGCTGGAG 69
rna2             23 CTCCTGCTGCTGGCTGCGCTCGGCGCCGCAGGTGGAGCACTGGAG 68
rna3             15 CTTTTGCTACTGGCCGCGCTCTGCGCCGCAGGTGGGGCCCTGGAG 60
```

7.12.3 Analyzing a codon alignment

Calculating the number of nonsynonymous and synonymous substitutions per site

The most important application of a codon alignment is to estimate the number of nonsynonymous substitutions per site (dN) and synonymous substitutions per site (dS). These can be calculated by the `calculate_dn_ds` function in `Bio.Align.analysis`. This function takes a pairwise codon alignment and input, as well as the optional arguments `method` specifying the calculation method, `codon_table` (defaulting to the Standard Code), the ratio `k` of the transition and transversion rates, and `cfreq` to specify the equilibrium codon frequency. Biopython currently supports three counting based methods (`NG86`, `LWL85`, `YN00`) as well as the maximum likelihood method (`ML`) to estimate dN and dS:

- `NG86`: Nei and Gojobori (1986) [Nei1986] (default). With this method, you can also specify the ratio of the transition and transversion rates via the argument `k`, defaulting to `1.0`.
- `LWL85`: Li *et al.* (1985) [Li1985].
- `YN00`: Yang and Nielsen (2000) [Yang2000].
- `ML`: Goldman and Yang (1994) [Goldman1994]. With this method, you can also specify the equilibrium codon frequency via the `cfreq` argument, with the following options:

- `F1x4`: count the nucleotide frequency in the provided codon sequences, and use it to calculate the background codon frequency;
- `F3x4`: (default) count the nucleotide frequency separately for the first, second, and third position in the provided codons, and use it to calculate the background codon frequency;
- `F61`: count the frequency of codons from the provided codon sequences, with a pseudocount of 0.1.

The `calculate_dN_dS` method can be applied to a pairwise codon alignment. In general, the different calculation methods will result in slightly different estimates for dN and dS:

```
>>> from Bio.Align import analysis
>>> pairwise_codon_alignment = codon_alignment[:2]
>>> print(pairwise_codon_alignment)
rna1              0 TCAGGGACTGCGAGAACCAAGCTA 24
                  0 |||||||||.||||||||||..
rna2              0 TCAGGGACTTCGAGAACCAAGCGC 24

rna1             24 CTGCTGCTGCTGGCTGCGCTCTGCGCCGCAGGTGGGGCGCTGGAG 69
                 24 ||.||||||||||||||||.||||||||||||.||.|||||| 69
rna2             23 CTCCTGCTGCTGGCTGCGCTCGGCGCCGCAGGTGGAGCACTGGAG 68

>>> dN, dS = analysis.calculate_dn_ds(pairwise_codon_alignment, method="NG86")
>>> print(dN, dS)
0.067715... 0.201197...
>>> dN, dS = analysis.calculate_dn_ds(pairwise_codon_alignment, method="LWL85")
>>> print(dN, dS)
0.068728... 0.207551...
```

```
>>> dN, dS = analysis.calculate_dn_ds(pairwise_codon_alignment, method="YN00")
>>> print(dN, dS)
0.081468... 0.127706...
>>> dN, dS = analysis.calculate_dn_ds(pairwise_codon_alignment, method="ML")
>>> print(dN, dS)
0.069475... 0.205754...
```

For a multiple alignment of codon sequences, you can calculate a matrix of dN and dS values:

```
>>> dN, dS = analysis.calculate_dn_ds_matrix(codon_alignment, method="NG86")
>>> print(dN)
rna1    0.000000
rna2    0.067715    0.000000
rna3    0.060204    0.145469    0.000000
    rna1    rna2    rna3
>>> print(dS)
rna1    0.000000
rna2    0.201198    0.000000
rna3    0.664268    0.798957    0.000000
    rna1    rna2    rna3
```

The objects `dN` and `dS` returned by `calculate_dn_ds_matrix` are instances of the `DistanceMatrix` class in `Bio.Phylo.TreeConstruction`. This function only takes `codon_table` as an optional argument.

From these two sequences, you can create a dN tree and a dS tree using `Bio.Phylo.TreeConstruction`:

```
>>> from Bio.Phylo.TreeConstruction import DistanceTreeConstructor
>>> dn_constructor = DistanceTreeConstructor()
>>> ds_constructor = DistanceTreeConstructor()
>>> dn_tree = dn_constructor.upgma(dN)
>>> ds_tree = ds_constructor.upgma(dS)
>>> print(type(dn_tree))
<class 'Bio.Phylo.BaseTree.Tree'>
>>> print(dn_tree)
Tree(rooted=True)
    Clade(branch_length=0, name='Inner2')
        Clade(branch_length=0.053296..., name='rna2')
        Clade(branch_length=0.023194..., name='Inner1')
            Clade(branch_length=0.0301021..., name='rna3')
            Clade(branch_length=0.0301021..., name='rna1')
>>> print(ds_tree)
Tree(rooted=True)
    Clade(branch_length=0, name='Inner2')
        Clade(branch_length=0.365806..., name='rna3')
        Clade(branch_length=0.265207..., name='Inner1')
            Clade(branch_length=0.100598..., name='rna2')
            Clade(branch_length=0.100598..., name='rna1')
```

Performing the McDonald-Kreitman test

The McDonald-Kreitman test assesses the amount of adaptive evolution by comparing the within species synonymous substitutions and nonsynonymous substitutions to the between species synonymous substitutions and nonsynonymous substitutions to see if they are from the same evolutionary process. The test requires gene sequences sampled from different individuals of the same species. In the following example, we will use Adh gene from fruit fly. The data includes 11 individuals from *Drosophila melanogaster*, 4 individuals from *Drosophila simulans*, and 12 individuals from *Drosophila yakuba*. The protein alignment data and the nucleotide sequences are available in the `Tests/codonalign` directory as the files `adh.aln` and `drosophila.fasta`, respectively, in the Biopython distribution. The function `mktest` in `Bio.Align.analysis` implements the Mcdonald-Kreitman test.

```
>>> from Bio import SeqIO
>>> from Bio import Align
>>> from Bio.Align import CodonAligner
>>> from Bio.Align.analysis import mktest
>>> aligner = CodonAligner()
>>> nucleotide_records = SeqIO.index("drosophila.fasta", "fasta")
>>> for nucleotide_record in nucleotide_records.values():
...     print(nucleotide_record.description)
...
gi|9097|emb|X57361.1| Drosophila simulans (individual c) ...
gi|9099|emb|X57362.1| Drosophila simulans (individual d) ...
gi|9101|emb|X57363.1| Drosophila simulans (individual e) ...
gi|9103|emb|X57364.1| Drosophila simulans (individual f) ...
gi|9217|emb|X57365.1| Drosophila yakuba (individual a) ...
gi|9219|emb|X57366.1| Drosophila yakuba (individual b) ...
gi|9221|emb|X57367.1| Drosophila yakuba (individual c) ...
gi|9223|emb|X57368.1| Drosophila yakuba (individual d) ...
gi|9225|emb|X57369.1| Drosophila yakuba (individual e) ...
gi|9227|emb|X57370.1| Drosophila yakuba (individual f) ...
```

(continues on next page)

(continued from previous page)

```
gi|9229|emb|X57371.1| Drosophila yakuba (individual g) ...
gi|9231|emb|X57372.1| Drosophila yakuba (individual h) ...
gi|9233|emb|X57373.1| Drosophila yakuba (individual i) ...
gi|9235|emb|X57374.1| Drosophila yakuba (individual j) ...
gi|9237|emb|X57375.1| Drosophila yakuba (individual k) ...
gi|9239|emb|X57376.1| Drosophila yakuba (individual l) ...
gi|156879|gb|M17837.1|DROADHCK D.melanogaster (strain Ja-F) ...
gi|156863|gb|M19547.1|DROADHCC D.melanogaster (strain Af-S) ...
gi|156877|gb|M17836.1|DROADHCJ D.melanogaster (strain Af-F) ...
gi|156875|gb|M17835.1|DROADHCI D.melanogaster (strain Wa-F) ...
gi|156873|gb|M17834.1|DROADHCH D.melanogaster (strain Fr-F) ...
gi|156871|gb|M17833.1|DROADHCG D.melanogaster (strain Fl-F) ...
gi|156869|gb|M17832.1|DROADHCF D.melanogaster (strain Ja-S) ...
gi|156867|gb|M17831.1|DROADHCE D.melanogaster (strain Fl-2S) ...
gi|156865|gb|M17830.1|DROADHCD D.melanogaster (strain Fr-S) ...
gi|156861|gb|M17828.1|DROADHCB D.melanogaster (strain Fl-1S) ...
gi|156859|gb|M17827.1|DROADHCA D.melanogaster (strain Wa-S) ...
>>> protein_alignment = Align.read("adh.aln", "clustal")
>>> len(protein_alignment)
27
>>> print(protein_alignment)
gi|9217|e         0 MAFTLTNKNVVFVAGLGGIGLDTSKELVKRDLKNLVILDRIENPAAIAELKAINPKVTVT
gi|9219|e         0 MAFTLTNKNVVFVAGLGGIGLDTSKELVKRDLKNLVILDRIENPAAIAELKAINPKVTVT
gi|9221|e         0 MAFTLTNKNVVFVAGLGGIGLDTSKELVKRDLKNLVILDRIENPAAIAELKAINPKVTVT
...
gi|156859         0 MSFTLTNKNVIFVAGLGGIGLDTSKELLKRDLKNLVILDRIENPAAIAELKAINPKVTVT

...

gi|9217|e       240 GTLEAIQWSKHWDSGI 256
gi|9219|e       240 GTLEAIQWSKHWDSGI 256
gi|9221|e       240 GTLEAIQWSKHWDSGI 256
...
gi|156859       240 GTLEAIQWTKHWDSGI 256

>>> codon_alignments = []
>>> for protein_record in protein_alignment.sequences:
...     nucleotide_record = nucleotide_records[protein_record.id]
...     alignments = aligner.align(protein_record, nucleotide_record)
...     assert len(alignments) == 1
...     codon_alignment = next(alignments)
...     codon_alignments.append(codon_alignment)
...
>>> print(codon_alignment)
gi|156859         0 M  S  F  T  L  T  N  K  N  V  I  F  V  A  G  L  G  G  I  G
gi|156859         0 ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTTGCCGGTCTGGGAGGCATTGGT

gi|156859        20 L  D  T  S  K  E  L  L  K  R  D  L  K  N  L  V  I  L  D  R
gi|156859        60 CTGGACACCAGCAAGGAGCTGCTCAAGCGCGATCTGAAGAACCTGGTGATCCTCGACCGC

...
```

(continues on next page)

(continued from previous page)

```
gi|156859       240 G  T  L  E  A  I  Q  W  T  K  H  W  D  S  G  I   256
gi|156859       720 GGCACCCTGGAGGCCATCCAGTGGACCAAGCACTGGGACTCCGGCATC 768

>>> nucleotide_records.close()  # Close indexed FASTA file
>>> alignment = protein_alignment.mapall(codon_alignments)
>>> print(alignment)
gi|9217|e         0 ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATTGGT
gi|9219|e         0 ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATTGGT
gi|9221|e         0 ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATTGGT
...
gi|156859         0 ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTTGCCGGTCTGGGAGGCATTGGT

...

gi|9217|e       720 GGCACCCTGGAGGCCATCCAGTGGTCCAAGCACTGGGACTCCGGCATC 768
gi|9219|e       720 GGCACCCTGGAGGCCATCCAGTGGTCCAAGCACTGGGACTCCGGCATC 768
gi|9221|e       720 GGTACCCTGGAGGCCATCCAGTGGTCCAAGCACTGGGACTCCGGCATC 768
...
gi|156859       720 GGCACCCTGGAGGCCATCCAGTGGACCAAGCACTGGGACTCCGGCATC 768

>>> unique_species = ["Drosophila simulans", "Drosophila yakuba", "D.melanogaster"]
>>> species = []
>>> for record in alignment.sequences:
...     description = record.description
...     for s in unique_species:
...         if s in description:
...             break
...     else:
...         raise Exception(f"Failed to find species for {description}")
...     species.append(s)
...
>>> print(species)
['Drosophila yakuba', 'Drosophila yakuba', 'Drosophila yakuba', 'Drosophila yakuba',
↪'Drosophila yakuba', 'Drosophila yakuba', 'Drosophila yakuba', 'Drosophila yakuba',
↪'Drosophila yakuba', 'Drosophila yakuba', 'Drosophila yakuba', 'Drosophila yakuba',
↪'Drosophila simulans', 'Drosophila simulans', 'Drosophila simulans', 'Drosophila␣
↪simulans', 'D.melanogaster', 'D.melanogaster', 'D.melanogaster', 'D.melanogaster', 'D.
↪melanogaster', 'D.melanogaster', 'D.melanogaster', 'D.melanogaster', 'D.melanogaster',
↪'D.melanogaster', 'D.melanogaster']
>>> pvalue = mktest(alignment, species)
>>> print(pvalue)
0.00206457...
```

In addition to the multiple codon alignment, the function `mktest` takes as input the species to which each sequence in the alignment belongs to. The codon table can be provided as an optional argument `codon_table`.

CHAPTER

EIGHT

MULTIPLE SEQUENCE ALIGNMENT OBJECTS

This chapter describes the older `MultipleSeqAlignment` class and the parsers in `Bio.AlignIO` that parse the output of sequence alignment software, generating `MultipleSeqAlignment` objects. By Multiple Sequence Alignments we mean a collection of multiple sequences which have been aligned together – usually with the insertion of gap characters, and addition of leading or trailing gaps – such that all the sequence strings are the same length. Such an alignment can be regarded as a matrix of letters, where each row is held as a `SeqRecord` object internally.

We will introduce the `MultipleSeqAlignment` object which holds this kind of data, and the `Bio.AlignIO` module for reading and writing them as various file formats (following the design of the `Bio.SeqIO` module from the previous chapter). Note that both `Bio.SeqIO` and `Bio.AlignIO` can read and write sequence alignment files. The appropriate choice will depend largely on what you want to do with the data.

The final part of this chapter is about using common multiple sequence alignment tools like ClustalW and MUSCLE from Python, and parsing the results with Biopython.

8.1 Parsing or Reading Sequence Alignments

We have two functions for reading in sequence alignments, `Bio.AlignIO.read()` and `Bio.AlignIO.parse()` which following the convention introduced in `Bio.SeqIO` are for files containing one or multiple alignments respectively.

Using `Bio.AlignIO.parse()` will return an *iterator* which gives `MultipleSeqAlignment` objects. Iterators are typically used in a for loop. Examples of situations where you will have multiple different alignments include resampled alignments from the PHYLIP tool `seqboot`, or multiple pairwise alignments from the EMBOSS tools `water` or `needle`, or Bill Pearson's FASTA tools.

However, in many situations you will be dealing with files which contain only a single alignment. In this case, you should use the `Bio.AlignIO.read()` function which returns a single `MultipleSeqAlignment` object.

Both functions expect two mandatory arguments:

1. The first argument is a *handle* to read the data from, typically an open file (see Section *What the heck is a handle?*), or a filename.
2. The second argument is a lower case string specifying the alignment format. As in `Bio.SeqIO` we don't try and guess the file format for you! See http://biopython.org/wiki/AlignIO for a full listing of supported formats.

There is also an optional `seq_count` argument which is discussed in Section *Ambiguous Alignments* below for dealing with ambiguous file formats which may contain more than one alignment.

8.1.1 Single Alignments

As an example, consider the following annotation rich protein alignment in the PFAM or Stockholm file format:

```
# STOCKHOLM 1.0
#=GS COATB_BPIKE/30-81  AC P03620.1
#=GS COATB_BPIKE/30-81  DR PDB; 1ifl ; 1-52;
#=GS Q9T0Q8_BPIKE/1-52  AC Q9T0Q8.1
#=GS COATB_BPI22/32-83  AC P15416.1
#=GS COATB_BPM13/24-72  AC P69541.1
#=GS COATB_BPM13/24-72  DR PDB; 2cpb ; 1-49;
#=GS COATB_BPM13/24-72  DR PDB; 2cps ; 1-49;
#=GS COATB_BPZJ2/1-49   AC P03618.1
#=GS Q9T0Q9_BPFD/1-49   AC Q9T0Q9.1
#=GS Q9T0Q9_BPFD/1-49   DR PDB; 1nh4 A; 1-49;
#=GS COATB_BPIF1/22-73  AC P03619.2
#=GS COATB_BPIF1/22-73  DR PDB; 1ifk ; 1-50;
COATB_BPIKE/30-81             AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA
#=GR COATB_BPIKE/30-81  SS    -HHHHHHHHHHHHHH--HHHHHHHH--HHHHHHHHHHHHHHHHHHHHHH----
Q9T0Q8_BPIKE/1-52             AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA
COATB_BPI22/32-83             DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA
COATB_BPM13/24-72             AEGDDP...AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
#=GR COATB_BPM13/24-72  SS    ---S-T...CHCHHHHCCCCTCCCTTCHHHHHHHHHHHHHHHHHHHHHCTT--
COATB_BPZJ2/1-49              AEGDDP...AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA
Q9T0Q9_BPFD/1-49              AEGDDP...AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
#=GR Q9T0Q9_BPFD/1-49   SS    ------...-HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH--
COATB_BPIF1/22-73              FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA
#=GR COATB_BPIF1/22-73  SS    XX-HHHH--HHHHHH--HHHHHHH--HHHHHHHHHHHHHHHHHHHHHHH---
#=GC SS_cons                  XHHHHHHHHHHHHHHHHCHHHHHHHHCHHHHHHHHHHHHHHHHHHHHHHC--
#=GC seq_cons                 AEssss...AptAhDSLpspAT-hIu.sWshVsslVsAsluIKLFKKFsSKA
//
```

This is the seed alignment for the Phage_Coat_Gp8 (PF05371) PFAM entry, downloaded from a now out of date release of PFAM from https://pfam.xfam.org/. We can load this file as follows (assuming it has been saved to disk as "PF05371_seed.sth" in the current working directory):

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
```

This code will print out a summary of the alignment:

```
>>> print(alignment)
Alignment with 7 rows and 52 columns
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRL...SKA COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKL...SRA Q9T0Q8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRL...SKA COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA Q9T0Q9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKL...SRA COATB_BPIF1/22-73
```

You'll notice in the above output the sequences have been truncated. We could instead write our own code to format this as we please by iterating over the rows as `SeqRecord` objects:

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> print("Alignment length %i" % alignment.get_alignment_length())
Alignment length 52
>>> for record in alignment:
...     print("%s - %s" % (record.seq, record.id))
...
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA - COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA - Q9T0Q8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA - COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA - COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - Q9T0Q9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA - COATB_BPIF1/22-73
```

You could also call Python's built-in `format` function on the alignment object to show it in a particular file format – see Section *Getting your alignment objects as formatted strings* for details.

Did you notice in the raw file above that several of the sequences include database cross-references to the PDB and the associated known secondary structure? Try this:

```
>>> for record in alignment:
...     if record.dbxrefs:
...         print("%s %s" % (record.id, record.dbxrefs))
...
COATB_BPIKE/30-81 ['PDB; 1ifl ; 1-52;']
COATB_BPM13/24-72 ['PDB; 2cpb ; 1-49;', 'PDB; 2cps ; 1-49;']
Q9T0Q9_BPFD/1-49 ['PDB; 1nh4 A; 1-49;']
COATB_BPIF1/22-73 ['PDB; 1ifk ; 1-50;']
```

To have a look at all the sequence annotation, try this:

```
>>> for record in alignment:
...     print(record)
...
```

PFAM provide a nice web interface at http://pfam.xfam.org/family/PF05371 which will actually let you download this alignment in several other formats. This is what the file looks like in the FASTA file format:

```
>COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA
>Q9T0Q8_BPIKE/1-52
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA
>COATB_BPI22/32-83
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA
>COATB_BPM13/24-72
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
>COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA
>Q9T0Q9_BPFD/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
>COATB_BPIF1/22-73
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA
```

Note the website should have an option about showing gaps as periods (dots) or dashes, we've shown dashes above.

Assuming you download and save this as file "PF05371_seed.faa" then you can load it with almost exactly the same code:

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.faa", "fasta")
>>> print(alignment)
```

All that has changed in this code is the filename and the format string. You'll get the same output as before, the sequences and record identifiers are the same. However, as you should expect, if you check each `SeqRecord` there is no annotation nor database cross-references because these are not included in the FASTA file format.

Note that rather than using the Sanger website, you could have used `Bio.AlignIO` to convert the original Stockholm format file into a FASTA file yourself (see below).

With any supported file format, you can load an alignment in exactly the same way just by changing the format string. For example, use "phylip" for PHYLIP files, "nexus" for NEXUS files or "emboss" for the alignments output by the EMBOSS tools. There is a full listing on the wiki page (http://biopython.org/wiki/AlignIO) and in the built-in documentation, *Bio.AlignIO*:

```
>>> from Bio import AlignIO
>>> help(AlignIO)
```

8.1.2 Multiple Alignments

The previous section focused on reading files containing a single alignment. In general however, files can contain more than one alignment, and to read these files we must use the `Bio.AlignIO.parse()` function.

Suppose you have a small alignment in PHYLIP format:

```
    5    6
Alpha     AACAAC
Beta      AACCCC
Gamma     ACCAAC
Delta     CCACCA
Epsilon   CCAAAC
```

If you wanted to bootstrap a phylogenetic tree using the PHYLIP tools, one of the steps would be to create a set of many resampled alignments using the tool `bootseq`. This would give output something like this, which has been abbreviated for conciseness:

```
    5     6
Alpha     AAACCA
Beta      AAACCC
Gamma     ACCCCA
Delta     CCCAAC
Epsilon   CCCAAA
    5     6
Alpha     AAACAA
Beta      AAACCC
Gamma     ACCCAA
Delta     CCCACC
Epsilon   CCCAAA
    5     6
Alpha     AAAAAC
```

(continues on next page)

(continued from previous page)

```
Beta      AAACCC
Gamma     AACAAC
Delta     CCCCCA
Epsilon   CCCAAC
...
    5     6
Alpha     AAAACC
Beta      ACCCCC
Gamma     AAAACC
Delta     CCCCAA
Epsilon   CAAACC
```

If you wanted to read this in using `Bio.AlignIO` you could use:

```
>>> from Bio import AlignIO
>>> alignments = AlignIO.parse("resampled.phy", "phylip")
>>> for alignment in alignments:
...     print(alignment)
...     print()
...
```

This would give the following output, again abbreviated for display:

```
Alignment with 5 rows and 6 columns
AAACCA Alpha
AAACCC Beta
ACCCCA Gamma
CCCAAC Delta
CCCAAA Epsilon

Alignment with 5 rows and 6 columns
AAACAA Alpha
AAACCC Beta
ACCCAA Gamma
CCCACC Delta
CCCAAA Epsilon

Alignment with 5 rows and 6 columns
AAAAAC Alpha
AAACCC Beta
AACAAC Gamma
CCCCCA Delta
CCCAAC Epsilon

...

Alignment with 5 rows and 6 columns
AAAACC Alpha
ACCCCC Beta
AAAACC Gamma
CCCCAA Delta
CAAACC Epsilon
```

As with the function `Bio.SeqIO.parse()`, using `Bio.AlignIO.parse()` returns an iterator. If you want to keep all the alignments in memory at once, which will allow you to access them in any order, then turn the iterator into a list:

```
>>> from Bio import AlignIO
>>> alignments = list(AlignIO.parse("resampled.phy", "phylip"))
>>> last_align = alignments[-1]
>>> first_align = alignments[0]
```

8.1.3 Ambiguous Alignments

Many alignment file formats can explicitly store more than one alignment, and the division between each alignment is clear. However, when a general sequence file format has been used there is no such block structure. The most common such situation is when alignments have been saved in the FASTA file format. For example consider the following:

```
>Alpha
ACTACGACTAGCTCAG--G
>Beta
ACTACCGCTAGCTCAGAAG
>Gamma
ACTACGGCTAGCACAGAAG
>Alpha
ACTACGACTAGCTCAGG--
>Beta
ACTACCGCTAGCTCAGAAG
>Gamma
ACTACGGCTAGCACAGAAG
```

This could be a single alignment containing six sequences (with repeated identifiers). Or, judging from the identifiers, this is probably two different alignments each with three sequences, which happen to all have the same length.

What about this next example?

```
>Alpha
ACTACGACTAGCTCAG--G
>Beta
ACTACCGCTAGCTCAGAAG
>Alpha
ACTACGACTAGCTCAGG--
>Gamma
ACTACGGCTAGCACAGAAG
>Alpha
ACTACGACTAGCTCAGG--
>Delta
ACTACGGCTAGCACAGAAG
```

Again, this could be a single alignment with six sequences. However this time based on the identifiers we might guess this is three pairwise alignments which by chance have all got the same lengths.

This final example is similar:

```
>Alpha
ACTACGACTAGCTCAG--G
>XXX
ACTACCGCTAGCTCAGAAG
```

(continues on next page)

(continued from previous page)

```
>Alpha
ACTACGACTAGCTCAGG
>YYY
ACTACGGCAAGCACAGG
>Alpha
--ACTACGAC--TAGCTCAGG
>ZZZ
GGACTACGACAATAGCTCAGG
```

In this third example, because of the differing lengths, this cannot be treated as a single alignment containing all six records. However, it could be three pairwise alignments.

Clearly trying to store more than one alignment in a FASTA file is not ideal. However, if you are forced to deal with these as input files `Bio.AlignIO` can cope with the most common situation where all the alignments have the same number of records. One example of this is a collection of pairwise alignments, which can be produced by the EMBOSS tools `needle` and `water` – although in this situation, `Bio.AlignIO` should be able to understand their native output using "emboss" as the format string.

To interpret these FASTA examples as several separate alignments, we can use `Bio.AlignIO.parse()` with the optional `seq_count` argument which specifies how many sequences are expected in each alignment (in these examples, 3, 2 and 2 respectively). For example, using the third example as the input data:

```
>>> for alignment in AlignIO.parse(handle, "fasta", seq_count=2):
...     print("Alignment length %i" % alignment.get_alignment_length())
...     for record in alignment:
...         print("%s - %s" % (record.seq, record.id))
...     print()
...
```

giving:

```
Alignment length 19
ACTACGACTAGCTCAG--G - Alpha
ACTACCGCTAGCTCAGAAG - XXX

Alignment length 17
ACTACGACTAGCTCAGG - Alpha
ACTACGGCAAGCACAGG - YYY

Alignment length 21
--ACTACGAC--TAGCTCAGG - Alpha
GGACTACGACAATAGCTCAGG - ZZZ
```

Using `Bio.AlignIO.read()` or `Bio.AlignIO.parse()` without the `seq_count` argument would give a single alignment containing all six records for the first two examples. For the third example, an exception would be raised because the lengths differ preventing them being turned into a single alignment.

If the file format itself has a block structure allowing `Bio.AlignIO` to determine the number of sequences in each alignment directly, then the `seq_count` argument is not needed. If it is supplied, and doesn't agree with the file contents, an error is raised.

Note that this optional `seq_count` argument assumes each alignment in the file has the same number of sequences. Hypothetically you may come across stranger situations, for example a FASTA file containing several alignments each with a different number of sequences – although I would love to hear of a real world example of this. Assuming you cannot get the data in a nicer file format, there is no straight forward way to deal with this using `Bio.AlignIO`. In this

case, you could consider reading in the sequences themselves using `Bio.SeqIO` and batching them together to create the alignments as appropriate.

8.2 Writing Alignments

We've talked about using `Bio.AlignIO.read()` and `Bio.AlignIO.parse()` for alignment input (reading files), and now we'll look at `Bio.AlignIO.write()` which is for alignment output (writing files). This is a function taking three arguments: some `MultipleSeqAlignment` objects (or for backwards compatibility the obsolete `Alignment` objects), a handle or filename to write to, and a sequence format.

Here is an example, where we start by creating a few `MultipleSeqAlignment` objects the hard way (by hand, rather than by loading them from a file). Note we create some `SeqRecord` objects to construct the alignment from.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> align1 = MultipleSeqAlignment(
...     [
...         SeqRecord(Seq("ACTGCTAGCTAG"), id="Alpha"),
...         SeqRecord(Seq("ACT-CTAGCTAG"), id="Beta"),
...         SeqRecord(Seq("ACTGCTAGDTAG"), id="Gamma"),
...     ]
... )
>>> align2 = MultipleSeqAlignment(
...     [
...         SeqRecord(Seq("GTCAGC-AG"), id="Delta"),
...         SeqRecord(Seq("GACAGCTAG"), id="Epsilon"),
...         SeqRecord(Seq("GTCAGCTAG"), id="Zeta"),
...     ]
... )
>>> align3 = MultipleSeqAlignment(
...     [
...         SeqRecord(Seq("ACTAGTACAGCTG"), id="Eta"),
...         SeqRecord(Seq("ACTAGTACAGCT-"), id="Theta"),
...         SeqRecord(Seq("-CTACTACAGGTG"), id="Iota"),
...     ]
... )
>>> my_alignments = [align1, align2, align3]
```

Now we have a list of `Alignment` objects, we'll write them to a PHYLIP format file:

```
>>> from Bio import AlignIO
>>> AlignIO.write(my_alignments, "my_example.phy", "phylip")
```

And if you open this file in your favorite text editor it should look like this:

```
 3 12
Alpha      ACTGCTAGCT AG
Beta       ACT-CTAGCT AG
Gamma      ACTGCTAGDT AG
 3 9
Delta      GTCAGC-AG
Epislon    GACAGCTAG
```

(continues on next page)

(continued from previous page)

```
Zeta       GTCAGCTAG
 3 13
Eta        ACTAGTACAG CTG
Theta      ACTAGTACAG CT-
Iota       -CTACTACAG GTG
```

Its more common to want to load an existing alignment, and save that, perhaps after some simple manipulation like removing certain rows or columns.

Suppose you wanted to know how many alignments the `Bio.AlignIO.write()` function wrote to the handle? If your alignments were in a list like the example above, you could just use `len(my_alignments)`, however you can't do that when your records come from a generator/iterator. Therefore the `Bio.AlignIO.write()` function returns the number of alignments written to the file.

Note - If you tell the `Bio.AlignIO.write()` function to write to a file that already exists, the old file will be overwritten without any warning.

8.2.1 Converting between sequence alignment file formats

Converting between sequence alignment file formats with `Bio.AlignIO` works in the same way as converting between sequence file formats with `Bio.SeqIO` (Section *Converting between sequence file formats*). We load generally the alignment(s) using `Bio.AlignIO.parse()` and then save them using the `Bio.AlignIO.write()` – or just use the `Bio.AlignIO.convert()` helper function.

For this example, we'll load the PFAM/Stockholm format file used earlier and save it as a Clustal W format file:

```
>>> from Bio import AlignIO
>>> count = AlignIO.convert("PF05371_seed.sth", "stockholm", "PF05371_seed.aln", "clustal
↪")
>>> print("Converted %i alignments" % count)
Converted 1 alignments
```

Or, using `Bio.AlignIO.parse()` and `Bio.AlignIO.write()`:

```
>>> from Bio import AlignIO
>>> alignments = AlignIO.parse("PF05371_seed.sth", "stockholm")
>>> count = AlignIO.write(alignments, "PF05371_seed.aln", "clustal")
>>> print("Converted %i alignments" % count)
Converted 1 alignments
```

The `Bio.AlignIO.write()` function expects to be given multiple alignment objects. In the example above we gave it the alignment iterator returned by `Bio.AlignIO.parse()`.

In this case, we know there is only one alignment in the file so we could have used `Bio.AlignIO.read()` instead, but notice we have to pass this alignment to `Bio.AlignIO.write()` as a single element list:

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> AlignIO.write([alignment], "PF05371_seed.aln", "clustal")
```

Either way, you should end up with the same new Clustal W format file "PF05371_seed.aln" with the following content:

```
CLUSTAL X (1.81) multiple sequence alignment
```

(continues on next page)

(continued from previous page)

```
COATB_BPIKE/30-81                AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSS
Q9T0Q8_BPIKE/1-52                AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVS
COATB_BPI22/32-83                DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSS
COATB_BPM13/24-72                AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTS
COATB_BPZJ2/1-49                 AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFAS
Q9T0Q9_BPFD/1-49                 AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTS
COATB_BPIF1/22-73                FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVS

COATB_BPIKE/30-81                KA
Q9T0Q8_BPIKE/1-52                RA
COATB_BPI22/32-83                KA
COATB_BPM13/24-72                KA
COATB_BPZJ2/1-49                 KA
Q9T0Q9_BPFD/1-49                 KA
COATB_BPIF1/22-73                RA
```

Alternatively, you could make a PHYLIP format file which we'll name "PF05371_seed.phy":

```
>>> from Bio import AlignIO
>>> AlignIO.convert("PF05371_seed.sth", "stockholm", "PF05371_seed.phy", "phylip")
```

This time the output looks like this:

```
 7 52
COATB_BPIK AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIRLFKKFSS
Q9T0Q8_BPI AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIKLFKKFVS
COATB_BPI2 DGTSTATSYA TEAMNSLKTQ ATDLIDQTWP VVTSVAVAGL AIRLFKKFSS
COATB_BPM1 AEGDDP---A KAAFNSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPZJ AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFAS
Q9T0Q9_BPF AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPIF FAADDATSQA KAAFDSLTAQ ATEMSGYAWA LVVLVVGATV GIKLFKKFVS

           KA
           RA
           KA
           KA
           KA
           KA
           RA
```

One of the big handicaps of the original PHYLIP alignment file format is that the sequence identifiers are strictly truncated at ten characters. In this example, as you can see the resulting names are still unique - but they are not very readable. As a result, a more relaxed variant of the original PHYLIP format is now quite widely used:

```
>>> from Bio import AlignIO
>>> AlignIO.convert("PF05371_seed.sth", "stockholm", "PF05371_seed.phy", "phylip-relaxed
↪")
```

This time the output looks like this, using a longer indentation to allow all the identifiers to be given in full:

```
 7 52
COATB_BPIKE/30-81  AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIRLFKKFSS
```

(continues on next page)

(continued from previous page)

```
Q9T0Q8_BPIKE/1-52  AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIKLFKKFVS
COATB_BPI22/32-83  DGTSTATSYA TEAMNSLKTQ ATDLIDQTWP VVTSVAVAGL AIRLFKKFSS
COATB_BPM13/24-72  AEGDDP---A KAAFNSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPZJ2/1-49   AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFAS
Q9T0Q9_BPFD/1-49   AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPIF1/22-73  FAADDATSQA KAAFDSLTAQ ATEMSGYAWA LVVLVVGATV GIKLFKKFVS

                   KA
                   RA
                   KA
                   KA
                   KA
                   KA
                   RA
```

If you have to work with the original strict PHYLIP format, then you may need to compress the identifiers somehow – or assign your own names or numbering system. This following bit of code manipulates the record identifiers before saving the output:

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> name_mapping = {}
>>> for i, record in enumerate(alignment):
...     name_mapping[i] = record.id
...     record.id = "seq%i" % i
...
>>> print(name_mapping)
{0: 'COATB_BPIKE/30-81', 1: 'Q9T0Q8_BPIKE/1-52', 2: 'COATB_BPI22/32-83', 3: 'COATB_BPM13/
↪24-72', 4: 'COATB_BPZJ2/1-49', 5: 'Q9T0Q9_BPFD/1-49', 6: 'COATB_BPIF1/22-73'}
>>> AlignIO.write([alignment], "PF05371_seed.phy", "phylip")
```

This code used a Python dictionary to record a simple mapping from the new sequence system to the original identifier:

```
{
    0: "COATB_BPIKE/30-81",
    1: "Q9T0Q8_BPIKE/1-52",
    2: "COATB_BPI22/32-83",
    # ...
}
```

Here is the new (strict) PHYLIP format output:

```
 7 52
seq0       AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIRLFKKFSS
seq1       AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIKLFKKFVS
seq2       DGTSTATSYA TEAMNSLKTQ ATDLIDQTWP VVTSVAVAGL AIRLFKKFSS
seq3       AEGDDP---A KAAFNSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
seq4       AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFAS
seq5       AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
seq6       FAADDATSQA KAAFDSLTAQ ATEMSGYAWA LVVLVVGATV GIKLFKKFVS

           KA
           RA
```

(continues on next page)

(continued from previous page)

```
          KA
          KA
          KA
          KA
          RA
```

In general, because of the identifier limitation, working with *strict* PHYLIP file formats shouldn't be your first choice. Using the PFAM/Stockholm format on the other hand allows you to record a lot of additional annotation too.

8.2.2 Getting your alignment objects as formatted strings

The `Bio.AlignIO` interface is based on handles, which means if you want to get your alignment(s) into a string in a particular file format you need to do a little bit more work (see below). However, you will probably prefer to call Python's built-in `format` function on the alignment object. This takes an output format specification as a single argument, a lower case string which is supported by `Bio.AlignIO` as an output format. For example:

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> print(format(alignment, "clustal"))
CLUSTAL X (1.81) multiple sequence alignment


COATB_BPIKE/30-81                     AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSS
Q9T0Q8_BPIKE/1-52                     AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVS
COATB_BPI22/32-83                     DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSS
...
```

Without an output format specification, `format` returns the same output as `str`.

As described in Section *The format method*, the `SeqRecord` object has a similar method using output formats supported by `Bio.SeqIO`.

Internally `format` is calling `Bio.AlignIO.write()` with a `StringIO` handle. You can do this in your own code if for example you are using an older version of Biopython:

```
>>> from io import StringIO
>>> from Bio import AlignIO
>>> alignments = AlignIO.parse("PF05371_seed.sth", "stockholm")
>>> out_handle = StringIO()
>>> AlignIO.write(alignments, out_handle, "clustal")
1
>>> clustal_data = out_handle.getvalue()
>>> print(clustal_data)
CLUSTAL X (1.81) multiple sequence alignment


COATB_BPIKE/30-81                     AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSS
Q9T0Q8_BPIKE/1-52                     AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVS
COATB_BPI22/32-83                     DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSS
COATB_BPM13/24-72                     AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTS
...
```

8.3 Manipulating Alignments

Now that we've covered loading and saving alignments, we'll look at what else you can do with them.

8.3.1 Slicing alignments

First of all, in some senses the alignment objects act like a Python `list` of `SeqRecord` objects (the rows). With this model in mind hopefully the actions of `len()` (the number of rows) and iteration (each row as a `SeqRecord`) make sense:

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> print("Number of rows: %i" % len(alignment))
Number of rows: 7
>>> for record in alignment:
...     print("%s - %s" % (record.seq, record.id))
...
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA - COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA - Q9T0Q8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA - COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA - COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - Q9T0Q9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA - COATB_BPIF1/22-73
```

You can also use the list-like `append` and `extend` methods to add more rows to the alignment (as `SeqRecord` objects). Keeping the list metaphor in mind, simple slicing of the alignment should also make sense - it selects some of the rows giving back another alignment object:

```
>>> print(alignment)
Alignment with 7 rows and 52 columns
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRL...SKA COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKL...SRA Q9T0Q8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRL...SKA COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA Q9T0Q9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKL...SRA COATB_BPIF1/22-73
>>> print(alignment[3:7])
Alignment with 4 rows and 52 columns
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA Q9T0Q9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKL...SRA COATB_BPIF1/22-73
```

What if you wanted to select by column? Those of you who have used the NumPy matrix or array objects won't be surprised at this - you use a double index.

```
>>> print(alignment[2, 6])
T
```

Using two integer indices pulls out a single letter, short hand for this:

```
>>> print(alignment[2].seq[6])
T
```

You can pull out a single column as a string like this:

```
>>> print(alignment[:, 6])
TTT---T
```

You can also select a range of columns. For example, to pick out those same three rows we extracted earlier, but take just their first six columns:

```
>>> print(alignment[3:6, :6])
Alignment with 3 rows and 6 columns
AEGDDP COATB_BPM13/24-72
AEGDDP COATB_BPZJ2/1-49
AEGDDP Q9T0Q9_BPFD/1-49
```

Leaving the first index as `:` means take all the rows:

```
>>> print(alignment[:, :6])
Alignment with 7 rows and 6 columns
AEPNAA COATB_BPIKE/30-81
AEPNAA Q9T0Q8_BPIKE/1-52
DGTSTA COATB_BPI22/32-83
AEGDDP COATB_BPM13/24-72
AEGDDP COATB_BPZJ2/1-49
AEGDDP Q9T0Q9_BPFD/1-49
FAADDA COATB_BPIF1/22-73
```

This brings us to a neat way to remove a section. Notice columns 7, 8 and 9 which are gaps in three of the seven sequences:

```
>>> print(alignment[:, 6:9])
Alignment with 7 rows and 3 columns
TNY COATB_BPIKE/30-81
TNY Q9T0Q8_BPIKE/1-52
TSY COATB_BPI22/32-83
--- COATB_BPM13/24-72
--- COATB_BPZJ2/1-49
--- Q9T0Q9_BPFD/1-49
TSQ COATB_BPIF1/22-73
```

Again, you can slice to get everything after the ninth column:

```
>>> print(alignment[:, 9:])
Alignment with 7 rows and 43 columns
ATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA COATB_BPIKE/30-81
ATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA Q9T0Q8_BPIKE/1-52
ATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA COATB_BPI22/32-83
AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA COATB_BPM13/24-72
AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA COATB_BPZJ2/1-49
AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA Q9T0Q9_BPFD/1-49
AKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA COATB_BPIF1/22-73
```

Now, the interesting thing is that addition of alignment objects works by column. This lets you do this as a way to remove a block of columns:

```
>>> edited = alignment[:, :6] + alignment[:, 9:]
>>> print(edited)
Alignment with 7 rows and 49 columns
AEPNAAATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA COATB_BPIKE/30-81
AEPNAAATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA Q9T0Q8_BPIKE/1-52
DGTSTAATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA COATB_BPI22/32-83
AEGDDPAKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA COATB_BPM13/24-72
AEGDDPAKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA COATB_BPZJ2/1-49
AEGDDPAKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA Q9T0Q9_BPFD/1-49
FAADDAAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA COATB_BPIF1/22-73
```

Another common use of alignment addition would be to combine alignments for several different genes into a meta-alignment. Watch out though - the identifiers need to match up (see Section *Adding SeqRecord objects* for how adding `SeqRecord` objects works). You may find it helpful to first sort the alignment rows alphabetically by id:

```
>>> edited.sort()
>>> print(edited)
Alignment with 7 rows and 49 columns
DGTSTAATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA COATB_BPI22/32-83
FAADDAAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA COATB_BPIF1/22-73
AEPNAAATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA COATB_BPIKE/30-81
AEGDDPAKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA COATB_BPM13/24-72
AEGDDPAKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA COATB_BPZJ2/1-49
AEPNAAATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA Q9T0Q8_BPIKE/1-52
AEGDDPAKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA Q9T0Q9_BPFD/1-49
```

Note that you can only add two alignments together if they have the same number of rows.

8.3.2 Alignments as arrays

Depending on what you are doing, it can be more useful to turn the alignment object into an array of letters – and you can do this with NumPy:

```
>>> import numpy as np
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> align_array = np.array(alignment)
>>> print("Array shape %i by %i" % align_array.shape)
Array shape 7 by 52
>>> align_array[:, :10]
array([['A', 'E', 'P', 'N', 'A', 'A', 'T', 'N', 'Y', 'A'],
       ['A', 'E', 'P', 'N', 'A', 'A', 'T', 'N', 'Y', 'A'],
       ['D', 'G', 'T', 'S', 'T', 'A', 'T', 'S', 'Y', 'A'],
       ['A', 'E', 'G', 'D', 'D', 'P', '-', '-', '-', 'A'],
       ['A', 'E', 'G', 'D', 'D', 'P', '-', '-', '-', 'A'],
       ['A', 'E', 'G', 'D', 'D', 'P', '-', '-', '-', 'A'],
       ['F', 'A', 'A', 'D', 'D', 'A', 'T', 'S', 'Q', 'A']],...
```

Note that this leaves the original Biopython alignment object and the NumPy array in memory as separate objects - editing one will not update the other!

8.3.3 Counting substitutions

The `substitutions` property of an alignment reports how often letters in the alignment are substituted for each other. This is calculated by taking all pairs of rows in the alignment, counting the number of times two letters are aligned to each other, and summing this over all pairs. For example,

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> msa = MultipleSeqAlignment(
...     [
...         SeqRecord(Seq("ACTCCTA"), id="seq1"),
...         SeqRecord(Seq("AAT-CTA"), id="seq2"),
...         SeqRecord(Seq("CCTACT-"), id="seq3"),
...         SeqRecord(Seq("TCTCCTC"), id="seq4"),
...     ]
... )
>>> print(msa)
Alignment with 4 rows and 7 columns
ACTCCTA seq1
AAT-CTA seq2
CCTACT- seq3
TCTCCTC seq4
>>> substitutions = msa.substitutions
>>> print(substitutions)
    A    C    T
A 2.0  4.5  1.0
C 4.5 10.0  0.5
T 1.0  0.5 12.0
```

As the ordering of pairs is arbitrary, counts are divided equally above and below the diagonal. For example, the 9 alignments of `A` to `C` are stored as 4.5 at position `['A', 'C']` and 4.5 at position `['C', 'A']`. This arrangement helps to make the math easier when calculating a substitution matrix from these counts, as described in Section *Substitution matrices*.

Note that `msa.substitutions` contains entries for the letters appearing in the alignment only. You can use the `select` method to add entries for missing letters, for example

```
>>> m = substitutions.select("ATCG")
>>> print(m)
    A    T    C   G
A 2.0  1.0  4.5 0.0
T 1.0 12.0  0.5 0.0
C 4.5  0.5 10.0 0.0
G 0.0  0.0  0.0 0.0
```

This also allows you to change the order of letters in the alphabet.

8.3.4 Calculating summary information

Once you have an alignment, you are very likely going to want to find out information about it. Instead of trying to have all of the functions that can generate information about an alignment in the alignment object itself, we've tried to separate out the functionality into separate classes, which act on the alignment.

Getting ready to calculate summary information about an object is quick to do. Let's say we've got an alignment object called `alignment`, for example read in using `Bio.AlignIO.read(...)` as described in Chapter *Multiple Sequence Alignment objects*. All we need to do to get an object that will calculate summary information is:

```
>>> from Bio.Align import AlignInfo
>>> summary_align = AlignInfo.SummaryInfo(msa)
```

The `summary_align` object is very useful, and will do the following neat things for you:

1. Calculate a quick consensus sequence – see section *Calculating a quick consensus sequence*
2. Get a position specific score matrix for the alignment – see section *Position Specific Score Matrices*
3. Calculate the information content for the alignment – see section *Information Content*
4. Generate information on substitutions in the alignment – section *Substitution matrices* details using this to generate a substitution matrix.

8.3.5 Calculating a quick consensus sequence

The `SummaryInfo` object, described in section *Calculating summary information*, provides functionality to calculate a quick consensus of an alignment. Assuming we've got a `SummaryInfo` object called `summary_align` we can calculate a consensus by doing:

```
>>> consensus = summary_align.dumb_consensus()
>>> consensus
Seq('XCTXCTX')
```

As the name suggests, this is a really simple consensus calculator, and will just add up all of the residues at each point in the consensus, and if the most common value is higher than some threshold value will add the common residue to the consensus. If it doesn't reach the threshold, it adds an ambiguity character to the consensus. The returned consensus object is a `Seq` object.

You can adjust how `dumb_consensus` works by passing optional parameters:

the threshold
: This is the threshold specifying how common a particular residue has to be at a position before it is added. The default is 0.7 (meaning 70%).

the ambiguous character
: This is the ambiguity character to use. The default is 'N'.

Alternatively, you can convert the multiple sequence alignment object `msa` to a new-style `Alignment` object (see section *Alignment objects*) by using the `alignment` attribute (see section *Getting a new-style Alignment object*):

```
>>> alignment = msa.alignment
```

You can then create a `Motif` object (see section *Motif objects*):

```
>>> from Bio.motifs import Motif
>>> motif = Motif("ACGT", alignment)
```

and obtain a quick consensus sequence:

```
>>> motif.consensus
Seq('ACTCCTA')
```

The `motif.counts.calculate_consensus` method (see section *Obtaining a consensus sequence*) lets you specify in detail how the consensus sequence should be calculated. For example,

```
>>> motif.counts.calculate_consensus(identity=0.7)
'NCTNCTN'
```

8.3.6 Position Specific Score Matrices

Position specific score matrices (PSSMs) summarize the alignment information in a different way than a consensus, and may be useful for different tasks. Basically, a PSSM is a count matrix. For each column in the alignment, the number of each alphabet letters is counted and totaled. The totals are displayed relative to some representative sequence along the left axis. This sequence may be the consensus sequence, but can also be any sequence in the alignment.

For instance for the alignment above:

```
>>> print(msa)
Alignment with 4 rows and 7 columns
ACTCCTA seq1
AAT-CTA seq2
CCTACT- seq3
TCTCCTC seq4
```

we get a PSSM with the consensus sequence along the side using

```
>>> my_pssm = summary_align.pos_specific_score_matrix(consensus, chars_to_ignore=["N"])
>>> print(my_pssm)
    A   C   T
X  2.0 1.0 1.0
C  1.0 3.0 0.0
T  0.0 0.0 4.0
X  1.0 2.0 0.0
C  0.0 4.0 0.0
T  0.0 0.0 4.0
X  2.0 1.0 0.0
```

where we ignore any `N` ambiguity residues when calculating the PSSM.

Two notes should be made about this:

1. To maintain strictness with the alphabets, you can only include characters along the top of the PSSM that are in the alphabet of the alignment object. Gaps are not included along the top axis of the PSSM.
2. The sequence passed to be displayed along the left side of the axis does not need to be the consensus. For instance, if you wanted to display the second sequence in the alignment along this axis, you would need to do:

```
>>> second_seq = msa[1]
>>> my_pssm = summary_align.pos_specific_score_matrix(second_seq, chars_to_ignore=[
↪"N"])
>>> print(my_pssm)
    A   C   T
```

(continues on next page)

(continued from previous page)

```
A  2.0 1.0 1.0
A  1.0 3.0 0.0
T  0.0 0.0 4.0
-  1.0 2.0 0.0
C  0.0 4.0 0.0
T  0.0 0.0 4.0
A  2.0 1.0 0.0
```

The command above returns a `PSSM` object. You can access any element of the PSSM by subscripting like `your_pssm[sequence_number][residue_count_name]`. For instance, to get the counts for the 'A' residue in the second element of the above PSSM you would do:

```
>>> print(my_pssm[5]["T"])
4.0
```

The structure of the PSSM class hopefully makes it easy both to access elements and to pretty print the matrix.

Alternatively, you can convert the multiple sequence alignment object `msa` to a new-style `Alignment` object (see section *Alignment objects*) by using the `alignment` attribute (see section *Getting a new-style Alignment object*):

```
>>> alignment = msa.alignment
```

You can then create a `Motif` object (see section *Motif objects*):

```
>>> from Bio.motifs import Motif
>>> motif = Motif("ACGT", alignment)
```

and obtain the counts of each nucleotide in each position:

```
>>> counts = motif.counts
>>> print(counts)
        0      1      2      3      4      5      6
A:   2.00   1.00   0.00   1.00   0.00   0.00   2.00
C:   1.00   3.00   0.00   2.00   4.00   0.00   1.00
G:   0.00   0.00   0.00   0.00   0.00   0.00   0.00
T:   1.00   0.00   4.00   0.00   0.00   4.00   0.00

>>> print(counts["T"][5])
4.0
```

8.3.7 Information Content

A potentially useful measure of evolutionary conservation is the information content of a sequence.

A useful introduction to information theory targeted towards molecular biologists can be found at http://www.lecb.ncifcrf.gov/~toms/paper/primer/. For our purposes, we will be looking at the information content of a consensus sequence, or a portion of a consensus sequence. We calculate information content at a particular column in a multiple sequence alignment using the following formula:

$$IC_j = \sum_{i=1}^{N_a} P_{ij} \log\left(\frac{P_{ij}}{Q_i}\right)$$

where:

- IC_j – The information content for the j-th column in an alignment.
- N_a – The number of letters in the alphabet.
- P_{ij} – The frequency of a particular letter i in the j-th column (i. e. if G occurred 3 out of 6 times in an alignment column, this would be 0.5)
- Q_i – The expected frequency of a letter i. This is an optional argument, usage of which is left at the user's discretion. By default, it is automatically assigned to $0.05 = 1/20$ for a protein alphabet, and $0.25 = 1/4$ for a nucleic acid alphabet. This is for getting the information content without any assumption of prior distributions. When assuming priors, or when using a non-standard alphabet, you should supply the values for Q_i.

Well, now that we have an idea what information content is being calculated in Biopython, let's look at how to get it for a particular region of the alignment.

First, we need to use our alignment to get an alignment summary object, which we'll assume is called `summary_align` (see section *Calculating summary information*) for instructions on how to get this. Once we've got this object, calculating the information content for a region is as easy as:

```
>>> e_freq_table = {"A": 0.3, "G": 0.2, "T": 0.3, "C": 0.2}
>>> info_content = summary_align.information_content(
...     2, 6, e_freq_table=e_freq_table, chars_to_ignore=["N"]
... )
>>> info_content
6.3910647...
```

Now, `info_content` will contain the relative information content over the region [2:6] in relation to the expected frequencies.

The value return is calculated using base 2 as the logarithm base in the formula above. You can modify this by passing the parameter `log_base` as the base you want:

```
>>> info_content = summary_align.information_content(
...     2, 6, e_freq_table=e_freq_table, log_base=10, chars_to_ignore=["N"]
... )
>>> info_content
1.923902...
```

By default nucleotide or amino acid residues with a frequency of 0 in a column are not take into account when the relative information column for that column is computed. If this is not the desired result, you can use `pseudo_count` instead.

```
>>> info_content = summary_align.information_content(
...     2, 6, e_freq_table=e_freq_table, chars_to_ignore=["N", "-"], pseudo_count=1
... )
>>> info_content
4.299651...
```

In this case, the observed frequency P_{ij} of a particular letter i in the j-th column is computed as follows:

$$P_{ij} = \frac{n_{ij} + k \times Q_i}{N_j + k}$$

where:

- k – the pseudo count you pass as argument.
- k – the pseudo count you pass as argument.
- Q_i – The expected frequency of the letter i as described above.

Well, now you are ready to calculate information content. If you want to try applying this to some real life problems, it would probably be best to dig into the literature on information content to get an idea of how it is used. Hopefully your digging won't reveal any mistakes made in coding this function!

8.4 Getting a new-style Alignment object

Use the `alignment` property to create a new-style `Alignment` object (see section *Alignment objects*) from an old-style `MultipleSeqAlignment` object:

```
>>> type(msa)
<class 'Bio.Align.MultipleSeqAlignment'>
>>> print(msa)
Alignment with 4 rows and 7 columns
ACTCCTA seq1
AAT-CTA seq2
CCTACT- seq3
TCTCCTC seq4
>>> alignment = msa.alignment
>>> type(alignment)
<class 'Bio.Align.Alignment'>
>>> print(alignment)
seq1              0 ACTCCTA 7
seq2              0 AAT-CTA 6
seq3              0 CCTACT- 6
seq4              0 TCTCCTC 7
```

Note that the `alignment` property creates and returns a new `Alignment` object that is consistent with the information stored in the `MultipleSeqAlignment` object at the time the `Alignment` object is created. Any changes to the `MultipleSeqAlignment` after calling the `alignment` property will not propagate to the `Alignment` object. However, you can of course call the `alignment` property again to create a new `Alignment` object consistent with the updated `MultipleSeqAlignment` object.

8.5 Calculating a substitution matrix from a multiple sequence alignment

You can create your own substitution matrix from an alignment. In this example, we'll first read a protein sequence alignment from the Clustalw file protein.aln (also available online here)

```
>>> from Bio import AlignIO
>>> filename = "protein.aln"
>>> msa = AlignIO.read(filename, "clustal")
```

Section *ClustalW* contains more information on doing this.

The `substitutions` property of the alignment stores the number of times different residues substitute for each other:

```
>>> observed_frequencies = msa.substitutions
```

To make the example more readable, we'll select only amino acids with polar charged side chains:

```
>>> observed_frequencies = observed_frequencies.select("DEHKR")
>>> print(observed_frequencies)
       D      E      H      K      R
D 2360.0  255.5    7.5    0.5   25.0
E  255.5 3305.0   16.5   27.0    2.0
H    7.5   16.5 1235.0   16.0    8.5
K    0.5   27.0   16.0 3218.0  116.5
R   25.0    2.0    8.5  116.5 2079.0
```

Rows and columns for other amino acids were removed from the matrix.

Next, we normalize the matrix:

```
>>> import numpy as np
>>> observed_frequencies /= np.sum(observed_frequencies)
```

Summing over rows or columns gives the relative frequency of occurrence of each residue:

```
>>> residue_frequencies = np.sum(observed_frequencies, 0)
>>> print(residue_frequencies.format("%.4f"))
D 0.2015
E 0.2743
H 0.0976
K 0.2569
R 0.1697

>>> sum(residue_frequencies) == 1.0
True
```

The expected frequency of residue pairs is then

```
>>> expected_frequencies = np.dot(
...     residue_frequencies[:, None], residue_frequencies[None, :]
... )
>>> print(expected_frequencies.format("%.4f"))
       D      E      H      K      R
D 0.0406 0.0553 0.0197 0.0518 0.0342
E 0.0553 0.0752 0.0268 0.0705 0.0465
H 0.0197 0.0268 0.0095 0.0251 0.0166
K 0.0518 0.0705 0.0251 0.0660 0.0436
R 0.0342 0.0465 0.0166 0.0436 0.0288
```

Here, `residue_frequencies[:, None]` creates a 2D array consisting of a single column with the values of `residue_frequencies`, and `residue_frequencies[None, :]` a 2D array with these values as a single row. Taking their dot product (inner product) creates a matrix of expected frequencies where each entry consists of two `residue_frequencies` values multiplied with each other. For example, `expected_frequencies['D', 'E']` is equal to `residue_frequencies['D'] * residue_frequencies['E']`.

We can now calculate the log-odds matrix by dividing the observed frequencies by the expected frequencies and taking the logarithm:

```
>>> m = np.log2(observed_frequencies / expected_frequencies)
>>> print(m)
      D    E    H    K    R
```

(continues on next page)

(continued from previous page)

```
D   2.1 -1.5 -5.1 -10.4 -4.2
E  -1.5  1.7 -4.4  -5.1 -8.3
H  -5.1 -4.4  3.3  -4.4 -4.7
K -10.4 -5.1 -4.4   1.9 -2.3
R  -4.2 -8.3 -4.7  -2.3  2.5
```

This matrix can be used as the substitution matrix when performing alignments. For example,

```
>>> from Bio.Align import PairwiseAligner
>>> aligner = PairwiseAligner()
>>> aligner.substitution_matrix = m
>>> aligner.gap_score = -3.0
>>> alignments = aligner.align("DEHEK", "DHHKK")
>>> print(alignments[0])
target            0 DEHEK 5
                  0 |.|.| 5
query             0 DHHKK 5

>>> print("%.2f" % alignments.score)
-2.18
>>> score = m["D", "D"] + m["E", "H"] + m["H", "H"] + m["E", "K"] + m["K", "K"]
>>> print("%.2f" % score)
-2.18
```

8.6 Alignment Tools

There are *lots* of algorithms out there for aligning sequences, both pairwise alignments and multiple sequence alignments. These calculations are relatively slow, and you generally wouldn't want to write such an algorithm in Python. For pairwise alignments, you can use Biopython's `PairwiseAligner` (see Chapter *Pairwise sequence alignment*), which is implemented in C and therefore fast. Alternatively, you can run an external alignment program by invoking it from Python. Normally you would:

1. Prepare an input file of your unaligned sequences, typically this will be a FASTA file which you might create using `Bio.SeqIO` (see Chapter *Sequence Input/Output*).
2. Run the alignment program by running its command using Python's `subprocess` module.
3. Read the output from the tool, i.e. your aligned sequences, typically using `Bio.AlignIO` (see earlier in this chapter).

Here, we will show a few examples of this workflow.

8.6.1 ClustalW

ClustalW is a popular command line tool for multiple sequence alignment (there is also a graphical interface called ClustalX). Before trying to use ClustalW from within Python, you should first try running the ClustalW tool yourself by hand at the command line, to familiarize yourself the other options.

For the most basic usage, all you need is to have a FASTA input file, such as opuntia.fasta (available online or in the Doc/examples subdirectory of the Biopython source code). This is a small FASTA file containing seven prickly-pear DNA sequences (from the cactus family *Opuntia*). By default ClustalW will generate an alignment and guide tree file with names based on the input FASTA file, in this case `opuntia.aln` and `opuntia.dnd`, but you can override this or make it explicit:

```
>>> import subprocess
>>> cmd = "clustalw2 -infile=opuntia.fasta"
>>> results = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, text=True)
```

Notice here we have given the executable name as `clustalw2`, indicating we have version two installed, which has a different filename to version one (`clustalw`, the default). Fortunately both versions support the same set of arguments at the command line (and indeed, should be functionally identical).

You may find that even though you have ClustalW installed, the above command doesn't work – you may get a message about "command not found" (especially on Windows). This indicated that the ClustalW executable is not on your PATH (an environment variable, a list of directories to be searched). You can either update your PATH setting to include the location of your copy of ClustalW tools (how you do this will depend on your OS), or simply type in the full path of the tool. Remember, in Python strings `\n` and `\t` are by default interpreted as a new line and a tab – which is why we're put a letter "r" at the start for a raw string that isn't translated in this way. This is generally good practice when specifying a Windows style file name.

```
>>> import os
>>> clustalw_exe = r"C:\Program Files\new clustal\clustalw2.exe"
>>> assert os.path.isfile(clustalw_exe), "Clustal W executable missing"
>>> cmd = clustalw_exe + " -infile=opuntia.fasta"
>>> results = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, text=True)
```

Now, at this point it helps to know about how command line tools "work". When you run a tool at the command line, it will often print text output directly to screen. This text can be captured or redirected, via two "pipes", called standard output (the normal results) and standard error (for error messages and debug messages). There is also standard input, which is any text fed into the tool. These names get shortened to stdin, stdout and stderr. When the tool finishes, it has a return code (an integer), which by convention is zero for success, while a non-zero return code indicates that an error has occurred.

In the example of ClustalW above, when run at the command line all the important output is written directly to the output files. Everything normally printed to screen while you wait is captured in `results.stdout` and `results.stderr`, while the return code is stored in `results.returncode`.

What we care about are the two output files, the alignment and the guide tree. We didn't tell ClustalW what filenames to use, but it defaults to picking names based on the input file. In this case the output should be in the file `opuntia.aln`. You should be able to work out how to read in the alignment using `Bio.AlignIO` by now:

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("opuntia.aln", "clustal")
>>> print(align)
Alignment with 7 rows and 906 columns
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.1|AF191
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273291|gb|AF191665.1|AF191
```

In case you are interested (and this is an aside from the main thrust of this chapter), the `opuntia.dnd` file ClustalW creates is just a standard Newick tree file, and `Bio.Phylo` can parse these:

```
>>> from Bio import Phylo
>>> tree = Phylo.read("opuntia.dnd", "newick")
>>> Phylo.draw_ascii(tree)
```

(continues on next page)

(continued from previous page)

```
                              _____________ gi|6273291|gb|AF191665.1|AF191665
 _____________________________|
|                             |   ______ gi|6273290|gb|AF191664.1|AF191664
|                             |__|
|                                |_____ gi|6273289|gb|AF191663.1|AF191663
|
_|__________________ gi|6273287|gb|AF191661.1|AF191661
 |
 |__________ gi|6273286|gb|AF191660.1|AF191660
 |
 |    __ gi|6273285|gb|AF191659.1|AF191659
 |___|
     | gi|6273284|gb|AF191658.1|AF191658
```

Chapter *Phylogenetics with Bio.Phylo* covers Biopython's support for phylogenetic trees in more depth.

8.6.2 MUSCLE

MUSCLE is a more recent multiple sequence alignment tool than ClustalW. As before, we recommend you try using MUSCLE from the command line before trying to run it from Python.

For the most basic usage, all you need is to have a FASTA input file, such as opuntia.fasta (available online or in the Doc/examples subdirectory of the Biopython source code). You can then tell MUSCLE to read in this FASTA file, and write the alignment to an output file named `opuntia.txt`:

```
>>> import subprocess
>>> cmd = "muscle -align opuntia.fasta -output opuntia.txt"
>>> results = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, text=True)
```

MUSCLE will output the alignment as a FASTA file (using gapped sequences). The `Bio.AlignIO` module is able to read this alignment using `format="fasta"`:

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("opuntia.txt", "fasta")
>>> print(align)
Alignment with 7 rows and 906 columns
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.1|AF191663
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273291|gb|AF191665.1|AF191665
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.1|AF191664
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.1|AF191661
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.1|AF191660
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.1|AF191659
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.1|AF191658
```

You can also set the other optional parameters; see MUSCLE's built-in help for details.

8.6.3 EMBOSS needle and water

The EMBOSS suite includes the `water` and `needle` tools for Smith-Waterman algorithm local alignment, and Needleman-Wunsch global alignment. The tools share the same style interface, so switching between the two is trivial – we'll just use `needle` here.

Suppose you want to do a global pairwise alignment between two sequences, prepared in FASTA format as follows:

```
>HBA_HUMAN
MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHG
KKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTP
AVHASLDKFLASVSTVLTSKYR
```

in a file `alpha.faa`, and secondly in a file `beta.faa`:

```
>HBB_HUMAN
MVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPK
VKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLAHHFG
KEFTPPVQAAYQKVVAGVANALAHKYH
```

You can find copies of these example files with the Biopython source code under the `Doc/examples/` directory.

The command to align these two sequences against each other using `needle` is as follows:

```
needle -outfile=needle.txt -asequence=alpha.faa -bsequence=beta.faa -gapopen=10 -
↪gapextend=0.5
```

Why not try running this by hand at the command prompt? You should see it does a pairwise comparison and records the output in the file `needle.txt` (in the default EMBOSS alignment file format).

Even if you have EMBOSS installed, running this command may not work – you might get a message about "command not found" (especially on Windows). This probably means that the EMBOSS tools are not on your PATH environment variable. You can either update your PATH setting, or simply use the full path to the tool, for example:

```
C:\EMBOSS\needle.exe -outfile=needle.txt -asequence=alpha.faa -bsequence=beta.faa -
↪gapopen=10 -gapextend=0.5
```

Next we want to use Python to run this command for us. As explained above, for full control, we recommend you use Python's built-in `subprocess` module:

```
>>> import sys
>>> import subprocess
>>> cmd = "needle -outfile=needle.txt -asequence=alpha.faa -bsequence=beta.faa -
↪gapopen=10 -gapextend=0.5"
>>> results = subprocess.run(
...     cmd,
...     stdout=subprocess.PIPE,
...     stderr=subprocess.PIPE,
...     text=True,
...     shell=(sys, platform != "win32"),
... )
>>> print(results.stdout)

>>> print(results.stderr)
Needleman-Wunsch global alignment of two sequences
```

Next we can load the output file with `Bio.AlignIO` as discussed earlier in this chapter, as the `emboss` format:

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("needle.txt", "emboss")
>>> print(align)
Alignment with 2 rows and 149 columns
MV-LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTY...KYR HBA_HUMAN
MVHLTPEEKSAVTALWGKV--NVDEVGGEALGRLLVVYPWTQRF...KYH HBB_HUMAN
```

In this example, we told EMBOSS to write the output to a file, but you *can* tell it to write the output to stdout instead (useful if you don't want a temporary output file to get rid of – use `outfile=stdout` argument):

```
>>> cmd = "needle -outfile=stdout -asequence=alpha.faa -bsequence=beta.faa -gapopen=10 -
↪gapextend=0.5"
>>> child = subprocess.Popen(
...     cmd,
...     stdout=subprocess.PIPE,
...     stderr=subprocess.PIPE,
...     text=True,
...     shell=(sys.platform != "win32"),
... )
>>> align = AlignIO.read(child.stdout, "emboss")
>>> print(align)
Alignment with 2 rows and 149 columns
MV-LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTY...KYR HBA_HUMAN
MVHLTPEEKSAVTALWGKV--NVDEVGGEALGRLLVVYPWTQRF...KYH HBB_HUMAN
```

Similarly, it is possible to read *one* of the inputs from stdin (e.g. `asequence="stdin"`).

This has only scratched the surface of what you can do with `needle` and `water`. One useful trick is that the second file can contain multiple sequences (say five), and then EMBOSS will do five pairwise alignments.

CHAPTER

NINE

PAIRWISE ALIGNMENTS USING PAIRWISE2

Please note that Bio.pairwise2 was deprecated in Release 1.80. As an alternative, please consider using `Bio.Align.PairwiseAligner` (described in Chapter *Pairwise sequence alignment*).

`Bio.pairwise2` contains essentially the same algorithms as `water` (local) and `needle` (global) from the EMBOSS suite (see above) and should return the same results. The `pairwise2` module has undergone some optimization regarding speed and memory consumption recently (Biopython versions >1.67) so that for short sequences (global alignments: ~2000 residues, local alignments ~600 residues) it's faster (or equally fast) to use `pairwise2` than calling EMBOSS' `water` or `needle` via the command line tools.

Suppose you want to do a global pairwise alignment between the same two hemoglobin sequences from above (`HBA_HUMAN`, `HBB_HUMAN`) stored in `alpha.faa` and `beta.faa`:

```
>>> from Bio import pairwise2
>>> from Bio import SeqIO
>>> seq1 = SeqIO.read("alpha.faa", "fasta")
>>> seq2 = SeqIO.read("beta.faa", "fasta")
>>> alignments = pairwise2.align.globalxx(seq1.seq, seq2.seq)
```

As you see, we call the alignment function with `align.globalxx`. The tricky part are the last two letters of the function name (here: `xx`), which are used for decoding the scores and penalties for matches (and mismatches) and gaps. The first letter decodes the match score, e.g. `x` means that a match counts 1 while mismatches have no costs. With `m` general values for either matches or mismatches can be defined (for full details see *Bio.pairwise2*). The second letter decodes the cost for gaps; `x` means no gap costs at all, with `s` different penalties for opening and extending a gap can be assigned. So, `globalxx` means that only matches between both sequences are counted.

Our variable `alignments` now contains a list of alignments (at least one) which have the same optimal score for the given conditions. In our example this are 80 different alignments with the score 72 (`Bio.pairwise2` will return up to 1000 alignments). Have a look at one of these alignments:

```
>>> len(alignments)
80
>>> print(alignments[0])
Alignment(seqA='MV-LSPADKTNV---K-A--A-WGKVGAHAG...YR-', seqB='MVHL-----T--PEEKSAVTALWGKV-
↪---...Y-H', score=72.0, start=0, end=217)
```

Each alignment is a named tuple consisting of the two aligned sequences, the score, the start and the end positions of the alignment (in global alignments the start is always 0 and the end the length of the alignment). `Bio.pairwise2` has a function `format_alignment` for a nicer printout:

```
>>> print(pairwise2.format_alignment(*alignments[0]))
MV-LSPADKTNV---K-A--A-WGKVGAHAG---EY-GA-EALE-RMFLSF----PTTK-TY--F...YR-
|| |     |     | |  | ||||        |  |  |||  |  |      |    |   |...|
```

(continues on next page)

(continued from previous page)

```
MVHL-----T--PEEKSAVTALWGKV-----NVDE-VG-GEAL-GR--L--LVVYP---WT-QRF...Y-H
  Score=72
```

Since Biopython 1.77 the required parameters can be supplied with keywords. The last example can now also be written as:

```
>>> alignments = pairwise2.align.globalxx(sequenceA=seq1.seq, sequenceB=seq2.seq)
```

Better alignments are usually obtained by penalizing gaps: higher costs for opening a gap and lower costs for extending an existing gap. For amino acid sequences match scores are usually encoded in matrices like `PAM` or `BLOSUM`. Thus, a more meaningful alignment for our example can be obtained by using the BLOSUM62 matrix, together with a gap open penalty of 10 and a gap extension penalty of 0.5 (using `globalds`):

```
>>> from Bio import pairwise2
>>> from Bio import SeqIO
>>> from Bio.Align import substitution_matrices
>>> blosum62 = substitution_matrices.load("BLOSUM62")
>>> seq1 = SeqIO.read("alpha.faa", "fasta")
>>> seq2 = SeqIO.read("beta.faa", "fasta")
>>> alignments = pairwise2.align.globalds(seq1.seq, seq2.seq, blosum62, -10, -0.5)
>>> len(alignments)
2
>>> print(pairwise2.format_alignment(*alignments[0]))
MV-LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTY...KYR
|| |.|..|..|.|.|||| ......|...........|.......||.
MVHLTPEEKSAVTALWGKV-NVDEVGGEALGRLLVVYPWTQRFF...KYH
  Score=292.5
```

This alignment has the same score that we obtained earlier with EMBOSS needle using the same sequences and the same parameters.

Local alignments are called similarly with the function `align.localXX`, where again XX stands for a two letter code for the match and gap functions:

```
>>> from Bio import pairwise2
>>> from Bio.Align import substitution_matrices
>>> blosum62 = substitution_matrices.load("BLOSUM62")
>>> alignments = pairwise2.align.localds("LSPADKTNVKAA", "PEEKSAV", blosum62, -10, -1)
>>> print(pairwise2.format_alignment(*alignments[0]))
3 PADKTNV
  |..|..|
1 PEEKSAV
  Score=16
```

In recent Biopython versions, `format_alignment` will only print the aligned part of a local alignment (together with the start positions in 1-based notation, as shown in the above example). If you are also interested in the non- aligned parts of the sequences, use the keyword-parameter `full_sequences=True`:

```
>>> from Bio import pairwise2
>>> from Bio.Align import substitution_matrices
>>> blosum62 = substitution_matrices.load("BLOSUM62")
>>> alignments = pairwise2.align.localds("LSPADKTNVKAA", "PEEKSAV", blosum62, -10, -1)
>>> print(pairwise2.format_alignment(*alignments[0], full_sequences=True))
```

(continues on next page)

(continued from previous page)

```
LSPADKTNVKAA
  |..|..|
--PEEKSAV---
  Score=16
```

Note that local alignments must, as defined by Smith & Waterman, have a positive score (>0). Thus, `pairwise2` may return no alignments if no score >0 has been obtained. Also, `pairwise2` will not report alignments which are the result of the addition of zero-scoring extensions on either site. In the next example, the pairs serine/aspartic acid (S/D) and lysine/asparagine (K/N) both have a match score of 0. As you see, the aligned part has not been extended:

```
>>> from Bio import pairwise2
>>> from Bio.Align import substitution_matrices
>>> blosum62 = substitution_matrices.load("BLOSUM62")
>>> alignments = pairwise2.align.localds("LSSPADKTNVKKAA", "DDPEEKSAVNN", blosum62, -10,␣
↪-1)
>>> print(pairwise2.format_alignment(*alignments[0]))
4 PADKTNV
  |..|..|
3 PEEKSAV
  Score=16
```

Instead of supplying a complete match/mismatch matrix, the match code `m` allows for easy defining general match/mismatch values. The next example uses match/mismatch scores of 5/-4 and gap penalties (open/extend) of 2/0.5 using `localms`:

```
>>> alignments = pairwise2.align.localms("AGAACT", "GAC", 5, -4, -2, -0.5)
>>> print(pairwise2.format_alignment(*alignments[0]))
2 GAAC
  | ||
1 G-AC
  Score=13
```

One useful keyword argument of the `Bio.pairwise2.align` functions is `score_only`. When set to `True` it will only return the score of the best alignment(s), but in a significantly shorter time. It will also allow the alignment of longer sequences before a memory error is raised. Another useful keyword argument is `one_alignment_only=True` which will also result in some speed gain.

Unfortunately, `Bio.pairwise2` does not work with Biopython's multiple sequence alignment objects (yet). However, the module has some interesting advanced features: you can define your own match and gap functions (interested in testing affine logarithmic gap costs?), gap penalties and end gaps penalties can be different for both sequences, sequences can be supplied as lists (useful if you have residues that are encoded by more than one character), etc. These features are hard (if at all) to realize with other alignment tools. For more details see the module's API documentation *Bio.pairwise2*.

CHAPTER

TEN

BLAST (NEW)

Hey, everybody loves BLAST right? I mean, geez, how can it get any easier to do comparisons between one of your sequences and every other sequence in the known world? But, of course, this section isn't about how cool BLAST is, since we already know that. It is about the problem with BLAST – it can be really difficult to deal with the volume of data generated by large runs, and to automate BLAST runs in general.

Fortunately, the Biopython folks know this only too well, so they've developed lots of tools for dealing with BLAST and making things much easier. This section details how to use these tools and do useful things with them.

Dealing with BLAST can be split up into two steps, both of which can be done from within Biopython. Firstly, running BLAST for your query sequence(s), and getting some output. Secondly, parsing the BLAST output in Python for further analysis.

Your first introduction to running BLAST was probably via the NCBI BLAST web page. In fact, there are lots of ways you can run BLAST, which can be categorized in several ways. The most important distinction is running BLAST locally (on your own machine), and running BLAST remotely (on another machine, typically the NCBI servers). We're going to start this chapter by invoking the NCBI online BLAST service from within a Python script.

10.1 Running BLAST over the Internet

We use the function `qblast` in the `Bio.Blast` module to call the online version of BLAST.

The NCBI guidelines state:

1. Do not contact the server more often than once every 10 seconds.
2. Do not poll for any single RID more often than once a minute.
3. Use the URL parameter email and tool, so that the NCBI can contact you if there is a problem.
4. Run scripts weekends or between 9 pm and 5 am Eastern time on weekdays if more than 50 searches will be submitted.

`Blast.qblast` follows the first two points automatically. To fulfill the third point, set the `Blast.email` variable (the `Blast.tool` variable is already set to `"biopython"` by default):

```
>>> from Bio import Blast
>>> Blast.tool
'biopython'
>>> Blast.email = "A.N.Other@example.com"
```

10.1.1 BLAST arguments

The `qblast` function has three non-optional arguments:

- The first argument is the BLAST program to use for the search, as a lower case string. The programs and their options are described at the NCBI BLAST web page. Currently `qblast` only works with blastn, blastp, blastx, tblast and tblastx.
- The second argument specifies the databases to search against. Again, the options for this are available on NCBI's BLAST Help pages.
- The third argument is a string containing your query sequence. This can either be the sequence itself, the sequence in fasta format, or an identifier like a GI number.

The `qblast` function also takes a number of other option arguments, which are basically analogous to the different parameters you can set on the BLAST web page. We'll just highlight a few of them here:

- The argument `url_base` sets the base URL for running BLAST over the internet. By default it connects to the NCBI, but one can use this to connect to an instance of NCBI BLAST running in the cloud. Please refer to the documentation for the `qblast` function for further details.
- The `qblast` function can return the BLAST results in various formats, which you can choose with the optional `format_type` keyword: `"XML"`, `"HTML"`, `"Text"`, `"XML2"`, `"JSON2"`, or `"Tabular"`. The default is `"XML"`, as that is the format expected by the parser, described in section *Parsing BLAST output* below.
- The argument `expect` sets the expectation or e-value threshold.

For more about the optional BLAST arguments, we refer you to the NCBI's own documentation, or that built into Biopython:

```
>>> from Bio import Blast
>>> help(Blast.qblast)
```

Note that the default settings on the NCBI BLAST website are not quite the same as the defaults on QBLAST. If you get different results, you'll need to check the parameters (e.g., the expectation value threshold and the gap values).

For example, if you have a nucleotide sequence you want to search against the nucleotide database (nt) using BLASTN, and you know the GI number of your query sequence, you can use:

```
>>> from Bio import Blast
>>> result_stream = Blast.qblast("blastn", "nt", "8332116")
```

Alternatively, if we have our query sequence already in a FASTA formatted file, we just need to open the file and read in this record as a string, and use that as the query argument:

```
>>> from Bio import Blast
>>> fasta_string = open("m_cold.fasta").read()
>>> result_stream = Blast.qblast("blastn", "nt", fasta_string)
```

We could also have read in the FASTA file as a `SeqRecord` and then supplied just the sequence itself:

```
>>> from Bio import Blast
>>> from Bio import SeqIO
>>> record = SeqIO.read("m_cold.fasta", "fasta")
>>> result_stream = Blast.qblast("blastn", "nt", record.seq)
```

Supplying just the sequence means that BLAST will assign an identifier for your sequence automatically. You might prefer to call `format` on the `SeqRecord` object to make a FASTA string (which will include the existing identifier):

```
>>> from Bio import Blast
>>> from Bio import SeqIO
>>> records = SeqIO.parse("ls_orchid.gbk", "genbank")
>>> record = next(records)
>>> result_stream = Blast.qblast("blastn", "nt", format(record, "fasta"))
```

This approach makes more sense if you have your sequence(s) in a non-FASTA file format which you can extract using `Bio.SeqIO` (see Chapter *Sequence Input/Output*).

10.1.2 Saving BLAST results

Whatever arguments you give the `qblast()` function, you should get back your results as a stream of `bytes` data (by default in XML format). The next step would be to parse the XML output into Python objects representing the search results (Section *Parsing BLAST output*), but you might want to save a local copy of the output file first. I find this especially useful when debugging my code that extracts info from the BLAST results (because re-running the online search is slow and wastes the NCBI computer time).

We need to be a bit careful since we can use `result_stream.read()` to read the BLAST output only once – calling `result_stream.read()` again returns an empty `bytes` object.

```
>>> with open("my_blast.xml", "wb") as out_stream:
...     out_stream.write(result_stream.read())
...
>>> result_stream.close()
```

After doing this, the results are in the file `my_blast.xml` and `result_stream` has had all its data extracted (so we closed it). However, the `parse` function of the BLAST parser (described in *Parsing BLAST output*) takes a file-like object, so we can just open the saved file for input as `bytes`:

```
>>> result_stream = open("my_blast.xml", "rb")
```

Now that we've got the BLAST results back into a data stream again, we are ready to do something with them, so this leads us right into the parsing section (see Section *Parsing BLAST output* below). You may want to jump ahead to that now ....

10.1.3 Obtaining BLAST output in other formats

By using the `format_type` argument when calling `qblast`, you can obtain BLAST output in formats other than XML. Below is an example of reading BLAST output in JSON format. Using `format_type="JSON2"`, the data provided by `Blast.qblast` will be in zipped JSON format:

```
>>> from Bio import Blast
>>> from Bio import SeqIO
>>> record = SeqIO.read("m_cold.fasta", "fasta")
>>> result_stream = Blast.qblast("blastn", "nt", record.seq, format_type="JSON2")
>>> data = result_stream.read()
>>> data[:4]
b'PK\x03\x04'
```

which is the ZIP file magic number.

```
>>> with open("myzipfile.zip", "wb") as out_stream:
...     out_stream.write(data)
...
13813
```

Note that we read and write the data as `bytes`. Now open the ZIP file we created:

```
>>> import zipfile
>>> myzipfile = zipfile.ZipFile("myzipfile.zip")
>>> myzipfile.namelist()
['N5KN7UMJ013.json', 'N5KN7UMJ013_1.json']
>>> stream = myzipfile.open("N5KN7UMJ013.json")
>>> data = stream.read()
```

These data are `bytes`, so we need to decode them to get a string object:

```
>>> data = data.decode()
>>> print(data)
{
    "BlastJSON": [
        {"File": "N5KN7UMJ013_1.json" }
    ]
}
```

Now open the second file contained in the ZIP file to get the BLAST results in JSON format:

```
>>> stream = myzipfile.open("N5KN7UMJ013_1.json")
>>> data = stream.read()
>>> len(data)
145707
>>> data = data.decode()
>>> print(data)
{
  "BlastOutput2": {
    "report": {
      "program": "blastn",
      "version": "BLASTN 2.14.1+",
      "reference": "Stephen F. Altschul, Thomas L. Madden, Alejandro A. ...
      "search_target": {
        "db": "nt"
      },
      "params": {
        "expect": 10,
        "sc_match": 2,
        "sc_mismatch": -3,
        "gap_open": 5,
        "gap_extend": 2,
        "filter": "L;m;"
      },
      "results": {
        "search": {
          "query_id": "Query_69183",
          "query_len": 1111,
```

(continues on next page)

(continued from previous page)

```
          "query_masking": [
            {
              "from": 797,
              "to": 1110
            }
          ],
          "hits": [
            {
              "num": 1,
              "description": [
                {
                  "id": "gi|1219041180|ref|XM_021875076.1|",
...
```

We can use the JSON parser in Python's standard library to convert the JSON data into a regular Python dictionary:

```
>>> import json
>>> d = json.loads(data)
>>> print(d)
{'BlastOutput2': {'report': {'program': 'blastn', 'version': 'BLASTN 2.14.1+',
 'reference': 'Stephen F. Altschul, Thomas L. Madden, Alejandro A. Sch&auml;ffer,
 Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
 "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs",
 Nucleic Acids Res. 25:3389-3402.',
 'search_target': {'db': 'nt'}, 'params': {'expect': 10, 'sc_match': 2,
 'sc_mismatch': -3, 'gap_open': 5, 'gap_extend': 2, 'filter': 'L;m;'},
 'results': {'search': {'query_id': 'Query_128889', 'query_len': 1111,
 'query_masking': [{'from': 797, 'to': 1110}], 'hits': [{'num': 1,
 'description': [{'id': 'gi|1219041180|ref|XM_021875076.1|', 'accession':
 'XM_021875076', 'title':
 'PREDICTED: Chenopodium quinoa cold-regulated 413 plasma membrane protein 2-like␣
↪(LOC110697660), mRNA',
 'taxid': 63459, 'sciname': 'Chenopodium quinoa'}], 'len': 1173, 'hsps':
 [{'num': 1, 'bit_score': 435.898, 'score': 482, 'evalue': 9.02832e-117,
 'identity': 473, 'query_from'
...
```

10.2 Running BLAST locally

10.2.1 Introduction

Running BLAST locally (as opposed to over the internet, see Section *Running BLAST over the Internet*) has at least major two advantages:

- Local BLAST may be faster than BLAST over the internet;
- Local BLAST allows you to make your own database to search for sequences against.

Dealing with proprietary or unpublished sequence data can be another reason to run BLAST locally. You may not be allowed to redistribute the sequences, so submitting them to the NCBI as a BLAST query would not be an option.

Unfortunately, there are some major drawbacks too – installing all the bits and getting it setup right takes some effort:

- Local BLAST requires command line tools to be installed.
- Local BLAST requires (large) BLAST databases to be setup (and potentially kept up to date).

10.2.2 Standalone NCBI BLAST+

The "new" NCBI BLAST+ suite was released in 2009. This replaces the old NCBI "legacy" BLAST package (see *Other versions of BLAST*).

This section will show briefly how to use these tools from within Python. If you have already read or tried the alignment tool examples in Section *Alignment Tools* this should all seem quite straightforward. First, we construct a command line string (as you would type in at the command line prompt if running standalone BLAST by hand). Then we can execute this command from within Python.

For example, taking a FASTA file of gene nucleotide sequences, you might want to run a BLASTX (translation) search against the non-redundant (NR) protein database. Assuming you (or your systems administrator) has downloaded and installed the NR database, you might run:

```
$ blastx -query opuntia.fasta -db nr -out opuntia.xml -evalue 0.001 -outfmt 5
```

This should run BLASTX against the NR database, using an expectation cut-off value of 0.001 and produce XML output to the specified file (which we can then parse). On my computer this takes about six minutes - a good reason to save the output to a file so you can repeat any analysis as needed.

From within python we can use the `subprocess` module to build the command line string, and run it:

```
>>> import subprocess
>>> cmd = "blastx -query opuntia.fasta -db nr -out opuntia.xml"
>>> cmd += " -evalue 0.001 -outfmt 5"
>>> subprocess.run(cmd, shell=True)
```

In this example there shouldn't be any output from BLASTX to the terminal. You may want to check the output file `opuntia.xml` has been created.

As you may recall from earlier examples in the tutorial, the `opuntia.fasta` contains seven sequences, so the BLAST XML output should contain multiple results. Therefore use `Bio.Blast.parse()` to parse it as described below in Section *Parsing BLAST output*.

10.2.3 Other versions of BLAST

NCBI BLAST+ (written in C++) was first released in 2009 as a replacement for the original NCBI "legacy" BLAST (written in C) which is no longer being updated. You may also come across Washington University BLAST (WU-BLAST), and its successor, Advanced Biocomputing BLAST (AB-BLAST, released in 2009, not free/open source). These packages include the command line tools `wu-blastall` and `ab-blastall`, which mimicked `blastall` from the NCBI "legacy" BLAST suite. Biopython does not currently provide wrappers for calling these tools, but should be able to parse any NCBI compatible output from them.

10.3 Parsing BLAST output

As mentioned above, BLAST can generate output in various formats, such as XML, HTML, and plain text. Originally, Biopython had parsers for BLAST plain text and HTML output, as these were the only output formats offered at the time. These parsers have now been removed from Biopython, as the BLAST output in these formats kept changing, each time breaking the Biopython parsers. Nowadays, Biopython can parse BLAST output in the XML format, the XML2 format, and tabular format. This chapter describes the parser for BLAST output in the XML and XML2 formats using the `Bio.Blast.parse` function. This function automatically detects if the XML file is in the XML format or in the XML2 format. BLAST output in tabular format can be parsed as alignments using the `Bio.Align.parse` function (see the section *Tabular output from BLAST or FASTA*).

You can get BLAST output in XML format in various ways. For the parser, it doesn't matter how the output was generated, as long as it is in the XML format.

- You can use Biopython to run BLAST over the internet, as described in section *Running BLAST over the Internet*.
- You can use Biopython to run BLAST locally, as described in section *Running BLAST locally*.
- You can do the BLAST search yourself on the NCBI site through your web browser, and then save the results. You need to choose XML as the format in which to receive the results, and save the final BLAST page you get (you know, the one with all of the interesting results!) to a file.
- You can also run BLAST locally without using Biopython, and save the output in a file. Again, you need to choose XML as the format in which to receive the results.

The important point is that you do not have to use Biopython scripts to fetch the data in order to be able to parse it. Doing things in one of these ways, you then need to get a file-like object to the results. In Python, a file-like object or handle is just a nice general way of describing input to any info source so that the info can be retrieved using `read()` and `readline()` functions (see Section *What the heck is a handle?*).

If you followed the code above for interacting with BLAST through a script, then you already have `result_stream`, the file-like object to the BLAST results. For example, using a GI number to do an online search:

```
>>> from Bio import Blast
>>> result_stream = Blast.qblast("blastn", "nt", "8332116")
```

If instead you ran BLAST some other way, and have the BLAST output (in XML format) in the file `my_blast.xml`, all you need to do is to open the file for reading (as `bytes`):

```
>>> result_stream = open("my_blast.xml", "rb")
```

Now that we've got a data stream, we are ready to parse the output. The code to parse it is really quite small. If you expect a single BLAST result (i.e., you used a single query):

```
>>> from Bio import Blast
>>> blast_record = Blast.read(result_stream)
```

or, if you have lots of results (i.e., multiple query sequences):

```
>>> from Bio import Blast
>>> blast_records = Blast.parse(result_stream)
```

Just like `Bio.SeqIO` and `Bio.Align` (see Chapters *Sequence Input/Output* and *Sequence alignments*), we have a pair of input functions, `read` and `parse`, where `read` is for when you have exactly one object, and `parse` is an iterator for when you can have lots of objects – but instead of getting `SeqRecord` or `Alignment` objects, we get BLAST record objects.

To be able to handle the situation where the BLAST file may be huge, containing thousands of results, `Blast.parse()` returns an iterator. In plain English, an iterator allows you to step through the BLAST output, retrieving BLAST records one by one for each BLAST search result:

```
>>> from Bio import Blast
>>> blast_records = Blast.parse(result_stream)
>>> blast_record = next(blast_records)
# ... do something with blast_record
>>> blast_record = next(blast_records)
# ... do something with blast_record
>>> blast_record = next(blast_records)
# ... do something with blast_record
>>> blast_record = next(blast_records)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
# No further records
```

Or, you can use a `for`-loop:

```
>>> for blast_record in blast_records:
...     pass  # Do something with blast_record
...
```

Note though that you can step through the BLAST records only once. Usually, from each BLAST record you would save the information that you are interested in.

Alternatively, you can use `blast_records` as a list, for example by extracting one record by index, or by calling `len` or `print` on `blast_records`. The parser will then automatically iterate over the records and store them:

```
>>> from Bio import Blast
>>> blast_records = Blast.parse("xml_2222_blastx_001.xml")
>>> len(blast_records)  # this causes the parser to iterate over all records
7
>>> blast_records[2].query.description
'gi|5690369|gb|AF158246.1|AF158246 Cricetulus griseus glucose phosphate isomerase (GPI)␣
↪gene, partial intron sequence'
```

If your BLAST file is huge though, you may run into memory problems trying to save them all in a list.

If you start iterating over the records *before* using `blast_records` as a list, the parser will first reset the file stream to the beginning of the data to ensure that all records are neing read. Note that this will fail if the stream cannot be reset to the beginning, for example if the data are being read remotely (e.g. by qblast; see subsection *Running BLAST over the Internet*). In those cases, you can explicitly read the records into a list by calling `blast_records = blast_records[:]` before iterating over them. After reading in the records, it is safe to iterate over them or use them as a list.

Instead of opening the file yourself, you can just provide the file name:

```
>>> from Bio import Blast
>>> with Blast.parse("my_blast.xml") as blast_records:
...     for blast_record in blast_records:
...         pass  # Do something with blast_record
...
```

In this case, Biopython opens the file for you, and closes it as soon as the file is not needed any more (while it is

possible to simply use `blast_records = Blast.parse("my_blast.xml")`, it has the disadvantage that the file may stay open longer than strictly necessary, thereby wasting resources).

You can `print` the records to get a quick overview of their contents:

```
>>> from Bio import Blast
>>> with Blast.parse("my_blast.xml") as blast_records:
...     print(blast_records)
...
Program: BLASTN 2.2.27+
     db: refseq_rna

  Query: 42291 (length=61)
         mystery_seq
   Hits: ----  -----  ------------------------------------------------------------
            #  # HSP  ID + description
         ----  -----  ------------------------------------------------------------
            0      1  gi|262205317|ref|NR_030195.1|  Homo sapiens microRNA 52...
            1      1  gi|301171311|ref|NR_035856.1|  Pan troglodytes microRNA...
            2      1  gi|270133242|ref|NR_032573.1|  Macaca mulatta microRNA ...
            3      2  gi|301171322|ref|NR_035857.1|  Pan troglodytes microRNA...
            4      1  gi|301171267|ref|NR_035851.1|  Pan troglodytes microRNA...
            5      2  gi|262205330|ref|NR_030198.1|  Homo sapiens microRNA 52...
            6      1  gi|262205302|ref|NR_030191.1|  Homo sapiens microRNA 51...
            7      1  gi|301171259|ref|NR_035850.1|  Pan troglodytes microRNA...
            8      1  gi|262205451|ref|NR_030222.1|  Homo sapiens microRNA 51...
            9      2  gi|301171447|ref|NR_035871.1|  Pan troglodytes microRNA...
           10      1  gi|301171276|ref|NR_035852.1|  Pan troglodytes microRNA...
           11      1  gi|262205290|ref|NR_030188.1|  Homo sapiens microRNA 51...
           12      1  gi|301171354|ref|NR_035860.1|  Pan troglodytes microRNA...
           13      1  gi|262205281|ref|NR_030186.1|  Homo sapiens microRNA 52...
           14      2  gi|262205298|ref|NR_030190.1|  Homo sapiens microRNA 52...
           15      1  gi|301171394|ref|NR_035865.1|  Pan troglodytes microRNA...
           16      1  gi|262205429|ref|NR_030218.1|  Homo sapiens microRNA 51...
           17      1  gi|262205423|ref|NR_030217.1|  Homo sapiens microRNA 52...
           18      1  gi|301171401|ref|NR_035866.1|  Pan troglodytes microRNA...
           19      1  gi|270133247|ref|NR_032574.1|  Macaca mulatta microRNA ...
           20      1  gi|262205309|ref|NR_030193.1|  Homo sapiens microRNA 52...
           21      2  gi|270132717|ref|NR_032716.1|  Macaca mulatta microRNA ...
           22      2  gi|301171437|ref|NR_035870.1|  Pan troglodytes microRNA...
           23      2  gi|270133306|ref|NR_032587.1|  Macaca mulatta microRNA ...
           24      2  gi|301171428|ref|NR_035869.1|  Pan troglodytes microRNA...
           25      1  gi|301171211|ref|NR_035845.1|  Pan troglodytes microRNA...
           26      2  gi|301171153|ref|NR_035838.1|  Pan troglodytes microRNA...
           27      2  gi|301171146|ref|NR_035837.1|  Pan troglodytes microRNA...
           28      2  gi|270133254|ref|NR_032575.1|  Macaca mulatta microRNA ...
           29      2  gi|262205445|ref|NR_030221.1|  Homo sapiens microRNA 51...
           ~~~
           97      1  gi|356517317|ref|XM_003527287.1|  PREDICTED: Glycine ma...
           98      1  gi|297814701|ref|XM_002875188.1|  Arabidopsis lyrata su...
           99      1  gi|397513516|ref|XM_003827011.1|  PREDICTED: Pan panisc...
```

Usually, you'll be running one BLAST search at a time. Then, all you need to do is to pick up the first (and only) BLAST record in `blast_records`:

```
>>> from Bio import Blast
>>> blast_records = Blast.parse("my_blast.xml")
>>> blast_record = next(blast_records)
```

or more elegantly:

```
>>> from Bio import Blast
>>> blast_record = Blast.read(result_stream)
```

or, equivalently,

```
>>> from Bio import Blast
>>> blast_record = Blast.read("my_blast.xml")
```

(here, you don't need to use a `with` block as `Blast.read` will read the whole file and close it immediately afterwards).

I guess by now you're wondering what is in a BLAST record.

10.4 The BLAST Records, Record, and Hit classes

10.4.1 The BLAST Records class

A single BLAST output file can contain output from multiple BLAST queries. In Biopython, the information in a BLAST output file is stored in an `Bio.Blast.Records` object. This is an iterator returning one `Bio.Blast.Record` object (see subsection *The BLAST Record class*) for each query. The `Bio.Blast.Records` object has the following attributes describing the BLAST run:

- `source`: The input data from which the `Bio.Blast.Records` object was constructed (this could be a file name or path, or a file-like object).
- `program`: The specific BLAST program that was used (e.g., 'blastn').
- `version`: The version of the BLAST program (e.g., 'BLASTN 2.2.27+').
- `reference`: The literature reference to the BLAST publication.
- `db`: The BLAST database against which the query was run (e.g., 'nr').
- `query`: A `SeqRecord` object which may contain some or all of the following information:
  - `query.id`: SeqId of the query;
  - `query.description`: Definition line of the query;
  - `query.seq`: The query sequence.
- `param`: A dictionary with the parameters used for the BLAST run. You may find the following keys in this dictionary:
  - `'matrix'`: the scoring matrix used in the BLAST run (e.g., 'BLOSUM62') (string);
  - `'expect'`: threshold on the expected number of chance matches (float);
  - `'include'`: e-value threshold for inclusion in multipass model in psiblast (float);
  - `'sc-match'`: score for matching nucleotides (integer);
  - `'sc-mismatch'`: score for mismatched nucleotides (integer;
  - `'gap-open'`: gap opening cost (integer);

  - `'gap-extend'`: gap extension cost (integer);
  - `'filter'`: filtering options applied in the BLAST run (string);
  - `'pattern'`: PHI-BLAST pattern (string);
  - `'entrez-query'`: Limit of request to Entrez query (string).

- `mbstat`: A dictionary with Mega BLAST search statistics. See the description of the `Record.stat` attribute below (in subsection *The BLAST Record class*) for a description of the items in this dictionary. Only older versions of Mega BLAST store this information. As it is stored near the end of the BLAST output file, this attribute can only be accessed after the file has been read completely (by iterating over the records until a `StopIteration` is issued).

For our example, we find:

```
>>> blast_records
<Bio.Blast.Records source='my_blast.xml' program='blastn' version='BLASTN 2.2.27+' db=
↪'refseq_rna'>
>>> blast_records.source
'my_blast.xml'
>>> blast_records.program
'blastn'
>>> blast_records.version
'BLASTN 2.2.27+'
>>> blast_records.reference
'Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng␣
↪Zhang, Webb Miller, and David J. Lipman (1997), "Gapped BLAST and PSI-BLAST: a new␣
↪generation of protein database search programs", Nucleic Acids Res. 25:3389-3402.'
>>> blast_records.db
'refseq_rna'
>>> blast_records.param
{'expect': 10.0, 'sc-match': 2, 'sc-mismatch': -3, 'gap-open': 5, 'gap-extend': 2,
↪'filter': 'L;m;'}
>>> print(blast_records)
Program: BLASTN 2.2.27+
     db: refseq_rna

  Query: 42291 (length=61)
         mystery_seq
   Hits: ----  -----  ------------------------------------------------------------
            #  # HSP  ID + description
         ----  -----  ------------------------------------------------------------
            0      1  gi|262205317|ref|NR_030195.1|  Homo sapiens microRNA 52...
            1      1  gi|301171311|ref|NR_035856.1|  Pan troglodytes microRNA...
            2      1  gi|270133242|ref|NR_032573.1|  Macaca mulatta microRNA ...
            3      2  gi|301171322|ref|NR_035857.1|  Pan troglodytes microRNA...
...
```

10.4.2 The BLAST Record class

A `Bio.Blast.Record` object stores the information provided by BLAST for a single query. The `Bio.Blast.Record` class inherits from `list`, and is essentially a list of `Bio.Blast.Hit` objects (see section *The BLAST Hit class*). A `Bio.Blast.Record` object has the following two attributes:

- `query`: A `SeqRecord` object which may contain some or all of the following information:
  - `query.id`: SeqId of the query;
  - `query.description`: Definition line of the query;
  - `query.seq`: The query sequence.
- `stat`: A dictionary with statistical data of the BLAST hit. You may find the following keys in this dictionary:
  - `'db-num'`: number of sequences in BLAST db (integer);
  - `'db-len'`: length of BLAST db (integer);
  - `'hsp-len'`: effective HSP (High Scoring Pair) length (integer);
  - `'eff-space'`: effective search space (float);
  - `'kappa'`: Karlin-Altschul parameter K (float);
  - `'lambda'`: Karlin-Altschul parameter Lambda (float);
  - `'entropy'`: Karlin-Altschul parameter H (float)
- `message`: Some (error?) information.

Continuing with our example,

```
>>> blast_record
<Bio.Blast.Record query.id='42291'; 100 hits>
>>> blast_record.query
SeqRecord(seq=Seq(None, length=61), id='42291', name='<unknown name>', description=
↪'mystery_seq', dbxrefs=[])
>>> blast_record.stat
{'db-num': 3056429, 'db-len': 673143725, 'hsp-len': 0, 'eff-space': 0, 'kappa': 0.41,
↪'lambda': 0.625, 'entropy': 0.78}
>>> print(blast_record)
  Query: 42291 (length=61)
         mystery_seq
   Hits: ----  -----  ------------------------------------------------------------
            #  # HSP  ID + description
         ----  -----  ------------------------------------------------------------
            0      1  gi|262205317|ref|NR_030195.1|  Homo sapiens microRNA 52...
            1      1  gi|301171311|ref|NR_035856.1|  Pan troglodytes microRNA...
            2      1  gi|270133242|ref|NR_032573.1|  Macaca mulatta microRNA ...
            3      2  gi|301171322|ref|NR_035857.1|  Pan troglodytes microRNA...
...
```

As the `Bio.Blast.Record` class inherits from `list`, you can use it as such. For example, you can iterate over the record:

```
>>> for hit in blast_record:
...     hit
...
```

(continues on next page)

(continued from previous page)

```
<Bio.Blast.Hit target.id='gi|262205317|ref|NR_030195.1|' query.id='42291'; 1 HSP>
<Bio.Blast.Hit target.id='gi|301171311|ref|NR_035856.1|' query.id='42291'; 1 HSP>
<Bio.Blast.Hit target.id='gi|270133242|ref|NR_032573.1|' query.id='42291'; 1 HSP>
<Bio.Blast.Hit target.id='gi|301171322|ref|NR_035857.1|' query.id='42291'; 2 HSPs>
<Bio.Blast.Hit target.id='gi|301171267|ref|NR_035851.1|' query.id='42291'; 1 HSP>
...
```

To check how many hits the `blast_record` has, you can simply invoke Python's `len` function:

```
>>> len(blast_record)
100
```

Like Python lists, you can retrieve hits from a `Bio.Blast.Record` using indices:

```
>>> blast_record[0]  # retrieves the top hit
<Bio.Blast.Hit target.id='gi|262205317|ref|NR_030195.1|' query.id='42291'; 1 HSP>
>>> blast_record[-1]  # retrieves the last hit
<Bio.Blast.Hit target.id='gi|397513516|ref|XM_003827011.1|' query.id='42291'; 1 HSP>
```

To retrieve multiple hits from a `Bio.Blast.Record`, you can use the slice notation. This will return a new `Bio.Blast.Record` object containing only the sliced hits:

```
>>> blast_slice = blast_record[:3]  # slices the first three hits
>>> print(blast_slice)
  Query: 42291 (length=61)
         mystery_seq
   Hits: ----  -----  ------------------------------------------------------------
            #  # HSP  ID + description
         ----  -----  ------------------------------------------------------------
            0      1  gi|262205317|ref|NR_030195.1|  Homo sapiens microRNA 52...
            1      1  gi|301171311|ref|NR_035856.1|  Pan troglodytes microRNA...
            2      1  gi|270133242|ref|NR_032573.1|  Macaca mulatta microRNA ...
```

To create a copy of the `Bio.Blast.Record`, take the full slice:

```
>>> blast_record_copy = blast_record[:]
>>> type(blast_record_copy)
<class 'Bio.Blast.Record'>
>>> blast_record_copy  # list of all hits
<Bio.Blast.Record query.id='42291'; 100 hits>
```

This is particularly useful if you want to sort or filter the BLAST record (see *Sorting and filtering BLAST output*), but want to retain a copy of the original BLAST output.

You can also access `blast_record` as a Python dictionary and retrieve hits using the hit's ID as key:

```
>>> blast_record["gi|262205317|ref|NR_030195.1|"]
<Bio.Blast.Hit target.id='gi|262205317|ref|NR_030195.1|' query.id='42291'; 1 HSP>
```

If the ID is not found in the `blast_record`, a `KeyError` is raised:

```
>>> blast_record["unicorn_gene"]
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
KeyError: 'unicorn_gene'
```

You can get the full list of keys by using `.keys()` as usual:

```
>>> blast_record.keys()
['gi|262205317|ref|NR_030195.1|', 'gi|301171311|ref|NR_035856.1|', 'gi|270133242|ref|NR_
↪032573.1|', ...]
```

What if you just want to check whether a particular hit is present in the query results? You can do a simple Python membership test using the `in` keyword:

```
>>> "gi|262205317|ref|NR_030195.1|" in blast_record
True
>>> "gi|262205317|ref|NR_030194.1|" in blast_record
False
```

Sometimes, knowing whether a hit is present is not enough; you also want to know the rank of the hit. Here, the `index` method comes to the rescue:

```
>>> blast_record.index("gi|301171437|ref|NR_035870.1|")
22
```

Remember that Python uses zero-based indexing, so the first hit will be at index 0.

10.4.3 The BLAST Hit class

Each `Bio.Blast.Hit` object in the `blast_record` list represents one BLAST hit of the query against a target.

```
>>> hit = blast_record[0]
>>> hit
<Bio.Blast.Hit target.id='gi|262205317|ref|NR_030195.1|' query.id='42291'; 1 HSP>
>>> hit.target
SeqRecord(seq=Seq(None, length=61), id='gi|262205317|ref|NR_030195.1|', name='NR_030195',
↪ description='Homo sapiens microRNA 520b (MIR520B), microRNA', dbxrefs=[])
```

We can get a summary of the `hit` by printing it:

```
>>> print(blast_record[3])
Query: 42291
       mystery_seq
  Hit: gi|301171322|ref|NR_035857.1| (length=86)
       Pan troglodytes microRNA mir-520c (MIR520C), microRNA
 HSPs: ----  --------  ---------  ------  ---------------  ---------------------
          #   E-value  Bit score    Span      Query range              Hit range
       ----  --------  ---------  ------  ---------------  ---------------------
          0   8.9e-20     100.47      60           [1:61]                [13:73]
          1   3.3e-06      55.39      60           [0:60]                [73:13]
```

You see that we've got the essentials covered here:

- A hit is always for one query; the query ID and description are shown at the top of the summary.

- A hit consists of one or more alignments of the query against one target sequence. The target information is shown next is the summary. As shown above, the target can be accessed via the `target` attribute of the hit.
- Finally, there's a table containing quick information about the alignments each hit contains. In BLAST parlance, these alignments are called "High-scoring Segment Pairs", or HSPs (see section *The BLAST HSP class*). Each row in the table summarizes one HSP, including the HSP index, e-value, bit score, span (the alignment length including gaps), query coordinates, and target coordinates.

The `Bio.Blast.Hit` class is a subclass of `Bio.Align.Alignments` (plural; see Section *The Alignments class*), and therefore in essence is a list of `Bio.Align.Alignment` (singular; see Section *Alignment objects*) objects. In particular when aligning nucleotide sequences against the genome, the `Bio.Blast.Hit` object may consist of more than one `Bio.Align.Alignment` if a particular query aligns to more than one region of a chromosome. For protein alignments, usually a hit consists of only one alignment, especially for alignments of highly homologous sequences.

```
>>> type(hit)
<class 'Bio.Blast.Hit'>
>>> from Bio.Align import Alignments
>>> isinstance(hit, Alignments)
True
>>> len(hit)
1
```

For BLAST output in the XML2 format, a hit may have several targets with identical sequences but different sequence IDs and descriptions. These targets are accessible as the `hit.targets` attribute. In most cases, `hit.targets` has length 1 and only contains `hit.target`:

```
>>> from Bio import Blast
>>> blast_record = Blast.read("xml_2900_blastx_001_v2.xml")
>>> for hit in blast_record:
...     print(len(hit.targets))
...
1
1
2
1
1
1
1
1
1
1
```

However, as you can see in the output above, the third hit has multiple targets.

```
>>> hit = blast_record[2]
>>> hit.targets[0].seq
Seq(None, length=246)
>>> hit.targets[1].seq
Seq(None, length=246)
>>> hit.targets[0].id
'gi|684409690|ref|XP_009175831.1|'
>>> hit.targets[1].id
'gi|663044098|gb|KER20427.1|'
>>> hit.targets[0].name
'XP_009175831'
```

(continues on next page)

(continued from previous page)

```
>>> hit.targets[1].name
'KER20427'
>>> hit.targets[0].description
'hypothetical protein T265_11027 [Opisthorchis viverrini]'
>>> hit.targets[1].description
'hypothetical protein T265_11027 [Opisthorchis viverrini]'
```

As the sequence contents for the two targets are identical to each other, their sequence alignments are also identical. The alignments for this hit therefore only refers to `hit.targets[0]` (which is identical to `hit.target`), as the alignment for `hit.targets[1]` would be the same anyway.

10.4.4 The BLAST HSP class

Let's return to our main example, and look at the first (and only) alignment in the first hit. This alignment is an instance of the `Bio.Blast.HSP` class, which is a subclass of the `Alignment` class in `Bio.Align`:

```
>>> from Bio import Blast
>>> blast_record = Blast.read("my_blast.xml")
>>> hit = blast_record[0]
>>> len(hit)
1
>>> alignment = hit[0]
>>> alignment
<Bio.Blast.HSP target.id='gi|262205317|ref|NR_030195.1|' query.id='42291'; 2 rows x 61␣
↪columns>
>>> type(alignment)
<class 'Bio.Blast.HSP'>
>>> from Bio.Align import Alignment
>>> isinstance(alignment, Alignment)
True
```

The `alignment` object has attributes pointing to the target and query sequences, as well as a `coordinates` attribute describing the sequence alignment.

```
>>> alignment.target
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG'), id=
↪'gi|262205317|ref|NR_030195.1|', name='NR_030195', description='Homo sapiens microRNA␣
↪520b (MIR520B), microRNA', dbxrefs=[])
>>> alignment.query
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG'), id=
↪'42291', name='<unknown name>', description='mystery_seq', dbxrefs=[])
>>> alignment.target
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG'), id=
↪'gi|262205317|ref|NR_030195.1|', name='NR_030195', description='Homo sapiens microRNA␣
↪520b (MIR520B), microRNA', dbxrefs=[])
>>> alignment.query
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG'), id=
↪'42291', name='<unknown name>', description='mystery_seq', dbxrefs=[])
>>> print(alignment.coordinates)
[[ 0 61]
 [ 0 61]]
```

For translated BLAST searches, the `features` attribute of the target or query may contain a `SeqFeature` of type CDS that stores the amino acid sequence region. The `qualifiers` attribute of such a feature is a dictionary with a single key `'coded_by'`; the corresponding value specifies the nucleotide sequence region, in a GenBank-style string with 1-based coordinates, that encodes the amino acid sequence.

Each `Alignment` object has the following additional attributes:

- `score`: score of the High Scoring Pair (HSP);
- `annotations`: a dictionary that may contain the following keys:
  - `'bit score'`: score (in bits) of HSP (float);
  - `'evalue'`: e-value of HSP (float);
  - `'identity'`: number of identities in HSP (integer);
  - `'positive'`: number of positives in HSP (integer);
  - `'gaps'`: number of gaps in HSP (integer);
  - `'midline'`: formatting middle line.

The usual `Alignment` methods (see Section *Alignment objects*) can be applied to the `alignment`. For example, we can print the alignment:

```
>>> print(alignment)
Query : 42291 Length: 61 Strand: Plus
        mystery_seq
Target: gi|262205317|ref|NR_030195.1| Length: 61 Strand: Plus
        Homo sapiens microRNA 520b (MIR520B), microRNA

Score:111 bits(122), Expect:5e-23,
Identities:61/61(100%),  Positives:61/61(100%),  Gaps:0.61(0%)

gi|262205         0 CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGG
                  0 ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
42291             0 CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGG

gi|262205        60 G 61
                  60 | 61
42291            60 G 61
```

Let's just print out some summary info about all hits in our BLAST record greater than a particular threshold:

```
>>> E_VALUE_THRESH = 0.04
>>> for alignments in blast_record:
...     for alignment in alignments:
...         if alignment.annotations["evalue"] < E_VALUE_THRESH:
...             print("****Alignment****")
...             print("sequence:", alignment.target.id, alignment.target.description)
...             print("length:", len(alignment.target))
...             print("score:", alignment.score)
...             print("e value:", alignment.annotations["evalue"])
...             print(alignment[:, :50])
...
****Alignment****
sequence: gi|262205317|ref|NR_030195.1| Homo sapiens microRNA 520b (MIR520B), microRNA
```

(continues on next page)

(continued from previous page)

```
length: 61
score: 122.0
e value: 4.91307e-23
gi|262205         0 CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTC 50
                  0 |||||||||||||||||||||||||||||||||||||||||||||||||| 50
42291             0 CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTC 50

****Alignment****
sequence: gi|301171311|ref|NR_035856.1| Pan troglodytes microRNA mir-520b (MIR520B),␣
↪microRNA
length: 60
score: 120.0
e value: 1.71483e-22
gi|301171         0 CCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCC 50
                  0 |||||||||||||||||||||||||||||||||||||||||||||||||| 50
42291             1 CCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCC 51

****Alignment****
sequence: gi|270133242|ref|NR_032573.1| Macaca mulatta microRNA mir-519a (MIR519A),␣
↪microRNA
length: 85
score: 112.0
e value: 2.54503e-20
gi|270133        12 CCCTCTAGAGGGAAGCGCTTTCTGTGGTCTGAAAGAAAAGAAAGTGCTTC 62
                  0 |||||||.||||||||||||||.|||||||||||||||||||||||||| 50
42291             0 CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTC 50
...
```

10.4.5 Sorting and filtering BLAST output

If the ordering of hits in the BLAST output file doesn't suit your taste, you can use the `sort` method to resort the hits in the `Bio.Blast.Record` object. As an example, here we sort the hits based on the sequence length of each target, setting the `reverse` flag to `True` so that we sort in descending order.

```
>>> for hit in blast_record[:5]:
...     print(f"{hit.target.id} {len(hit.target)}")
...
gi|262205317|ref|NR_030195.1| 61
gi|301171311|ref|NR_035856.1| 60
gi|270133242|ref|NR_032573.1| 85
gi|301171322|ref|NR_035857.1| 86
gi|301171267|ref|NR_035851.1| 80

>>> sort_key = lambda hit: len(hit.target)
>>> blast_record.sort(key=sort_key, reverse=True)
>>> for hit in blast_recordt[:5]:
...     print(f"{hit.target.id} {len(hit.target)}")
...
gi|397513516|ref|XM_003827011.1| 6002
gi|390332045|ref|XM_776818.2| 4082
gi|390332043|ref|XM_003723358.1| 4079
```

(continues on next page)

(continued from previous page)

```
gi|356517317|ref|XM_003527287.1| 3251
gi|356543101|ref|XM_003539954.1| 2936
```

This will sort `blast_record` in place. Use `original_blast_record = blast_record[:]` before sorting if you want to retain a copy of the original, unsorted BLAST output.

To filter BLAST hits based on their properties, you can use Python's built-in `filter` with the approciate callback function to evaluate each hit. The callback function must accept as its argument a single `Hit` object and return `True` or `False`. Here is an example in which we filter out `Hit` objects that only have one HSP:

```
>>> filter_func = lambda hit: len(hit) > 1  # the callback function
>>> len(blast_record)  # no. of hits before filtering
100
>>> blast_record[:] = filter(filter_func, blast_record)
>>> len(blast_record)  # no. of hits after filtering
37
>>> for hit in blast_record[:5]:  # quick check for the hit lengths
...     print(f"{hit.target.id} {len(hit)}")
...
gi|301171322|ref|NR_035857.1| 2
gi|262205330|ref|NR_030198.1| 2
gi|301171447|ref|NR_035871.1| 2
gi|262205298|ref|NR_030190.1| 2
gi|270132717|ref|NR_032716.1| 2
```

Similarly, you can filter HSPs in each hit, for example on their e-value:

```
>>> filter_func = lambda hsp: hsp.annotations["evalue"] < 1.0e-12
>>> for hit in blast_record:
...     hit[:] = filter(filter_func, hit)
...
```

Probably you'd want to follow this up by removing all hits with no HSPs remaining:

```
>>> filter_func = lambda hit: len(hit) > 0
>>> blast_record[:] = filter(filter_func, blast_record)
>>> len(blast_record)
16
```

Use Python's built-in `map` function to modify hits or HSPs in the BLAST record. The `map` function accepts a callback function returning the modified hit object. For example, we can use `map` to rename the hit IDs:

```
>>> for hit in blast_record[:5]:
...     print(hit.target.id)
...
gi|301171322|ref|NR_035857.1|
gi|262205330|ref|NR_030198.1|
gi|301171447|ref|NR_035871.1|
gi|262205298|ref|NR_030190.1|
gi|270132717|ref|NR_032716.1|
>>> import copy
>>> original_blast_record = copy.deepcopy(blast_record)
>>> def map_func(hit):
```

(continues on next page)

(continued from previous page)

```
...     # renames "gi|301171322|ref|NR_035857.1|" to "NR_035857.1"
...     hit.target.id = hit.target.id.split("|")[3]
...     return hit
...
>>> blast_record[:] = map(map_func, blast_record)
>>> for hit in blast_record[:5]:
...     print(hit.target.id)
...
NR_035857.1
NR_030198.1
NR_035871.1
NR_030190.1
NR_032716.1
>>> for hit in original_blast_record[:5]:
...     print(hit.target.id)
...
gi|301171322|ref|NR_035857.1|
gi|262205330|ref|NR_030198.1|
gi|301171447|ref|NR_035871.1|
gi|262205298|ref|NR_030190.1|
gi|270132717|ref|NR_032716.1|
```

Note that in this example, `map_func` modifies the hit in-place. In contrast to sorting and filtering (see above), using `original_blast_record = blast_record[:]` is not sufficient to retain a copy of the unmodified BLAST record, as it creates a shallow copy of the BLAST record, consisting of pointers to the same `Hit` objects. Instead, we use `copy.deepcopy` to create a copy of the BLAST record in which each `Hit` object is duplicated.

10.5 Writing BLAST records

Use the `write` function in `Bio.Blast` to save BLAST records as an XML file. By default, the (DTD-based) XML format is used; you can also save the BLAST records in the (schema-based) XML2 format by using the `fmt="XML2"` argument to the `write` function.

```
>>> from Bio import Blast
>>> stream = Blast.qblast("blastn", "nt", "8332116")
>>> records = Blast.parse(stream)
>>> Blast.write(records, "my_qblast_output.xml")

or
```

```
>>> Blast.write(records, "my_qblast_output.xml", fmt="XML2")
```

In this example, we could have saved the data returned by `Blast.qblast` directly to an XML file (see section *Saving BLAST results*). However, by parsing the data returned by qblast into records, we can sort or filter the BLAST records before saving them. For example, we may be interested only in BLAST HSPs with a positive score of at least 400:

```
>>> filter_func = lambda hsp: hsp.annotations["positive"] >= 400
>>> for hit in records[0]:
...     hit[:] = filter(filter_func, hit)
...
>>> Blast.write(records, "my_qblast_output_selected.xml")
```

Instead of a file name, the second argument to `Blast.write` can also be a file stream. In that case, the stream must be opened in binary format for writing:

```
>>> with open("my_qblast_output.xml", "wb") as stream:
...     Blast.write(records, stream)
...
```

10.6 Dealing with PSI-BLAST

You can run the standalone version of PSI-BLAST (`psiblast`) directly from the command line or using python's `subprocess` module.

At the time of writing, the NCBI do not appear to support tools running a PSI-BLAST search via the internet.

Note that the `Bio.Blast` parser can read the XML output from current versions of PSI-BLAST, but information like which sequences in each iteration is new or reused isn't present in the XML file.

10.7 Dealing with RPS-BLAST

You can run the standalone version of RPS-BLAST (`rpsblast`) directly from the command line or using python's `subprocess` module.

At the time of writing, the NCBI do not appear to support tools running an RPS-BLAST search via the internet.

You can use the `Bio.Blast` parser to read the XML output from current versions of RPS-BLAST.

CHAPTER

ELEVEN

BLAST (OLD)

Hey, everybody loves BLAST right? I mean, geez, how can it get any easier to do comparisons between one of your sequences and every other sequence in the known world? But, of course, this section isn't about how cool BLAST is, since we already know that. It is about the problem with BLAST – it can be really difficult to deal with the volume of data generated by large runs, and to automate BLAST runs in general.

Fortunately, the Biopython folks know this only too well, so they've developed lots of tools for dealing with BLAST and making things much easier. This section details how to use these tools and do useful things with them.

Dealing with BLAST can be split up into two steps, both of which can be done from within Biopython. Firstly, running BLAST for your query sequence(s), and getting some output. Secondly, parsing the BLAST output in Python for further analysis.

Your first introduction to running BLAST was probably via the NCBI web-service. In fact, there are lots of ways you can run BLAST, which can be categorized in several ways. The most important distinction is running BLAST locally (on your own machine), and running BLAST remotely (on another machine, typically the NCBI servers). We're going to start this chapter by invoking the NCBI online BLAST service from within a Python script.

NOTE: The following Chapter *BLAST and other sequence search tools* describes `Bio.SearchIO`. We intend this to ultimately replace the older `Bio.Blast` module, as it provides a more general framework handling other related sequence searching tools as well. However, for now you can use either that or the older `Bio.Blast` module for dealing with NCBI BLAST.

11.1 Running BLAST over the Internet

We use the function `qblast()` in the `Bio.Blast.NCBIWWW` module to call the online version of BLAST. This has three non-optional arguments:

- The first argument is the blast program to use for the search, as a lower case string. The options and descriptions of the programs are available at https://blast.ncbi.nlm.nih.gov/Blast.cgi. Currently `qblast` only works with blastn, blastp, blastx, tblast and tblastx.
- The second argument specifies the databases to search against. Again, the options for this are available on the NCBI Guide to BLAST https://blast.ncbi.nlm.nih.gov/doc/blast-help/.
- The third argument is a string containing your query sequence. This can either be the sequence itself, the sequence in fasta format, or an identifier like a GI number.

The NCBI guidelines, from https://blast.ncbi.nlm.nih.gov/doc/blast-help/developerinfo.html#developerinfo state:

1. Do not contact the server more often than once every 10 seconds.
2. Do not poll for any single RID more often than once a minute.
3. Use the URL parameter email and tool, so that the NCBI can contact you if there is a problem.

4. Run scripts weekends or between 9 pm and 5 am Eastern time on weekdays if more than 50 searches will be submitted.

To fulfill the third point, one can set the `NCBIWWW.email` variable.

```
>>> from Bio.Blast import NCBIWWW
>>> NCBIWWW.email = "A.N.Other@example.com"
```

The `qblast` function also takes a number of other option arguments, which are basically analogous to the different parameters you can set on the BLAST web page. We'll just highlight a few of them here:

- The argument `url_base` sets the base URL for running BLAST over the internet. By default it connects to the NCBI, but one can use this to connect to an instance of NCBI BLAST running in the cloud. Please refer to the documentation for the `qblast` function for further details.
- The `qblast` function can return the BLAST results in various formats, which you can choose with the optional `format_type` keyword: `"HTML"`, `"Text"`, `"ASN.1"`, or `"XML"`. The default is `"XML"`, as that is the format expected by the parser, described in section *Parsing BLAST output* below.
- The argument `expect` sets the expectation or e-value threshold.

For more about the optional BLAST arguments, we refer you to the NCBI's own documentation, or that built into Biopython:

```
>>> from Bio.Blast import NCBIWWW
>>> help(NCBIWWW.qblast)
```

Note that the default settings on the NCBI BLAST website are not quite the same as the defaults on QBLAST. If you get different results, you'll need to check the parameters (e.g., the expectation value threshold and the gap values).

For example, if you have a nucleotide sequence you want to search against the nucleotide database (nt) using BLASTN, and you know the GI number of your query sequence, you can use:

```
>>> from Bio.Blast import NCBIWWW
>>> result_handle = NCBIWWW.qblast("blastn", "nt", "8332116")
```

Alternatively, if we have our query sequence already in a FASTA formatted file, we just need to open the file and read in this record as a string, and use that as the query argument:

```
>>> from Bio.Blast import NCBIWWW
>>> fasta_string = open("m_cold.fasta").read()
>>> result_handle = NCBIWWW.qblast("blastn", "nt", fasta_string)
```

We could also have read in the FASTA file as a `SeqRecord` and then supplied just the sequence itself:

```
>>> from Bio.Blast import NCBIWWW
>>> from Bio import SeqIO
>>> record = SeqIO.read("m_cold.fasta", format="fasta")
>>> result_handle = NCBIWWW.qblast("blastn", "nt", record.seq)
```

Supplying just the sequence means that BLAST will assign an identifier for your sequence automatically. You might prefer to use the `SeqRecord` object's format method to make a FASTA string (which will include the existing identifier):

```
>>> from Bio.Blast import NCBIWWW
>>> from Bio import SeqIO
>>> record = SeqIO.read("m_cold.fasta", format="fasta")
>>> result_handle = NCBIWWW.qblast("blastn", "nt", record.format("fasta"))
```

This approach makes more sense if you have your sequence(s) in a non-FASTA file format which you can extract using `Bio.SeqIO` (see Chapter *Sequence Input/Output*).

Whatever arguments you give the `qblast()` function, you should get back your results in a handle object (by default in XML format). The next step would be to parse the XML output into Python objects representing the search results (Section *Parsing BLAST output*), but you might want to save a local copy of the output file first. I find this especially useful when debugging my code that extracts info from the BLAST results (because re-running the online search is slow and wastes the NCBI computer time).

We need to be a bit careful since we can use `result_handle.read()` to read the BLAST output only once – calling `result_handle.read()` again returns an empty string.

```
>>> with open("my_blast.xml", "w") as out_handle:
...     out_handle.write(result_handle.read())
...
>>> result_handle.close()
```

After doing this, the results are in the file `my_blast.xml` and the original handle has had all its data extracted (so we closed it). However, the `parse` function of the BLAST parser (described in *Parsing BLAST output*) takes a file-handle-like object, so we can just open the saved file for input:

```
>>> result_handle = open("my_blast.xml")
```

Now that we've got the BLAST results back into a handle again, we are ready to do something with them, so this leads us right into the parsing section (see Section *Parsing BLAST output* below). You may want to jump ahead to that now ....

11.2 Running BLAST locally

11.2.1 Introduction

Running BLAST locally (as opposed to over the internet, see Section *Running BLAST over the Internet*) has at least major two advantages:

- Local BLAST may be faster than BLAST over the internet;
- Local BLAST allows you to make your own database to search for sequences against.

Dealing with proprietary or unpublished sequence data can be another reason to run BLAST locally. You may not be allowed to redistribute the sequences, so submitting them to the NCBI as a BLAST query would not be an option.

Unfortunately, there are some major drawbacks too – installing all the bits and getting it setup right takes some effort:

- Local BLAST requires command line tools to be installed.
- Local BLAST requires (large) BLAST databases to be setup (and potentially kept up to date).

To further confuse matters there are several different BLAST packages available, and there are also other tools which can produce imitation BLAST output files, such as BLAT.

11.2.2 Standalone NCBI BLAST+

The "new" NCBI BLAST+ suite was released in 2009. This replaces the old NCBI "legacy" BLAST package (see below).

This section will show briefly how to use these tools from within Python. If you have already read or tried the alignment tool examples in Section *Alignment Tools* this should all seem quite straightforward. First, we construct a command line string (as you would type in at the command line prompt if running standalone BLAST by hand). Then we can execute this command from within Python.

For example, taking a FASTA file of gene nucleotide sequences, you might want to run a BLASTX (translation) search against the non-redundant (NR) protein database. Assuming you (or your systems administrator) has downloaded and installed the NR database, you might run:

```
$ blastx -query opuntia.fasta -db nr -out opuntia.xml -evalue 0.001 -outfmt 5
```

This should run BLASTX against the NR database, using an expectation cut-off value of 0.001 and produce XML output to the specified file (which we can then parse). On my computer this takes about six minutes - a good reason to save the output to a file so you can repeat any analysis as needed.

From within python we can use the `subprocess` module to build the command line string, and run it:

```
>>> import subprocess
>>> cmd = "blastx -query opuntia.fasta -db nr -out opuntia.xml"
>>> cmd += " -evalue 0.001 -outfmt 5"
>>> subprocess.run(cmd, shell=True)
```

In this example there shouldn't be any output from BLASTX to the terminal. You may want to check the output file `opuntia.xml` has been created.

As you may recall from earlier examples in the tutorial, the `opuntia.fasta` contains seven sequences, so the BLAST XML output should contain multiple results. Therefore use `Bio.Blast.NCBIXML.parse()` to parse it as described below in Section *Parsing BLAST output*.

11.2.3 Other versions of BLAST

NCBI BLAST+ (written in C++) was first released in 2009 as a replacement for the original NCBI "legacy" BLAST (written in C) which is no longer being updated. There were a lot of changes – the old version had a single core command line tool `blastall` which covered multiple different BLAST search types (which are now separate commands in BLAST+), and all the command line options were renamed. Biopython's wrappers for the NCBI "legacy" BLAST tools have been deprecated and will be removed in a future release. To try to avoid confusion, we do not cover calling these old tools from Biopython in this tutorial.

You may also come across Washington University BLAST (WU-BLAST), and its successor, Advanced Biocomputing BLAST (AB-BLAST, released in 2009, not free/open source). These packages include the command line tools `wu-blastall` and `ab-blastall`, which mimicked `blastall` from the NCBI "legacy" BLAST suite. Biopython does not currently provide wrappers for calling these tools, but should be able to parse any NCBI compatible output from them.

11.3 Parsing BLAST output

As mentioned above, BLAST can generate output in various formats, such as XML, HTML, and plain text. Originally, Biopython had parsers for BLAST plain text and HTML output, as these were the only output formats offered at the time. Unfortunately, the BLAST output in these formats kept changing, each time breaking the Biopython parsers. Our HTML BLAST parser has been removed, while the deprecated plain text BLAST parser is now only available via `Bio.SearchIO`. Use it at your own risk, it may or may not work, depending on which BLAST version you're using.

As keeping up with changes in BLAST became a hopeless endeavor, especially with users running different BLAST versions, we now recommend to parse the output in XML format, which can be generated by recent versions of BLAST. Not only is the XML output more stable than the plain text and HTML output, it is also much easier to parse automatically, making Biopython a whole lot more stable.

You can get BLAST output in XML format in various ways. For the parser, it doesn't matter how the output was generated, as long as it is in the XML format.

- You can use Biopython to run BLAST over the internet, as described in section *Running BLAST over the Internet*.
- You can use Biopython to run BLAST locally, as described in section *Running BLAST locally*.
- You can do the BLAST search yourself on the NCBI site through your web browser, and then save the results. You need to choose XML as the format in which to receive the results, and save the final BLAST page you get (you know, the one with all of the interesting results!) to a file.
- You can also run BLAST locally without using Biopython, and save the output in a file. Again, you need to choose XML as the format in which to receive the results.

The important point is that you do not have to use Biopython scripts to fetch the data in order to be able to parse it. Doing things in one of these ways, you then need to get a handle to the results. In Python, a handle is just a nice general way of describing input to any info source so that the info can be retrieved using `read()` and `readline()` functions (see Section *What the heck is a handle?*).

If you followed the code above for interacting with BLAST through a script, then you already have `result_handle`, the handle to the BLAST results. For example, using a GI number to do an online search:

```
>>> from Bio.Blast import NCBIWWW
>>> result_handle = NCBIWWW.qblast("blastn", "nt", "8332116")
```

If instead you ran BLAST some other way, and have the BLAST output (in XML format) in the file `my_blast.xml`, all you need to do is to open the file for reading:

```
>>> result_handle = open("my_blast.xml")
```

Now that we've got a handle, we are ready to parse the output. The code to parse it is really quite small. If you expect a single BLAST result (i.e., you used a single query):

```
>>> from Bio.Blast import NCBIXML
>>> blast_record = NCBIXML.read(result_handle)
```

or, if you have lots of results (i.e., multiple query sequences):

```
>>> from Bio.Blast import NCBIXML
>>> blast_records = NCBIXML.parse(result_handle)
```

Just like `Bio.SeqIO` and `Bio.Align` (see Chapters *Sequence Input/Output* and *Sequence alignments*), we have a pair of input functions, `read` and `parse`, where `read` is for when you have exactly one object, and `parse` is an iterator for when you can have lots of objects – but instead of getting `SeqRecord` or `MultipleSeqAlignment` objects, we get BLAST record objects.

To be able to handle the situation where the BLAST file may be huge, containing thousands of results, `NCBIXML.parse()` returns an iterator. In plain English, an iterator allows you to step through the BLAST output, retrieving BLAST records one by one for each BLAST search result:

```
>>> from Bio.Blast import NCBIXML
>>> blast_records = NCBIXML.parse(result_handle)
>>> blast_record = next(blast_records)
# ... do something with blast_record
>>> blast_record = next(blast_records)
# ... do something with blast_record
>>> blast_record = next(blast_records)
# ... do something with blast_record
>>> blast_record = next(blast_records)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
# No further records
```

Or, you can use a `for`-loop:

```
>>> for blast_record in blast_records:
...     pass  # Do something with blast_record
...
```

Note though that you can step through the BLAST records only once. Usually, from each BLAST record you would save the information that you are interested in. If you want to save all returned BLAST records, you can convert the iterator into a list:

```
>>> blast_records = list(blast_records)
```

Now you can access each BLAST record in the list with an index as usual. If your BLAST file is huge though, you may run into memory problems trying to save them all in a list.

Usually, you'll be running one BLAST search at a time. Then, all you need to do is to pick up the first (and only) BLAST record in `blast_records`:

```
>>> from Bio.Blast import NCBIXML
>>> blast_records = NCBIXML.parse(result_handle)
>>> blast_record = next(blast_records)
```

or more elegantly:

```
>>> from Bio.Blast import NCBIXML
>>> blast_record = NCBIXML.read(result_handle)
```

I guess by now you're wondering what is in a BLAST record.

11.4 The BLAST record class

A BLAST Record contains everything you might ever want to extract from the BLAST output. Right now we'll just show an example of how to get some info out of the BLAST report, but if you want something in particular that is not described here, look at the info on the record class in detail, and take a gander into the code or automatically generated documentation – the docstrings have lots of good info about what is stored in each piece of information.

To continue with our example, let's just print out some summary info about all hits in our blast report greater than a particular threshold. The following code does this:

```
>>> E_VALUE_THRESH = 0.04

>>> for alignment in blast_record.alignments:
...     for hsp in alignment.hsps:
...         if hsp.expect < E_VALUE_THRESH:
...             print("****Alignment****")
...             print("sequence:", alignment.title)
...             print("length:", alignment.length)
...             print("e value:", hsp.expect)
...             print(hsp.query[0:75] + "...")
...             print(hsp.match[0:75] + "...")
...             print(hsp.sbjct[0:75] + "...")
...
```

This will print out summary reports like the following:

```
****Alignment****
sequence: >gb|AF283004.1|AF283004 Arabidopsis thaliana cold acclimation protein WCOR413-
↪like protein
alpha form mRNA, complete cds
length: 783
e value: 0.034
tacttgttgatattggatcgaacaaactggagaaccaacatgctcacgtcacttttagtcccttacatattcctc...
||||||||| | |||||||||| || ||||  || || |||||||| |||||| |  | |||||||| ||| ||...
tacttgttggtgttggatcgaaccaattggaagacgaatatgctcacatcacttctcattccttacatcttcttc...
```

Basically, you can do anything you want to with the info in the BLAST report once you have parsed it. This will, of course, depend on what you want to use it for, but hopefully this helps you get started on doing what you need to do!

An important consideration for extracting information from a BLAST report is the type of objects that the information is stored in. In Biopython, the parsers return `Record` objects, either `Blast` or `PSIBlast` depending on what you are parsing. These objects are defined in `Bio.Blast.Record` and are quite complete.

Figures *Class diagram for the Blast Record class representing a BLAST report.* and *Class diagram for the PSIBlast Record class.* and are my attempts at UML class diagrams for the `Blast` and `PSIBlast` record classes. The PSIBlast record object is similar, but has support for the rounds that are used in the iteration steps of PSIBlast.

If you are good at UML and see mistakes/improvements that can be made, please let me know.

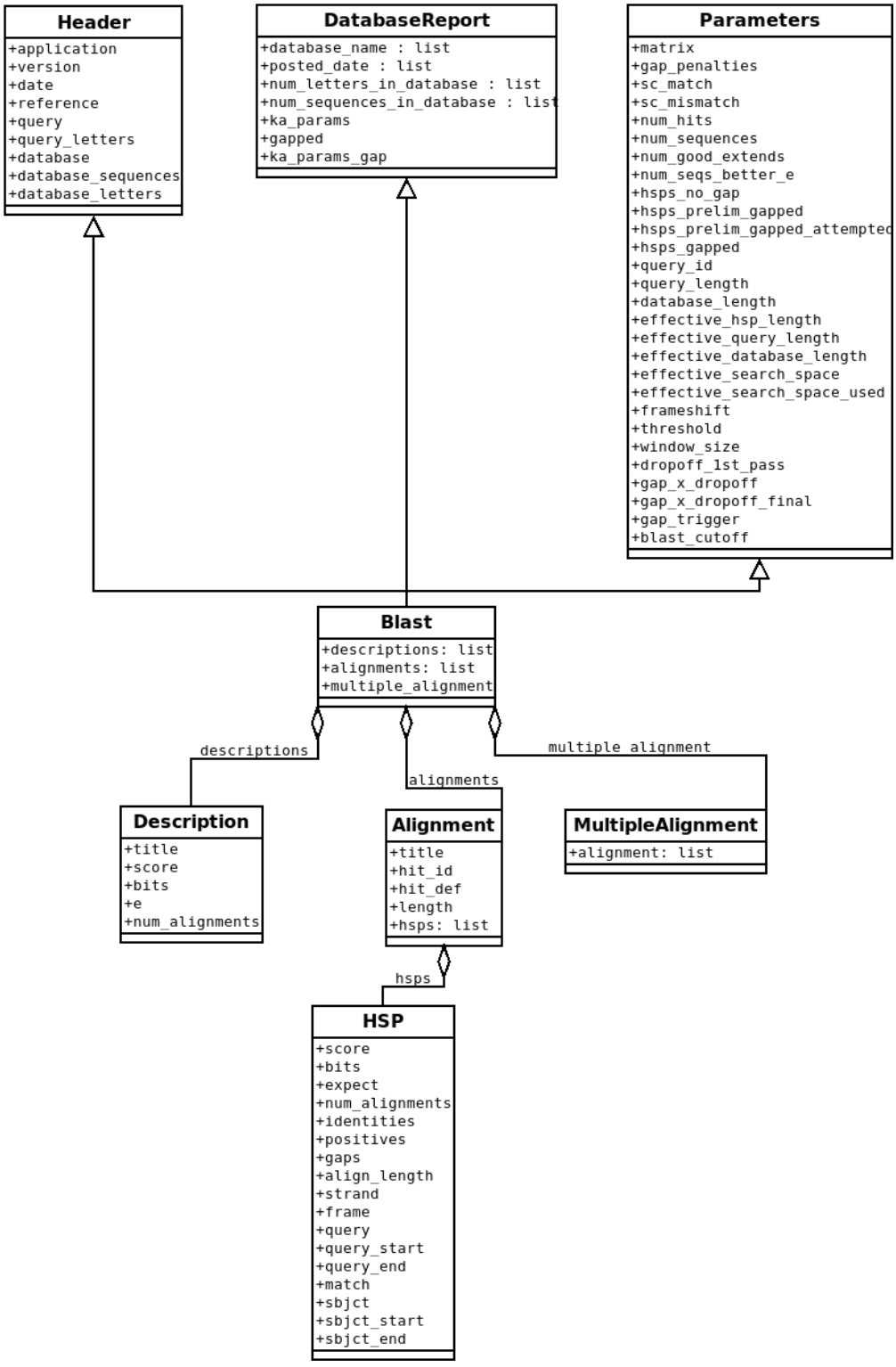


Fig. 1: Class diagram for the Blast Record class representing a BLAST report.

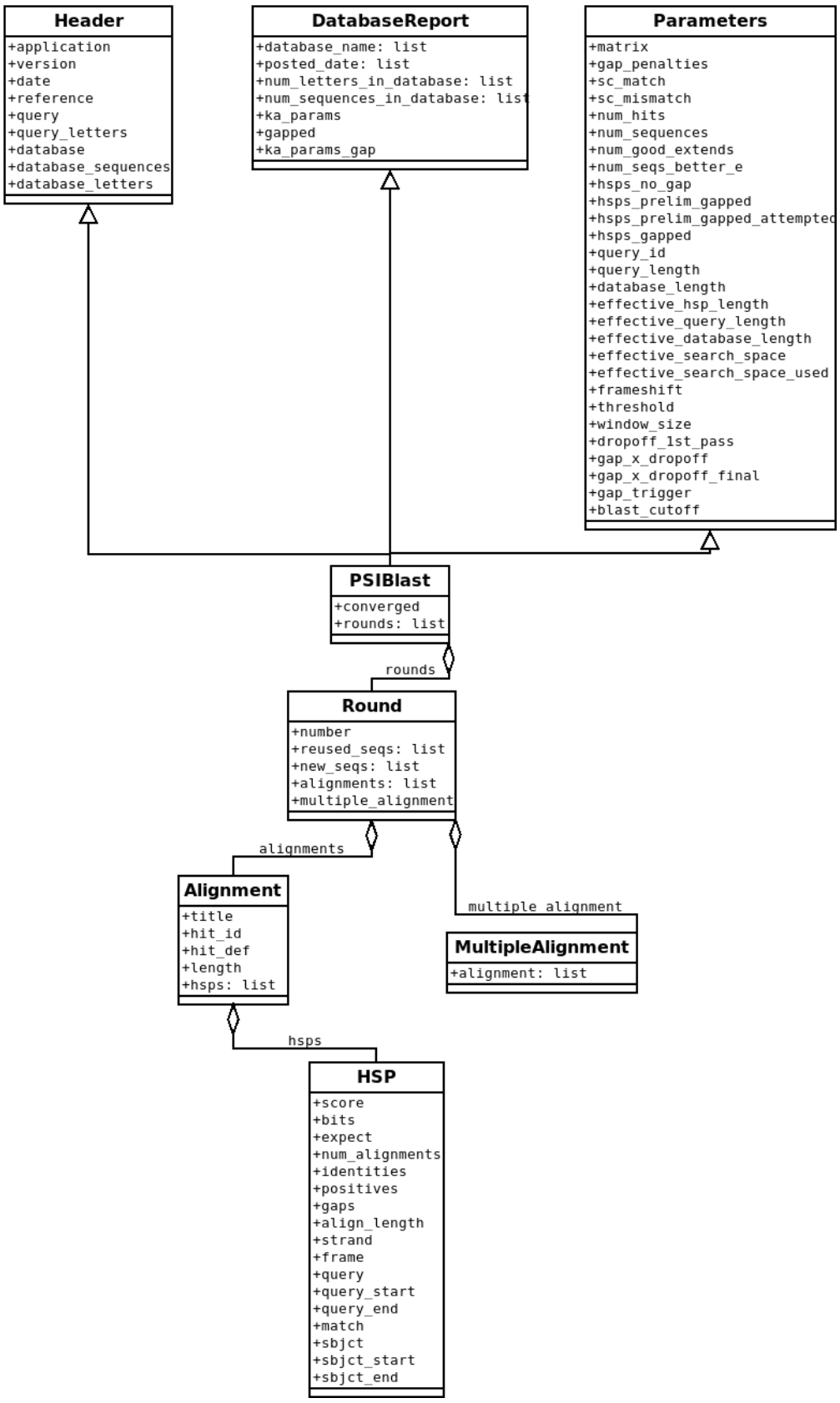


Fig. 2: Class diagram for the PSIBlast Record class.

11.5 Dealing with PSI-BLAST

You can run the standalone version of PSI-BLAST (the legacy NCBI command line tool `blastpgp`, or its replacement `psiblast`) directly from the command line or using python's `subprocess` module.

At the time of writing, the NCBI do not appear to support tools running a PSI-BLAST search via the internet.

Note that the `Bio.Blast.NCBIXML` parser can read the XML output from current versions of PSI-BLAST, but information like which sequences in each iteration is new or reused isn't present in the XML file.

11.6 Dealing with RPS-BLAST

You can run the standalone version of RPS-BLAST (either the legacy NCBI command line tool `rpsblast`, or its replacement with the same name) directly from the command line or using python's `subprocess` module.

At the time of writing, the NCBI do not appear to support tools running an RPS-BLAST search via the internet.

You can use the `Bio.Blast.NCBIXML` parser to read the XML output from current versions of RPS-BLAST.

CHAPTER

TWELVE

BLAST AND OTHER SEQUENCE SEARCH TOOLS

Biological sequence identification is an integral part of bioinformatics. Several tools are available for this, each with their own algorithms and approaches, such as BLAST (arguably the most popular), FASTA, HMMER, and many more. In general, these tools usually use your sequence to search a database of potential matches. With the growing number of known sequences (hence the growing number of potential matches), interpreting the results becomes increasingly hard as there could be hundreds or even thousands of potential matches. Naturally, manual interpretation of these searches' results is out of the question. Moreover, you often need to work with several sequence search tools, each with its own statistics, conventions, and output format. Imagine how daunting it would be when you need to work with multiple sequences using multiple search tools.

We know this too well ourselves, which is why we created the `Bio.SearchIO` submodule in Biopython. `Bio.SearchIO` allows you to extract information from your search results in a convenient way, while also dealing with the different standards and conventions used by different search tools. The name `SearchIO` is a homage to BioPerl's module of the same name.

In this chapter, we'll go through the main features of `Bio.SearchIO` to show what it can do for you. We'll use two popular search tools along the way: BLAST and BLAT. They are used merely for illustrative purposes, and you should be able to adapt the workflow to any other search tools supported by `Bio.SearchIO` in a breeze. You're very welcome to follow along with the search output files we'll be using. The BLAST output file can be downloaded here, and the BLAT output file here or are included with the Biopython source code under the `Doc/examples/` folder. Both output files were generated using this sequence:

```
>mystery_seq
CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
```

The BLAST result is an XML file generated using `blastn` against the NCBI `refseq_rna` database. For BLAT, the sequence database was the February 2009 `hg19` human genome draft and the output format is PSL.

We'll start from an introduction to the `Bio.SearchIO` object model. The model is the representation of your search results, thus it is core to `Bio.SearchIO` itself. After that, we'll check out the main functions in `Bio.SearchIO` that you may often use.

Now that we're all set, let's go to the first step: introducing the core object model.

12.1 The SearchIO object model

Despite the wildly differing output styles among many sequence search tools, it turns out that their underlying concept is similar:

- The output file may contain results from one or more search queries.
- In each search query, you will see one or more hits from the given search database.
- In each database hit, you will see one or more regions containing the actual sequence alignment between your query sequence and the database sequence.
- Some programs like BLAT or Exonerate may further split these regions into several alignment fragments (or blocks in BLAT and possibly exons in exonerate). This is not something you always see, as programs like BLAST and HMMER do not do this.

Realizing this generality, we decided use it as base for creating the `Bio.SearchIO` object model. The object model consists of a nested hierarchy of Python objects, each one representing one concept outlined above. These objects are:

- `QueryResult`, to represent a single search query.
- `Hit`, to represent a single database hit. `Hit` objects are contained within `QueryResult` and in each `QueryResult` there is zero or more `Hit` objects.
- `HSP` (short for high-scoring pair), to represent region(s) of significant alignments between query and hit sequences. `HSP` objects are contained within `Hit` objects and each `Hit` has one or more `HSP` objects.
- `HSPFragment`, to represent a single contiguous alignment between query and hit sequences. `HSPFragment` objects are contained within `HSP` objects. Most sequence search tools like BLAST and HMMER unify `HSP` and `HSPFragment` objects as each `HSP` will only have a single `HSPFragment`. However there are tools like BLAT and Exonerate that produce `HSP` containing multiple `HSPFragment`. Don't worry if this seems a tad confusing now, we'll elaborate more on these two objects later on.

These four objects are the ones you will interact with when you use `Bio.SearchIO`. They are created using one of the main `Bio.SearchIO` methods: `read`, `parse`, `index`, or `index_db`. The details of these methods are provided in later sections. For this section, we'll only be using read and parse. These functions behave similarly to their `Bio.SeqIO` and `Bio.AlignIO` counterparts:

- `read` is used for search output files with a single query and returns a `QueryResult` object
- `parse` is used for search output files with multiple queries and returns a generator that yields `QueryResult` objects

With that settled, let's start probing each `Bio.SearchIO` object, beginning with `QueryResult`.

12.1.1 QueryResult

The QueryResult object represents a single search query and contains zero or more Hit objects. Let's see what it looks like using the BLAST file we have:

```
>>> from Bio import SearchIO
>>> blast_qresult = SearchIO.read("my_blast.xml", "blast-xml")
>>> print(blast_qresult)
Program: blastn (2.2.27+)
  Query: 42291 (61)
         mystery_seq
 Target: refseq_rna
   Hits: ----  -----  ------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
     #  # HSP  ID + description
----  -----  ------------------------------------------------------------
   0      1  gi|262205317|ref|NR_030195.1|  Homo sapiens microRNA 52...
   1      1  gi|301171311|ref|NR_035856.1|  Pan troglodytes microRNA...
   2      1  gi|270133242|ref|NR_032573.1|  Macaca mulatta microRNA ...
   3      2  gi|301171322|ref|NR_035857.1|  Pan troglodytes microRNA...
   4      1  gi|301171267|ref|NR_035851.1|  Pan troglodytes microRNA...
   5      2  gi|262205330|ref|NR_030198.1|  Homo sapiens microRNA 52...
   6      1  gi|262205302|ref|NR_030191.1|  Homo sapiens microRNA 51...
   7      1  gi|301171259|ref|NR_035850.1|  Pan troglodytes microRNA...
   8      1  gi|262205451|ref|NR_030222.1|  Homo sapiens microRNA 51...
   9      2  gi|301171447|ref|NR_035871.1|  Pan troglodytes microRNA...
  10      1  gi|301171276|ref|NR_035852.1|  Pan troglodytes microRNA...
  11      1  gi|262205290|ref|NR_030188.1|  Homo sapiens microRNA 51...
  12      1  gi|301171354|ref|NR_035860.1|  Pan troglodytes microRNA...
  13      1  gi|262205281|ref|NR_030186.1|  Homo sapiens microRNA 52...
  14      2  gi|262205298|ref|NR_030190.1|  Homo sapiens microRNA 52...
  15      1  gi|301171394|ref|NR_035865.1|  Pan troglodytes microRNA...
  16      1  gi|262205429|ref|NR_030218.1|  Homo sapiens microRNA 51...
  17      1  gi|262205423|ref|NR_030217.1|  Homo sapiens microRNA 52...
  18      1  gi|301171401|ref|NR_035866.1|  Pan troglodytes microRNA...
  19      1  gi|270133247|ref|NR_032574.1|  Macaca mulatta microRNA ...
  20      1  gi|262205309|ref|NR_030193.1|  Homo sapiens microRNA 52...
  21      2  gi|270132717|ref|NR_032716.1|  Macaca mulatta microRNA ...
  22      2  gi|301171437|ref|NR_035870.1|  Pan troglodytes microRNA...
  23      2  gi|270133306|ref|NR_032587.1|  Macaca mulatta microRNA ...
  24      2  gi|301171428|ref|NR_035869.1|  Pan troglodytes microRNA...
  25      1  gi|301171211|ref|NR_035845.1|  Pan troglodytes microRNA...
  26      2  gi|301171153|ref|NR_035838.1|  Pan troglodytes microRNA...
  27      2  gi|301171146|ref|NR_035837.1|  Pan troglodytes microRNA...
  28      2  gi|270133254|ref|NR_032575.1|  Macaca mulatta microRNA ...
  29      2  gi|262205445|ref|NR_030221.1|  Homo sapiens microRNA 51...
 ~~~
  97      1  gi|356517317|ref|XM_003527287.1|  PREDICTED: Glycine ma...
  98      1  gi|297814701|ref|XM_002875188.1|  Arabidopsis lyrata su...
  99      1  gi|397513516|ref|XM_003827011.1|  PREDICTED: Pan panisc...
```

We've just begun to scratch the surface of the object model, but you can see that there's already some useful information. By invoking `print` on the `QueryResult` object, you can see:

- The program name and version (blastn version 2.2.27+)
- The query ID, description, and its sequence length (ID is 42291, description is 'mystery_seq', and it is 61 nucleotides long)
- The target database to search against (refseq_rna)
- A quick overview of the resulting hits. For our query sequence, there are 100 potential hits (numbered 0–99 in the table). For each hit, we can also see how many HSPs it contains, its ID, and a snippet of its description. Notice here that `Bio.SearchIO` truncates the hit table overview, by showing only hits numbered 0–29, and then 97–99.

Now let's check our BLAT results using the same procedure as above:

```
>>> blat_qresult = SearchIO.read("my_blat.psl", "blat-psl")
>>> print(blat_qresult)
Program: blat (<unknown version>)
  Query: mystery_seq (61)
         <unknown description>
 Target: <unknown target>
   Hits: ----  -----  ----------------------------------------------------------
            #  # HSP  ID + description
         ----  -----  ----------------------------------------------------------
            0     17  chr19  <unknown description>
```

You'll immediately notice that there are some differences. Some of these are caused by the way PSL format stores its details, as you'll see. The rest are caused by the genuine program and target database differences between our BLAST and BLAT searches:

- The program name and version. `Bio.SearchIO` knows that the program is BLAT, but in the output file there is no information regarding the program version so it defaults to '<unknown version>'.
- The query ID, description, and its sequence length. Notice here that these details are slightly different from the ones we saw in BLAST. The ID is 'mystery_seq' instead of 42991, there is no known description, but the query length is still 61. This is actually a difference introduced by the file formats themselves. BLAST sometimes creates its own query IDs and uses your original ID as the sequence description.
- The target database is not known, as it is not stated in the BLAT output file.
- And finally, the list of hits we have is completely different. Here, we see that our query sequence only hits the 'chr19' database entry, but in it we see 17 HSP regions. This should not be surprising however, given that we are using a different program, each with its own target database.

All the details you saw when invoking the `print` method can be accessed individually using Python's object attribute access notation (a.k.a. the dot notation). There are also other format-specific attributes that you can access using the same method.

```
>>> print("%s %s" % (blast_qresult.program, blast_qresult.version))
blastn 2.2.27+
>>> print("%s %s" % (blat_qresult.program, blat_qresult.version))
blat <unknown version>
>>> blast_qresult.param_evalue_threshold  # blast-xml specific
10.0
```

For a complete list of accessible attributes, you can check each format-specific documentation. e.g. *Bio.SearchIO.BlastIO* and *Bio.SearchIO.BlatIO*.

Having looked at using `print` on `QueryResult` objects, let's drill down deeper. What exactly is a `QueryResult`? In terms of Python objects, `QueryResult` is a hybrid between a list and a dictionary. In other words, it is a container object with all the convenient features of lists and dictionaries.

Like Python lists and dictionaries, `QueryResult` objects are iterable. Each iteration returns a `Hit` object:

```
>>> for hit in blast_qresult:
...     hit
...
Hit(id='gi|262205317|ref|NR_030195.1|', query_id='42291', 1 hsps)
Hit(id='gi|301171311|ref|NR_035856.1|', query_id='42291', 1 hsps)
Hit(id='gi|270133242|ref|NR_032573.1|', query_id='42291', 1 hsps)
Hit(id='gi|301171322|ref|NR_035857.1|', query_id='42291', 2 hsps)
```

(continues on next page)

(continued from previous page)

```
Hit(id='gi|301171267|ref|NR_035851.1|', query_id='42291', 1 hsps)
...
```

To check how many items (hits) a `QueryResult` has, you can simply invoke Python's `len` method:

```
>>> len(blast_qresult)
100
>>> len(blat_qresult)
1
```

Like Python lists, you can retrieve items (hits) from a `QueryResult` using the slice notation:

```
>>> blast_qresult[0]  # retrieves the top hit
Hit(id='gi|262205317|ref|NR_030195.1|', query_id='42291', 1 hsps)
>>> blast_qresult[-1]  # retrieves the last hit
Hit(id='gi|397513516|ref|XM_003827011.1|', query_id='42291', 1 hsps)
```

To retrieve multiple hits, you can slice `QueryResult` objects using the slice notation as well. In this case, the slice will return a new `QueryResult` object containing only the sliced hits:

```
>>> blast_slice = blast_qresult[:3]  # slices the first three hits
>>> print(blast_slice)
Program: blastn (2.2.27+)
  Query: 42291 (61)
         mystery_seq
 Target: refseq_rna
   Hits: ----  -----  ------------------------------------------------------------
            #  # HSP  ID + description
         ----  -----  ------------------------------------------------------------
            0      1  gi|262205317|ref|NR_030195.1|  Homo sapiens microRNA 52...
            1      1  gi|301171311|ref|NR_035856.1|  Pan troglodytes microRNA...
            2      1  gi|270133242|ref|NR_032573.1|  Macaca mulatta microRNA ...
```

Like Python dictionaries, you can also retrieve hits using the hit's ID. This is particularly useful if you know a given hit ID exists within a search query results:

```
>>> blast_qresult["gi|262205317|ref|NR_030195.1|"]
Hit(id='gi|262205317|ref|NR_030195.1|', query_id='42291', 1 hsps)
```

You can also get a full list of `Hit` objects using `hits` and a full list of `Hit` IDs using `hit_keys`:

```
>>> blast_qresult.hits
[...]       # list of all hits
>>> blast_qresult.hit_keys
[...]       # list of all hit IDs
```

What if you just want to check whether a particular hit is present in the query results? You can do a simple Python membership test using the `in` keyword:

```
>>> "gi|262205317|ref|NR_030195.1|" in blast_qresult
True
>>> "gi|262205317|ref|NR_030194.1|" in blast_qresult
False
```

Sometimes, knowing whether a hit is present is not enough; you also want to know the rank of the hit. Here, the `index` method comes to the rescue:

```
>>> blast_qresult.index("gi|301171437|ref|NR_035870.1|")
22
```

Remember that we're using Python's indexing style here, which is zero-based. This means our hit above is ranked at no. 23, not 22.

Also, note that the hit rank you see here is based on the native hit ordering present in the original search output file. Different search tools may order these hits based on different criteria.

If the native hit ordering doesn't suit your taste, you can use the `sort` method of the `QueryResult` object. It is very similar to Python's `list.sort` method, with the addition of an option to create a new sorted `QueryResult` object or not.

Here is an example of using `QueryResult.sort` to sort the hits based on each hit's full sequence length. For this particular sort, we'll set the `in_place` flag to `False` so that sorting will return a new `QueryResult` object and leave our initial object unsorted. We'll also set the `reverse` flag to `True` so that we sort in descending order.

```
>>> for hit in blast_qresult[:5]:  # id and sequence length of the first five hits
...     print("%s %i" % (hit.id, hit.seq_len))
...
gi|262205317|ref|NR_030195.1| 61
gi|301171311|ref|NR_035856.1| 60
gi|270133242|ref|NR_032573.1| 85
gi|301171322|ref|NR_035857.1| 86
gi|301171267|ref|NR_035851.1| 80

>>> sort_key = lambda hit: hit.seq_len
>>> sorted_qresult = blast_qresult.sort(key=sort_key, reverse=True, in_place=False)
>>> for hit in sorted_qresult[:5]:
...     print("%s %i" % (hit.id, hit.seq_len))
...
gi|397513516|ref|XM_003827011.1| 6002
gi|390332045|ref|XM_776818.2| 4082
gi|390332043|ref|XM_003723358.1| 4079
gi|356517317|ref|XM_003527287.1| 3251
gi|356543101|ref|XM_003539954.1| 2936
```

The advantage of having the `in_place` flag here is that we're preserving the native ordering, so we may use it again later. You should note that this is not the default behavior of `QueryResult.sort`, however, which is why we needed to set the `in_place` flag to `True` explicitly.

At this point, you've known enough about `QueryResult` objects to make it work for you. But before we go on to the next object in the `Bio.SearchIO` model, let's take a look at two more sets of methods that could make it even easier to work with `QueryResult` objects: the `filter` and `map` methods.

If you're familiar with Python's list comprehensions, generator expressions or the built-in `filter` and `map` functions, you'll know how useful they are for working with list-like objects (if you're not, check them out!). You can use these built-in methods to manipulate `QueryResult` objects, but you'll end up with regular Python lists and lose the ability to do more interesting manipulations.

That's why, `QueryResult` objects provide its own flavor of `filter` and `map` methods. Analogous to `filter`, there are `hit_filter` and `hsp_filter` methods. As their name implies, these methods filter its `QueryResult` object either on its `Hit` objects or `HSP` objects. Similarly, analogous to `map`, `QueryResult` objects also provide the `hit_map` and `hsp_map` methods. These methods apply a given function to all hits or HSPs in a `QueryResult` object, respectively.

Let's see these methods in action, beginning with `hit_filter`. This method accepts a callback function that checks whether a given `Hit` object passes the condition you set or not. In other words, the function must accept as its argument a single `Hit` object and returns `True` or `False`.

Here is an example of using `hit_filter` to filter out `Hit` objects that only have one HSP:

```
>>> filter_func = lambda hit: len(hit.hsps) > 1  # the callback function
>>> len(blast_qresult)  # no. of hits before filtering
100
>>> filtered_qresult = blast_qresult.hit_filter(filter_func)
>>> len(filtered_qresult)  # no. of hits after filtering
37
>>> for hit in filtered_qresult[:5]:  # quick check for the hit lengths
...     print("%s %i" % (hit.id, len(hit.hsps)))
...
gi|301171322|ref|NR_035857.1| 2
gi|262205330|ref|NR_030198.1| 2
gi|301171447|ref|NR_035871.1| 2
gi|262205298|ref|NR_030190.1| 2
gi|270132717|ref|NR_032716.1| 2
```

`hsp_filter` works the same as `hit_filter`, only instead of looking at the `Hit` objects, it performs filtering on the `HSP` objects in each hits.

As for the `map` methods, they too accept a callback function as their arguments. However, instead of returning `True` or `False`, the callback function must return the modified `Hit` or `HSP` object (depending on whether you're using `hit_map` or `hsp_map`).

Let's see an example where we're using `hit_map` to rename the hit IDs:

```
>>> def map_func(hit):
...     # renames "gi|301171322|ref|NR_035857.1|" to "NR_035857.1"
...     hit.id = hit.id.split("|")[3]
...     return hit
...
>>> mapped_qresult = blast_qresult.hit_map(map_func)
>>> for hit in mapped_qresult[:5]:
...     print(hit.id)
...
NR_030195.1
NR_035856.1
NR_032573.1
NR_035857.1
NR_035851.1
```

Again, `hsp_map` works the same as `hit_map`, but on `HSP` objects instead of `Hit` objects.

12.1.2 Hit

`Hit` objects represent all query results from a single database entry. They are the second-level container in the `Bio.SearchIO` object hierarchy. You've seen that they are contained by `QueryResult` objects, but they themselves contain `HSP` objects.

Let's see what they look like, beginning with our BLAST search:

```
>>> from Bio import SearchIO
>>> blast_qresult = SearchIO.read("my_blast.xml", "blast-xml")
>>> blast_hit = blast_qresult[3]  # fourth hit from the query result
>>> print(blast_hit)
Query: 42291
       mystery_seq
  Hit: gi|301171322|ref|NR_035857.1| (86)
       Pan troglodytes microRNA mir-520c (MIR520C), microRNA
 HSPs: ----  --------  ---------  ------  ---------------  ----------------------
          #   E-value  Bit score    Span      Query range              Hit range
       ----  --------  ---------  ------  ---------------  ----------------------
          0   8.9e-20     100.47      60           [1:61]                [13:73]
          1   3.3e-06      55.39      60           [0:60]                [13:73]
```

You see that we've got the essentials covered here:

- The query ID and description is present. A hit is always tied to a query, so we want to keep track of the originating query as well. These values can be accessed from a hit using the `query_id` and `query_description` attributes.
- We also have the unique hit ID, description, and full sequence lengths. They can be accessed using `id`, `description`, and `seq_len`, respectively.
- Finally, there's a table containing quick information about the HSPs this hit contains. In each row, we've got the important HSP details listed: the HSP index, its e-value, its bit score, its span (the alignment length including gaps), its query coordinates, and its hit coordinates.

Now let's contrast this with the BLAT search. Remember that in the BLAT search we had one hit with 17 HSPs.

```
>>> blat_qresult = SearchIO.read("my_blat.psl", "blat-psl")
>>> blat_hit = blat_qresult[0]  # the only hit
>>> print(blat_hit)
Query: mystery_seq
       <unknown description>
  Hit: chr19 (59128983)
       <unknown description>
 HSPs: ----  --------  ---------  ------  ---------------  ----------------------
          #   E-value  Bit score    Span      Query range              Hit range
       ----  --------  ---------  ------  ---------------  ----------------------
          0         ?          ?       ?           [0:61]    [54204480:54204541]
          1         ?          ?       ?           [0:61]    [54233104:54264463]
          2         ?          ?       ?           [0:61]    [54254477:54260071]
          3         ?          ?       ?           [1:61]    [54210720:54210780]
          4         ?          ?       ?           [0:60]    [54198476:54198536]
          5         ?          ?       ?           [0:61]    [54265610:54265671]
          6         ?          ?       ?           [0:61]    [54238143:54240175]
          7         ?          ?       ?           [0:60]    [54189735:54189795]
          8         ?          ?       ?           [0:61]    [54185425:54185486]
          9         ?          ?       ?           [0:60]    [54197657:54197717]
```

(continues on next page)

(continued from previous page)

```
        10        ?         ?       ?          [0:61]    [54255662:54255723]
        11        ?         ?       ?          [0:61]    [54201651:54201712]
        12        ?         ?       ?          [8:60]    [54206009:54206061]
        13        ?         ?       ?         [10:61]    [54178987:54179038]
        14        ?         ?       ?          [8:61]    [54212018:54212071]
        15        ?         ?       ?          [8:51]    [54234278:54234321]
        16        ?         ?       ?          [8:61]    [54238143:54238196]
```

Here, we've got a similar level of detail as with the BLAST hit we saw earlier. There are some differences worth explaining, though:

- The e-value and bit score column values. As BLAT HSPs do not have e-values and bit scores, the display defaults to '?'.
- What about the span column? The span values is meant to display the complete alignment length, which consists of all residues and any gaps that may be present. The PSL format do not have this information readily available and `Bio.SearchIO` does not attempt to try guess what it is, so we get a '?' similar to the e-value and bit score columns.

In terms of Python objects, `Hit` behaves almost the same as Python lists, but contain `HSP` objects exclusively. If you're familiar with lists, you should encounter no difficulties working with the `Hit` object.

Just like Python lists, `Hit` objects are iterable, and each iteration returns one `HSP` object it contains:

```
>>> for hsp in blast_hit:
...     hsp
...
HSP(hit_id='gi|301171322|ref|NR_035857.1|', query_id='42291', 1 fragments)
HSP(hit_id='gi|301171322|ref|NR_035857.1|', query_id='42291', 1 fragments)
```

You can invoke `len` on a `Hit` to see how many `HSP` objects it has:

```
>>> len(blast_hit)
2
>>> len(blat_hit)
17
```

You can use the slice notation on `Hit` objects, whether to retrieve single `HSP` or multiple `HSP` objects. Like `QueryResult`, if you slice for multiple `HSP`, a new `Hit` object will be returned containing only the sliced `HSP` objects:

```
>>> blat_hit[0]  # retrieve single items
HSP(hit_id='chr19', query_id='mystery_seq', 1 fragments)
>>> sliced_hit = blat_hit[4:9]  # retrieve multiple items
>>> len(sliced_hit)
5
>>> print(sliced_hit)
Query: mystery_seq
       <unknown description>
  Hit: chr19 (59128983)
       <unknown description>
 HSPs: ----  --------  ---------  ------  ---------------  ----------------------
         #   E-value  Bit score    Span      Query range              Hit range
       ----  --------  ---------  ------  ---------------  ----------------------
         0         ?          ?       ?           [0:60]    [54198476:54198536]
```

(continues on next page)

(continued from previous page)

```
          1          ?          ?          ?          [0:61]     [54265610:54265671]
          2          ?          ?          ?          [0:61]     [54238143:54240175]
          3          ?          ?          ?          [0:60]     [54189735:54189795]
          4          ?          ?          ?          [0:61]     [54185425:54185486]
```

You can also sort the HSP inside a `Hit`, using the exact same arguments like the sort method you saw in the `QueryResult` object.

Finally, there are also the `filter` and `map` methods you can use on `Hit` objects. Unlike in the `QueryResult` object, `Hit` objects only have one variant of `filter` (`Hit.filter`) and one variant of `map` (`Hit.map`). Both of `Hit.filter` and `Hit.map` work on the HSP objects a `Hit` has.

12.1.3 HSP

HSP (high-scoring pair) represents region(s) in the hit sequence that contains significant alignment(s) to the query sequence. It contains the actual match between your query sequence and a database entry. As this match is determined by the sequence search tool's algorithms, the HSP object contains the bulk of the statistics computed by the search tool. This also makes the distinction between HSP objects from different search tools more apparent compared to the differences you've seen in `QueryResult` or `Hit` objects.

Let's see some examples from our BLAST and BLAT searches. We'll look at the BLAST HSP first:

```
>>> from Bio import SearchIO
>>> blast_qresult = SearchIO.read("my_blast.xml", "blast-xml")
>>> blast_hsp = blast_qresult[0][0]  # first hit, first hsp
>>> print(blast_hsp)
      Query: 42291 mystery_seq
        Hit: gi|262205317|ref|NR_030195.1| Homo sapiens microRNA 520b (MIR520...
Query range: [0:61] (1)
  Hit range: [0:61] (1)
Quick stats: evalue 4.9e-23; bitscore 111.29
  Fragments: 1 (61 columns)
     Query - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
             |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
       Hit - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
```

Just like `QueryResult` and `Hit`, invoking `print` on an HSP shows its general details:

- There are the query and hit IDs and descriptions. We need these to identify our HSP.
- We've also got the matching range of the query and hit sequences. The slice notation we're using here is an indication that the range is displayed using Python's indexing style (zero-based, half open). The number inside the parenthesis denotes the strand. In this case, both sequences have the plus strand.
- Some quick statistics are available: the e-value and bitscore.
- There is information about the HSP fragments. Ignore this for now; it will be explained later on.
- And finally, we have the query and hit sequence alignment itself.

These details can be accessed on their own using the dot notation, just like in `QueryResult` and `Hit`:

```
>>> blast_hsp.query_range
(0, 61)
```

```
>>> blast_hsp.evalue
4.91307e-23
```

They're not the only attributes available, though. HSP objects come with a default set of properties that makes it easy to probe their various details. Here are some examples:

```
>>> blast_hsp.hit_start  # start coordinate of the hit sequence
0
>>> blast_hsp.query_span  # how many residues in the query sequence
61
>>> blast_hsp.aln_span  # how long the alignment is
61
```

Check out the HSP documentation under *Bio.SearchIO* for a full list of these predefined properties.

Furthermore, each sequence search tool usually computes its own statistics / details for its HSP objects. For example, an XML BLAST search also outputs the number of gaps and identical residues. These attributes can be accessed like so:

```
>>> blast_hsp.gap_num  # number of gaps
0
>>> blast_hsp.ident_num  # number of identical residues
61
```

These details are format-specific; they may not be present in other formats. To see which details are available for a given sequence search tool, you should check the format's documentation in `Bio.SearchIO`. Alternatively, you may also use `.__dict__.keys()` for a quick list of what's available:

```
>>> blast_hsp.__dict__.keys()
['bitscore', 'evalue', 'ident_num', 'gap_num', 'bitscore_raw', 'pos_num', '_items']
```

Finally, you may have noticed that the `query` and `hit` attributes of our HSP are not just regular strings:

```
>>> blast_hsp.query
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG'), id=
↪'42291', name='aligned query sequence', description='mystery_seq', dbxrefs=[])
>>> blast_hsp.hit
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG'), id=
↪'gi|262205317|ref|NR_030195.1|', name='aligned hit sequence', description='Homo␣
↪sapiens microRNA 520b (MIR520B), microRNA', dbxrefs=[])
```

They are `SeqRecord` objects you saw earlier in Section *Sequence annotation objects*! This means that you can do all sorts of interesting things you can do with `SeqRecord` objects on `HSP.query` and/or `HSP.hit`.

It should not surprise you now that the HSP object has an `alignment` property which is a `MultipleSeqAlignment` object:

```
>>> print(blast_hsp.aln)
Alignment with 2 rows and 61 columns
CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAG...GGG 42291
CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAG...GGG gi|262205317|ref|NR_030195.1|
```

Having probed the BLAST HSP, let's now take a look at HSPs from our BLAT results for a different kind of HSP. As usual, we'll begin by invoking `print` on it:

```
>>> blat_qresult = SearchIO.read("my_blat.psl", "blat-psl")
>>> blat_hsp = blat_qresult[0][0]  # first hit, first hsp
>>> print(blat_hsp)
      Query: mystery_seq <unknown description>
        Hit: chr19 <unknown description>
Query range: [0:61] (1)
  Hit range: [54204480:54204541] (1)
Quick stats: evalue ?; bitscore ?
  Fragments: 1 (? columns)
```

Some of the outputs you may have already guessed. We have the query and hit IDs and descriptions and the sequence coordinates. Values for evalue and bitscore is '?' as BLAT HSPs do not have these attributes. But The biggest difference here is that you don't see any sequence alignments displayed. If you look closer, PSL formats themselves do not have any hit or query sequences, so `Bio.SearchIO` won't create any sequence or alignment objects. What happens if you try to access `HSP.query`, `HSP.hit`, or `HSP.aln`? You'll get the default values for these attributes, which is `None`:

```
>>> blat_hsp.hit is None
True
>>> blat_hsp.query is None
True
>>> blat_hsp.aln is None
True
```

This does not affect other attributes, though. For example, you can still access the length of the query or hit alignment. Despite not displaying any attributes, the PSL format still have this information so `Bio.SearchIO` can extract them:

```
>>> blat_hsp.query_span  # length of query match
61
>>> blat_hsp.hit_span  # length of hit match
61
```

Other format-specific attributes are still present as well:

```
>>> blat_hsp.score  # PSL score
61
>>> blat_hsp.mismatch_num  # the mismatch column
0
```

So far so good? Things get more interesting when you look at another 'variant' of HSP present in our BLAT results. You might recall that in BLAT searches, sometimes we get our results separated into 'blocks'. These blocks are essentially alignment fragments that may have some intervening sequence between them.

Let's take a look at a BLAT HSP that contains multiple blocks to see how `Bio.SearchIO` deals with this:

```
>>> blat_hsp2 = blat_qresult[0][1]  # first hit, second hsp
>>> print(blat_hsp2)
      Query: mystery_seq <unknown description>
        Hit: chr19 <unknown description>
Query range: [0:61] (1)
  Hit range: [54233104:54264463] (1)
Quick stats: evalue ?; bitscore ?
  Fragments: ---  --------------  ----------------------  ----------------------
               #            Span             Query range               Hit range
             ---  --------------  ----------------------  ----------------------
```

(continues on next page)

(continued from previous page)

```
                0             ?              [0:18]     [54233104:54233122]
                1             ?             [18:61]     [54264420:54264463]
```

What's happening here? We still some essential details covered: the IDs and descriptions, the coordinates, and the quick statistics are similar to what you've seen before. But the fragments detail is all different. Instead of showing 'Fragments: 1', we now have a table with two data rows.

This is how `Bio.SearchIO` deals with HSPs having multiple fragments. As mentioned before, an HSP alignment may be separated by intervening sequences into fragments. The intervening sequences are not part of the query-hit match, so they should not be considered part of query nor hit sequence. However, they do affect how we deal with sequence coordinates, so we can't ignore them.

Take a look at the hit coordinate of the HSP above. In the `Hit range:` field, we see that the coordinate is `[54233104:54264463]`. But looking at the table rows, we see that not the entire region spanned by this coordinate matches our query. Specifically, the intervening region spans from `54233122` to `54264420`.

Why then, is the query coordinates seem to be contiguous, you ask? This is perfectly fine. In this case it means that the query match is contiguous (no intervening regions), while the hit match is not.

All these attributes are accessible from the HSP directly, by the way:

```
>>> blat_hsp2.hit_range  # hit start and end coordinates of the entire HSP
(54233104, 54264463)
>>> blat_hsp2.hit_range_all  # hit start and end coordinates of each fragment
[(54233104, 54233122), (54264420, 54264463)]
>>> blat_hsp2.hit_span  # hit span of the entire HSP
31359
>>> blat_hsp2.hit_span_all  # hit span of each fragment
[18, 43]
>>> blat_hsp2.hit_inter_ranges  # start and end coordinates of intervening regions in␣
↪the hit sequence
[(54233122, 54264420)]
>>> blat_hsp2.hit_inter_spans  # span of intervening regions in the hit sequence
[31298]
```

Most of these attributes are not readily available from the PSL file we have, but `Bio.SearchIO` calculates them for you on the fly when you parse the PSL file. All it needs are the start and end coordinates of each fragment.

What about the `query`, `hit`, and `aln` attributes? If the HSP has multiple fragments, you won't be able to use these attributes as they only fetch single `SeqRecord` or `MultipleSeqAlignment` objects. However, you can use their `*_all` counterparts: `query_all`, `hit_all`, and `aln_all`. These properties will return a list containing `SeqRecord` or `MultipleSeqAlignment` objects from each of the HSP fragment. There are other attributes that behave similarly, i.e. they only work for HSPs with one fragment. Check out the `HSP` documentation under *Bio.SearchIO* for a full list.

Finally, to check whether you have multiple fragments or not, you can use the `is_fragmented` property like so:

```
>>> blat_hsp2.is_fragmented  # BLAT HSP with 2 fragments
True
>>> blat_hsp.is_fragmented  # BLAT HSP from earlier, with one fragment
False
```

Before we move on, you should also know that we can use the slice notation on `HSP` objects, just like `QueryResult` or `Hit` objects. When you use this notation, you'll get an `HSPFragment` object in return, the last component of the object model.

12.1.4 HSPFragment

`HSPFragment` represents a single, contiguous match between the query and hit sequences. You could consider it the core of the object model and search result, since it is the presence of these fragments that determine whether your search have results or not.

In most cases, you don't have to deal with `HSPFragment` objects directly since not that many sequence search tools fragment their HSPs. When you do have to deal with them, what you should remember is that `HSPFragment` objects were written with to be as compact as possible. In most cases, they only contain attributes directly related to sequences: strands, reading frames, molecule types, coordinates, the sequences themselves, and their IDs and descriptions.

These attributes are readily shown when you invoke `print` on an `HSPFragment`. Here's an example, taken from our BLAST search:

```
>>> from Bio import SearchIO
>>> blast_qresult = SearchIO.read("my_blast.xml", "blast-xml")
>>> blast_frag = blast_qresult[0][0][0]  # first hit, first hsp, first fragment
>>> print(blast_frag)
      Query: 42291 mystery_seq
        Hit: gi|262205317|ref|NR_030195.1| Homo sapiens microRNA 520b (MIR520...
Query range: [0:61] (1)
  Hit range: [0:61] (1)
  Fragments: 1 (61 columns)
     Query - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
             |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
       Hit - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
```

At this level, the BLAT fragment looks quite similar to the BLAST fragment, save for the query and hit sequences which are not present:

```
>>> blat_qresult = SearchIO.read("my_blat.psl", "blat-psl")
>>> blat_frag = blat_qresult[0][0][0]  # first hit, first hsp, first fragment
>>> print(blat_frag)
      Query: mystery_seq <unknown description>
        Hit: chr19 <unknown description>
Query range: [0:61] (1)
  Hit range: [54204480:54204541] (1)
  Fragments: 1 (? columns)
```

In all cases, these attributes are accessible using our favorite dot notation. Some examples:

```
>>> blast_frag.query_start  # query start coordinate
0
>>> blast_frag.hit_strand  # hit sequence strand
1
>>> blast_frag.hit  # hit sequence, as a SeqRecord object
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG'), id=
↪'gi|262205317|ref|NR_030195.1|', name='aligned hit sequence', description='Homo␣
↪sapiens microRNA 520b (MIR520B), microRNA', dbxrefs=[])
```

12.2 A note about standards and conventions

Before we move on to the main functions, there is something you ought to know about the standards `Bio.SearchIO` uses. If you've worked with multiple sequence search tools, you might have had to deal with the many different ways each program deals with things like sequence coordinates. It might not have been a pleasant experience as these search tools usually have their own standards. For example, one tools might use one-based coordinates, while the other uses zero-based coordinates. Or, one program might reverse the start and end coordinates if the strand is minus, while others don't. In short, these often creates unnecessary mess must be dealt with.

We realize this problem ourselves and we intend to address it in `Bio.SearchIO`. After all, one of the goals of `Bio.SearchIO` is to create a common, easy to use interface to deal with various search output files. This means creating standards that extend beyond the object model you just saw.

Now, you might complain, "Not another standard!". Well, eventually we have to choose one convention or the other, so this is necessary. Plus, we're not creating something entirely new here; just adopting a standard we think is best for a Python programmer (it is Biopython, after all).

There are three implicit standards that you can expect when working with `Bio.SearchIO`:

- The first one pertains to sequence coordinates. In `Bio.SearchIO`, all sequence coordinates follows Python's coordinate style: zero-based and half open. For example, if in a BLAST XML output file the start and end coordinates of an HSP are 10 and 28, they would become 9 and 28 in `Bio.SearchIO`. The start coordinate becomes 9 because Python indices start from zero, while the end coordinate remains 28 as Python slices omit the last item in an interval.
- The second is on sequence coordinate orders. In `Bio.SearchIO`, start coordinates are always less than or equal to end coordinates. This isn't always the case with all sequence search tools, as some of them have larger start coordinates when the sequence strand is minus.
- The last one is on strand and reading frame values. For strands, there are only four valid choices: `1` (plus strand), `-1` (minus strand), `0` (protein sequences), and `None` (no strand). For reading frames, the valid choices are integers from `-3` to `3` and `None`.

Note that these standards only exist in `Bio.SearchIO` objects. If you write `Bio.SearchIO` objects into an output format, `Bio.SearchIO` will use the format's standard for the output. It does not force its standard over to your output file.

12.3 Reading search output files

There are two functions you can use for reading search output files into `Bio.SearchIO` objects: `read` and `parse`. They're essentially similar to `read` and `parse` functions in other submodules like `Bio.SeqIO` or `Bio.AlignIO`. In both cases, you need to supply the search output file name and the file format name, both as Python strings. You can check the documentation for a list of format names `Bio.SearchIO` recognizes.

`Bio.SearchIO.read` is used for reading search output files with only one query and returns a `QueryResult` object. You've seen `read` used in our previous examples. What you haven't seen is that `read` may also accept additional keyword arguments, depending on the file format.

Here are some examples. In the first one, we use `read` just like previously to read a BLAST tabular output file. In the second one, we use a keyword argument to modify so it parses the BLAST tabular variant with comments in it:

```
>>> from Bio import SearchIO
>>> qresult = SearchIO.read("tab_2226_tblastn_003.txt", "blast-tab")
>>> qresult
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> qresult2 = SearchIO.read("tab_2226_tblastn_007.txt", "blast-tab", comments=True)
```

(continues on next page)

(continued from previous page)

```
>>> qresult2
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
```

These keyword arguments differs among file formats. Check the format documentation to see if it has keyword arguments that modifies its parser's behavior.

As for the `Bio.SearchIO.parse`, it is used for reading search output files with any number of queries. The function returns a generator object that yields a `QueryResult` object in each iteration. Like `Bio.SearchIO.read`, it also accepts format-specific keyword arguments:

```
>>> from Bio import SearchIO
>>> qresults = SearchIO.parse("tab_2226_tblastn_001.txt", "blast-tab")
>>> for qresult in qresults:
...     print(qresult.id)
...
gi|16080617|ref|NP_391444.1|
gi|11464971:4-101
>>> qresults2 = SearchIO.parse("tab_2226_tblastn_005.txt", "blast-tab", comments=True)
>>> for qresult in qresults2:
...     print(qresult.id)
...
random_s00
gi|16080617|ref|NP_391444.1|
gi|11464971:4-101
```

12.4 Dealing with large search output files with indexing

Sometimes, you're handed a search output file containing hundreds or thousands of queries that you need to parse. You can of course use `Bio.SearchIO.parse` for this file, but that would be grossly inefficient if you need to access only a few of the queries. This is because `parse` will parse all queries it sees before it fetches your query of interest.

In this case, the ideal choice would be to index the file using `Bio.SearchIO.index` or `Bio.SearchIO.index_db`. If the names sound familiar, it's because you've seen them before in Section *Sequence files as Dictionaries – Indexed files*. These functions also behave similarly to their `Bio.SeqIO` counterparts, with the addition of format-specific keyword arguments.

Here are some examples. You can use `index` with just the filename and format name:

```
>>> from Bio import SearchIO
>>> idx = SearchIO.index("tab_2226_tblastn_001.txt", "blast-tab")
>>> sorted(idx.keys())
['gi|11464971:4-101', 'gi|16080617|ref|NP_391444.1|']
>>> idx["gi|16080617|ref|NP_391444.1|"]
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> idx.close()
```

Or also with the format-specific keyword argument:

```
>>> idx = SearchIO.index("tab_2226_tblastn_005.txt", "blast-tab", comments=True)
>>> sorted(idx.keys())
['gi|11464971:4-101', 'gi|16080617|ref|NP_391444.1|', 'random_s00']
>>> idx["gi|16080617|ref|NP_391444.1|"]
```

(continues on next page)

(continued from previous page)

```
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> idx.close()
```

Or with the `key_function` argument, as in `Bio.SeqIO`:

```
>>> key_function = lambda id: id.upper()  # capitalizes the keys
>>> idx = SearchIO.index("tab_2226_tblastn_001.txt", "blast-tab", key_function=key_
↪function)
>>> sorted(idx.keys())
['GI|11464971:4-101', 'GI|16080617|REF|NP_391444.1|']
>>> idx["GI|16080617|REF|NP_391444.1|"]
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> idx.close()
```

`Bio.SearchIO.index_db` works like as `index`, only it writes the query offsets into an SQLite database file.

12.5 Writing and converting search output files

It is occasionally useful to be able to manipulate search results from an output file and write it again to a new file. `Bio.SearchIO` provides a `write` function that lets you do exactly this. It takes as its arguments an iterable returning `QueryResult` objects, the output filename to write to, the format name to write to, and optionally some format-specific keyword arguments. It returns a four-item tuple, which denotes the number or `QueryResult`, `Hit`, `HSP`, and `HSPFragment` objects that were written.

```
>>> from Bio import SearchIO
>>> qresults = SearchIO.parse("mirna.xml", "blast-xml")  # read XML file
>>> SearchIO.write(qresults, "results.tab", "blast-tab")  # write to tabular file
(3, 239, 277, 277)
```

You should note different file formats require different attributes of the `QueryResult`, `Hit`, `HSP` and `HSPFragment` objects. If these attributes are not present, writing won't work. In other words, you can't always write to the output format that you want. For example, if you read a BLAST XML file, you wouldn't be able to write the results to a PSL file as PSL files require attributes not calculated by BLAST (e.g. the number of repeat matches). You can always set these attributes manually, if you really want to write to PSL, though.

Like `read`, `parse`, `index`, and `index_db`, `write` also accepts format-specific keyword arguments. Check out the documentation for a complete list of formats `Bio.SearchIO` can write to and their arguments.

Finally, `Bio.SearchIO` also provides a `convert` function, which is simply a shortcut for `Bio.SearchIO.parse` and `Bio.SearchIO.write`. Using the convert function, our example above would be:

```
>>> from Bio import SearchIO
>>> SearchIO.convert("mirna.xml", "blast-xml", "results.tab", "blast-tab")
(3, 239, 277, 277)
```

As `convert` uses `write`, it is only limited to format conversions that have all the required attributes. Here, the BLAST XML file provides all the default values a BLAST tabular file requires, so it works just fine. However, other format conversions are less likely to work since you need to manually assign the required attributes first.

CHAPTER

THIRTEEN

ACCESSING NCBI'S ENTREZ DATABASES

Entrez (https://www.ncbi.nlm.nih.gov/Web/Search/entrezfs.html) is a data retrieval system that provides users access to NCBI's databases such as PubMed, GenBank, GEO, and many others. You can access Entrez from a web browser to manually enter queries, or you can use Biopython's `Bio.Entrez` module for programmatic access to Entrez. The latter allows you for example to search PubMed or download GenBank records from within a Python script.

The `Bio.Entrez` module makes use of the Entrez Programming Utilities (also known as EUtils), consisting of eight tools that are described in detail on NCBI's page at https://www.ncbi.nlm.nih.gov/books/NBK25501/. Each of these tools corresponds to one Python function in the `Bio.Entrez` module, as described in the sections below. This module makes sure that the correct URL is used for the queries, and that NCBI's guidelines for responsible data access are being followed.

The output returned by the Entrez Programming Utilities is typically in XML format. To parse such output, you have several options:

1. Use `Bio.Entrez`'s parser to parse the XML output into a Python object;
2. Use one of the XML parsers available in Python's standard library;
3. Read the XML output as raw text, and parse it by string searching and manipulation.

See the Python documentation for a description of the XML parsers in Python's standard library. Here, we discuss the parser in Biopython's `Bio.Entrez` module. This parser can be used to parse data as provided through `Bio.Entrez`'s programmatic access functions to Entrez, but can also be used to parse XML data from NCBI Entrez that are stored in a file. In the latter case, the XML file should be opened in binary mode (e.g. `open("myfile.xml", "rb")`) for the XML parser in `Bio.Entrez` to work correctly. Alternatively, you can pass the file name or path to the XML file, and let `Bio.Entrez` take care of opening and closing the file.

NCBI uses DTD (Document Type Definition) files to describe the structure of the information contained in XML files. Most of the DTD files used by NCBI are included in the Biopython distribution. The `Bio.Entrez` parser makes use of the DTD files when parsing an XML file returned by NCBI Entrez.

Occasionally, you may find that the DTD file associated with a specific XML file is missing in the Biopython distribution. In particular, this may happen when NCBI updates its DTD files. If this happens, `Entrez.read` will show a warning message with the name and URL of the missing DTD file. The parser will proceed to access the missing DTD file through the internet, allowing the parsing of the XML file to continue. However, the parser is much faster if the DTD file is available locally. For this purpose, please download the DTD file from the URL in the warning message and place it in the directory `...site-packages/Bio/Entrez/DTDs`, containing the other DTD files. If you don't have write access to this directory, you can also place the DTD file in `~/.biopython/Bio/Entrez/DTDs`, where ~ represents your home directory. Since this directory is read before the directory `...site-packages/Bio/Entrez/DTDs`, you can also put newer versions of DTD files there if the ones in `...site-packages/Bio/Entrez/DTDs` become outdated. Alternatively, if you installed Biopython from source, you can add the DTD file to the source code's `Bio/Entrez/DTDs` directory, and reinstall Biopython. This will install the new DTD file in the correct location together with the other DTD files.

The Entrez Programming Utilities can also generate output in other formats, such as the Fasta or GenBank file formats for sequence databases, or the MedLine format for the literature database, discussed in Section *Specialized parsers*.

The functions in `Bio.Entrez` for programmatic access to Entrez return data either in binary format or in text format, depending on the type of data requested. In most cases, these functions return data in text format by decoding the data obtained from NCBI Entrez to Python strings under the assumption that the encoding is UTF-8. However, XML data are returned in binary format. The reason for this is that the encoding is specified in the XML document itself, which means that we won't know the correct encoding to use until we start parsing the file. `Bio.Entrez`'s parser therefore accepts data in binary format, extracts the encoding from the XML, and uses it to decode all text in the XML document to Python strings, ensuring that all text (in particular in languages other than English) are interpreted correctly. This is also the reason why you should open an XML file a binary mode when you want to use `Bio.Entrez`'s parser to parse the file.

13.1 Entrez Guidelines

Before using Biopython to access the NCBI's online resources (via `Bio.Entrez` or some of the other modules), please read the NCBI's Entrez User Requirements. If the NCBI finds you are abusing their systems, they can and will ban your access!

To paraphrase:

- For any series of more than 100 requests, do this at weekends or outside USA peak times. This is up to you to obey.
- Use the https://eutils.ncbi.nlm.nih.gov address, not the standard NCBI Web address. Biopython uses this web address.
- If you are using a API key, you can make at most 10 queries per second, otherwise at most 3 queries per second. This is automatically enforced by Biopython. Include `api_key="MyAPIkey"` in the argument list or set it as a module level variable:

```
>>> from Bio import Entrez
>>> Entrez.api_key = "MyAPIkey"
```

- Use the optional email parameter so the NCBI can contact you if there is a problem. You can either explicitly set this as a parameter with each call to Entrez (e.g. include `email="A.N.Other@example.com"` in the argument list), or you can set a global email address:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"
```

  `Bio.Entrez` will then use this email address with each call to Entrez. The `example.com` address is a reserved domain name specifically for documentation (RFC 2606). Please DO NOT use a random email – it's better not to give an email at all. The email parameter has been mandatory since June 1, 2010. In case of excessive usage, NCBI will attempt to contact a user at the e-mail address provided prior to blocking access to the E-utilities.

- If you are using Biopython within some larger software suite, use the tool parameter to specify this. You can either explicitly set the tool name as a parameter with each call to Entrez (e.g. include `tool="MyLocalScript"` in the argument list), or you can set a global tool name:

```
>>> from Bio import Entrez
>>> Entrez.tool = "MyLocalScript"
```

  The tool parameter will default to Biopython.

- For large queries, the NCBI also recommend using their session history feature (the WebEnv session cookie string, see Section *Using the history and WebEnv*). This is only slightly more complicated.

In conclusion, be sensible with your usage levels. If you plan to download lots of data, consider other options. For example, if you want easy access to all the human genes, consider fetching each chromosome by FTP as a GenBank file, and importing these into your own BioSQL database (see Section *BioSQL – storing sequences in a relational database*).

13.2 EInfo: Obtaining information about the Entrez databases

EInfo provides field index term counts, last update, and available links for each of NCBI's databases. In addition, you can use EInfo to obtain a list of all database names accessible through the Entrez utilities:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.einfo()
>>> result = stream.read()
>>> stream.close()
```

The variable `result` now contains a list of databases in XML format:

```
>>> print(result)
<?xml version="1.0"?>
<!DOCTYPE eInfoResult PUBLIC "-//NLM//DTD eInfoResult, 11 May 2002//EN"
 "https://www.ncbi.nlm.nih.gov/entrez/query/DTD/eInfo_020511.dtd">
<eInfoResult>
<DbList>
        <DbName>pubmed</DbName>
        <DbName>protein</DbName>
        <DbName>nucleotide</DbName>
        <DbName>nuccore</DbName>
        <DbName>nucgss</DbName>
        <DbName>nucest</DbName>
        <DbName>structure</DbName>
        <DbName>genome</DbName>
        <DbName>books</DbName>
        <DbName>cancerchromosomes</DbName>
        <DbName>cdd</DbName>
        <DbName>gap</DbName>
        <DbName>domains</DbName>
        <DbName>gene</DbName>
        <DbName>genomeprj</DbName>
        <DbName>gensat</DbName>
        <DbName>geo</DbName>
        <DbName>gds</DbName>
        <DbName>homologene</DbName>
        <DbName>journals</DbName>
        <DbName>mesh</DbName>
        <DbName>ncbisearch</DbName>
        <DbName>nlmcatalog</DbName>
        <DbName>omia</DbName>
        <DbName>omim</DbName>
        <DbName>pmc</DbName>
        <DbName>popset</DbName>
        <DbName>probe</DbName>
```

(continues on next page)

(continued from previous page)

```
        <DbName>proteinclusters</DbName>
        <DbName>pcassay</DbName>
        <DbName>pccompound</DbName>
        <DbName>pcsubstance</DbName>
        <DbName>snp</DbName>
        <DbName>taxonomy</DbName>
        <DbName>toolkit</DbName>
        <DbName>unigene</DbName>
        <DbName>unists</DbName>
</DbList>
</eInfoResult>
```

Since this is a fairly simple XML file, we could extract the information it contains simply by string searching. Using `Bio.Entrez`'s parser instead, we can directly parse this XML file into a Python object:

```
>>> from Bio import Entrez
>>> stream = Entrez.einfo()
>>> record = Entrez.read(stream)
```

Now `record` is a dictionary with exactly one key:

```
>>> record.keys()
dict_keys(['DbList'])
```

The values stored in this key is the list of database names shown in the XML above:

```
>>> record["DbList"]
['pubmed', 'protein', 'nucleotide', 'nuccore', 'nucgss', 'nucest',
 'structure', 'genome', 'books', 'cancerchromosomes', 'cdd', 'gap',
 'domains', 'gene', 'genomeprj', 'gensat', 'geo', 'gds', 'homologene',
 'journals', 'mesh', 'ncbisearch', 'nlmcatalog', 'omia', 'omim', 'pmc',
 'popset', 'probe', 'proteinclusters', 'pcassay', 'pccompound',
 'pcsubstance', 'snp', 'taxonomy', 'toolkit', 'unigene', 'unists']
```

For each of these databases, we can use EInfo again to obtain more information:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.einfo(db="pubmed")
>>> record = Entrez.read(stream)
>>> record["DbInfo"]["Description"]
'PubMed bibliographic record'
```

```
>>> record["DbInfo"]["Count"]
'17989604'
>>> record["DbInfo"]["LastUpdate"]
'2008/05/24 06:45'
```

Try `record["DbInfo"].keys()` for other information stored in this record. One of the most useful is a list of possible search fields for use with ESearch:

```
>>> for field in record["DbInfo"]["FieldList"]:
...     print("%(Name)s, %(FullName)s, %(Description)s" % field)
```

(continues on next page)

(continued from previous page)

```
...
ALL, All Fields, All terms from all searchable fields
UID, UID, Unique number assigned to publication
FILT, Filter, Limits the records
TITL, Title, Words in title of publication
WORD, Text Word, Free text associated with publication
MESH, MeSH Terms, Medical Subject Headings assigned to publication
MAJR, MeSH Major Topic, MeSH terms of major importance to publication
AUTH, Author, Author(s) of publication
JOUR, Journal, Journal abbreviation of publication
AFFL, Affiliation, Author's institutional affiliation and address
...
```

That's a long list, but indirectly this tells you that for the PubMed database, you can do things like `Jones[AUTH]` to search the author field, or `Sanger[AFFL]` to restrict to authors at the Sanger Centre. This can be very handy - especially if you are not so familiar with a particular database.

13.3 ESearch: Searching the Entrez databases

To search any of these databases, we use `Bio.Entrez.esearch()`. For example, let's search in PubMed for publications that include Biopython in their title:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="pubmed", term="biopython[title]", retmax="40")
>>> record = Entrez.read(stream)
>>> "19304878" in record["IdList"]
True
```

```
>>> print(record["IdList"])
['22909249', '19304878']
```

In this output, you see PubMed IDs (including 19304878 which is the PMID for the Biopython application note), which can be retrieved by EFetch (see section *EFetch: Downloading full records from Entrez*).

You can also use ESearch to search GenBank. Here we'll do a quick search for the *matK* gene in *Cypripedioideae* orchids (see Section *EInfo: Obtaining information about the Entrez databases* about EInfo for one way to find out which fields you can search in each Entrez database):

```
>>> stream = Entrez.esearch(
...     db="nucleotide", term="Cypripedioideae[Orgn] AND matK[Gene]", idtype="acc"
... )
>>> record = Entrez.read(stream)
>>> record["Count"]
'348'
>>> record["IdList"]
['JQ660909.1', 'JQ660908.1', 'JQ660907.1', 'JQ660906.1', ..., 'JQ660890.1']
```

Each of the IDs (JQ660909.1, JQ660908.1, JQ660907.1, ...) is a GenBank identifier (Accession number). See section *EFetch: Downloading full records from Entrez* for information on how to actually download these GenBank records.

Note that instead of a species name like `Cypripedioideae[Orgn]`, you can restrict the search using an NCBI taxon identifier, here this would be `txid158330[Orgn]`. This isn't currently documented on the ESearch help page - the NCBI explained this in reply to an email query. You can often deduce the search term formatting by playing with the Entrez web interface. For example, including `complete[prop]` in a genome search restricts to just completed genomes.

As a final example, let's get a list of computational journal titles:

```
>>> stream = Entrez.esearch(db="nlmcatalog", term="computational[Journal]", retmax="20")
>>> record = Entrez.read(stream)
>>> print("{} computational journals found".format(record["Count"]))
117 computational Journals found
>>> print("The first 20 are\n{}".format(record["IdList"]))
['101660833', '101664671', '101661657', '101659814', '101657941',
 '101653734', '101669877', '101649614', '101647835', '101639023',
 '101627224', '101647801', '101589678', '101585369', '101645372',
 '101586429', '101582229', '101574747', '101564639', '101671907']
```

Again, we could use EFetch to obtain more information for each of these journal IDs.

ESearch has many useful options — see the ESearch help page for more information.

13.4 EPost: Uploading a list of identifiers

EPost uploads a list of UIs for use in subsequent search strategies; see the EPost help page for more information. It is available from Biopython through the `Bio.Entrez.epost()` function.

To give an example of when this is useful, suppose you have a long list of IDs you want to download using EFetch (maybe sequences, maybe citations – anything). When you make a request with EFetch your list of IDs, the database etc, are all turned into a long URL sent to the server. If your list of IDs is long, this URL gets long, and long URLs can break (e.g. some proxies don't cope well).

Instead, you can break this up into two steps, first uploading the list of IDs using EPost (this uses an "HTML post" internally, rather than an "HTML get", getting round the long URL problem). With the history support, you can then refer to this long list of IDs, and download the associated data with EFetch.

Let's look at a simple example to see how EPost works – uploading some PubMed identifiers:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> id_list = ["19304878", "18606172", "16403221", "16377612", "14871861", "14630660"]
>>> print(Entrez.epost("pubmed", id=",".join(id_list)).read())
<?xml version="1.0"?>
<!DOCTYPE ePostResult PUBLIC "-//NLM//DTD ePostResult, 11 May 2002//EN"
 "https://www.ncbi.nlm.nih.gov/entrez/query/DTD/ePost_020511.dtd">
<ePostResult>
    <QueryKey>1</QueryKey>
    <WebEnv>NCID_01_206841095_130.14.22.101_9001_1242061629</WebEnv>
</ePostResult>
```

The returned XML includes two important strings, `QueryKey` and `WebEnv` which together define your history session. You would extract these values for use with another Entrez call such as EFetch:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
```

(continues on next page)

(continued from previous page)

```
>>> id_list = ["19304878", "18606172", "16403221", "16377612", "14871861", "14630660"]
>>> search_results = Entrez.read(Entrez.epost("pubmed", id=",".join(id_list)))
>>> webenv = search_results["WebEnv"]
>>> query_key = search_results["QueryKey"]
```

Section *Using the history and WebEnv* shows how to use the history feature.

13.5 ESummary: Retrieving summaries from primary IDs

ESummary retrieves document summaries from a list of primary IDs (see the ESummary help page for more information). In Biopython, ESummary is available as `Bio.Entrez.esummary()`. Using the search result above, we can for example find out more about the journal with ID 30367:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esummary(db="nlmcatalog", id="101660833")
>>> record = Entrez.read(stream)
>>> info = record[0]["TitleMainList"][0]
>>> print("Journal info\nid: {}\nTitle: {}".format(record[0]["Id"], info["Title"]))
Journal info
id: 101660833
Title: IEEE transactions on computational imaging.
```

13.6 EFetch: Downloading full records from Entrez

EFetch is what you use when you want to retrieve a full record from Entrez. This covers several possible databases, as described on the main EFetch Help page.

For most of their databases, the NCBI support several different file formats. Requesting a specific file format from Entrez using `Bio.Entrez.efetch()` requires specifying the `rettype` and/or `retmode` optional arguments. The different combinations are described for each database type on the pages linked to on NCBI efetch webpage.

One common usage is downloading sequences in the FASTA or GenBank/GenPept plain text formats (which can then be parsed with `Bio.SeqIO`, see Sections *Parsing GenBank records from the net* and *EFetch: Downloading full records from Entrez*). From the *Cypripedioideae* example above, we can download GenBank record EU490707 using `Bio.Entrez.efetch`:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.efetch(db="nucleotide", id="EU490707", rettype="gb", retmode="text")
>>> print(stream.read())
LOCUS       EU490707               1302 bp    DNA     linear   PLN 26-JUL-2016
DEFINITION  Selenipedium aequinoctiale maturase K (matK) gene, partial cds;
            chloroplast.
ACCESSION   EU490707
VERSION     EU490707.1
KEYWORDS    .
SOURCE      chloroplast Selenipedium aequinoctiale
  ORGANISM  Selenipedium aequinoctiale
            Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;
```

(continues on next page)

(continued from previous page)

```
            Spermatophyta; Magnoliopsida; Liliopsida; Asparagales; Orchidaceae;
            Cypripedioideae; Selenipedium.
REFERENCE   1  (bases 1 to 1302)
  AUTHORS   Neubig,K.M., Whitten,W.M., Carlsward,B.S., Blanco,M.A., Endara,L.,
            Williams,N.H. and Moore,M.
  TITLE     Phylogenetic utility of ycf1 in orchids: a plastid gene more
            variable than matK
  JOURNAL   Plant Syst. Evol. 277 (1-2), 75-84 (2009)
REFERENCE   2  (bases 1 to 1302)
  AUTHORS   Neubig,K.M., Whitten,W.M., Carlsward,B.S., Blanco,M.A.,
            Endara,C.L., Williams,N.H. and Moore,M.J.
  TITLE     Direct Submission
  JOURNAL   Submitted (14-FEB-2008) Department of Botany, University of
            Florida, 220 Bartram Hall, Gainesville, FL 32611-8526, USA
FEATURES             Location/Qualifiers
     source          1..1302
                     /organism="Selenipedium aequinoctiale"
                     /organelle="plastid:chloroplast"
                     /mol_type="genomic DNA"
                     /specimen_voucher="FLAS:Blanco 2475"
                     /db_xref="taxon:256374"
     gene            <1..>1302
                     /gene="matK"
     CDS             <1..>1302
                     /gene="matK"
                     /codon_start=1
                     /transl_table=11
                     /product="maturase K"
                     /protein_id="ACC99456.1"
                     /translation="IFYEPVEIFGYDNKSSLVLVKRLITRMYQQNFLISSVNDSNQKG
                     FWGHKHFFSSHFSSQMVSEGFGVILEIPFSSQLVSSLEEKKIPKYQNLRSIHSIFPFL
                     EDKFLHLNYVSDLLIPHPIHLEILVQILQCRIKDVPSLHLLRLLFHEYHNLNSLITSK
                     KFIYAFSKRKKRFLWLLYNSYVYECEYLFQFLRKQSSYLRSTSSGVFLERTHLYVKIE
                     HLLVVCCNSFQRILCFLKDPFMHYVRYQGKAILASKGTLILMKKWKFHLVNFWQSYFH
                     FWSQPYRIHIKQLSNYSFSFLGYFSSVLENHLVVRNQMLENSFIINLLTKKFDTIAPV
                     ISLIGSLSKAQFCTVLGHPISKPIWTDFSDSDILDRFCRICRNLCRYHSGSSKKQVLY
                     RIKYILRLSCARTLARKHKSTVRTFMRRLGSGLLEEFFMEEE"
ORIGIN
        1 atttttacg aacctgtgga aatttttggt tatgacaata aatctagttt agtacttgtg
       61 aaacgtttaa ttactcgaat gtatcaacag aatttttga tttcttcggt taatgattct
      121 aaccaaaaag gattttgggg gcacaagcat ttttttttctt ctcatttttc ttctcaaatg
      181 gtatcagaag gttttggagt cattctggaa attccattct cgtcgcaatt agtatcttct
      241 cttgaagaaa aaaaaatacc aaaatatcag aatttacgat ctattcattc aatatttccc
      301 tttttagaag acaaattttt acatttgaat tatgtgtcag atctactaat accccatccc
      361 atccatctgg aaatcttggt tcaaatcctt caatgccgga tcaaggatgt tccttctttg
      421 catttattgc gattgctttt ccacgaatat cataatttga atagtctcat tacttcaaag
      481 aaattcattt acgccttttc aaaaagaaag aaaagattcc tttggttact atataattct
      541 tatgtatatg aatgcgaata tctattccag tttcttcgta aacagtcttc ttatttacga
      601 tcaacatctt ctggagtctt tcttgagcga acacatttat atgtaaaaat agaacatctt
      661 ctagtagtgt gttgtaattc ttttcagagg atcctatgct ttctcaagga tcctttcatg
      721 cattatgttc gatatcaagg aaaagcaatt ctggcttcaa agggaactct tattctgatg
      781 aagaaatgga aatttcatct tgtgaatttt tggcaatctt attttcactt ttggtctcaa
```

(continues on next page)

(continued from previous page)

```
      841 ccgtatagga ttcatataaa gcaattatcc aactattcct tctcttttct ggggtatttt
      901 tcaagtgtac tagaaaatca tttggtagta agaaatcaaa tgctagagaa ttcatttata
      961 ataaatcttc tgactaagaa attcgatacc atagccccag ttatttctct tattggatca
     1021 ttgtcgaaag ctcaattttg tactgtattg ggtcatccta ttagtaaacc gatctggacc
     1081 gatttctcgg attctgatat tcttgatcga ttttgccgga tatgtagaaa tctttgtcgt
     1141 tatcacagcg gatcctcaaa aaaacaggtt ttgtatcgta taaaatatat acttcgactt
     1201 tcgtgtgcta gaactttggc acggaaacat aaaagtacag tacgcacttt tatgcgaaga
     1261 ttaggttcgg gattattaga agaattcttt atggaagaag aa
//
```

Please be aware that as of October 2016 GI identifiers are discontinued in favor of accession numbers. You can still fetch sequences based on their GI, but new sequences are no longer given this identifier. You should instead refer to them by the "Accession number" as done in the example.

The arguments `rettype="gb"` and `retmode="text"` let us download this record in the GenBank format.

Note that until Easter 2009, the Entrez EFetch API let you use "genbank" as the return type, however the NCBI now insist on using the official return types of "gb" or "gbwithparts" (or "gp" for proteins) as described on online. Also note that until Feb 2012, the Entrez EFetch API would default to returning plain text files, but now defaults to XML.

Alternatively, you could for example use `rettype="fasta"` to get the Fasta-format; see the EFetch Sequences Help page for other options. Remember – the available formats depend on which database you are downloading from - see the main EFetch Help page.

If you fetch the record in one of the formats accepted by `Bio.SeqIO` (see Chapter *Sequence Input/Output*), you could directly parse it into a `SeqRecord`:

```
>>> from Bio import SeqIO
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.efetch(db="nucleotide", id="EU490707", rettype="gb", retmode="text")
>>> record = SeqIO.read(stream, "genbank")
>>> stream.close()
>>> print(record.id)
EU490707.1
>>> print(record.name)
EU490707
>>> print(record.description)
Selenipedium aequinoctiale maturase K (matK) gene, partial cds; chloroplast
>>> print(len(record.features))
3
>>> record.seq
Seq('ATTTTTTACGAACCTGTGGAAATTTTTGGTTATGACAATAAATCTAGTTTAGTA...GAA')
```

Note that a more typical use would be to save the sequence data to a local file, and *then* parse it with `Bio.SeqIO`. This can save you having to re-download the same file repeatedly while working on your script, and places less load on the NCBI's servers. For example:

```
import os
from Bio import SeqIO
from Bio import Entrez

Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
```

(continues on next page)

(continued from previous page)

```
filename = "EU490707.gbk"
if not os.path.isfile(filename):
    # Downloading...
    stream = Entrez.efetch(db="nucleotide", id="EU490707", rettype="gb", retmode="text")
    output = open(filename, "w")
    output.write(streame.read())
    output.close()
    stream.close()
    print("Saved")

print("Parsing...")
record = SeqIO.read(filename, "genbank")
print(record)
```

To get the output in XML format, which you can parse using the `Bio.Entrez.read()` function, use `retmode="xml"`:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.efetch(db="nucleotide", id="EU490707", retmode="xml")
>>> record = Entrez.read(stream)
>>> stream.close()
>>> record[0]["GBSeq_definition"]
'Selenipedium aequinoctiale maturase K (matK) gene, partial cds; chloroplast'
>>> record[0]["GBSeq_source"]
'chloroplast Selenipedium aequinoctiale'
```

So, that dealt with sequences. For examples of parsing file formats specific to the other databases (e.g. the `MEDLINE` format used in PubMed), see Section *Specialized parsers*.

If you want to perform a search with `Bio.Entrez.esearch()`, and then download the records with `Bio.Entrez.efetch()`, you should use the WebEnv history feature – see Section *Using the history and WebEnv*.

13.7 ELink: Searching for related items in NCBI Entrez

ELink, available from Biopython as `Bio.Entrez.elink()`, can be used to find related items in the NCBI Entrez databases. For example, you can us this to find nucleotide entries for an entry in the gene database, and other cool stuff.

Let's use ELink to find articles related to the Biopython application note published in *Bioinformatics* in 2009. The PubMed ID of this article is 19304878:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> pmid = "19304878"
>>> record = Entrez.read(Entrez.elink(dbfrom="pubmed", id=pmid))
```

The `record` variable consists of a Python list, one for each database in which we searched. Since we specified only one PubMed ID to search for, `record` contains only one item. This item is a dictionary containing information about our search term, as well as all the related items that were found:

```
>>> record[0]["DbFrom"]
'pubmed'
```

(continues on next page)

(continued from previous page)

```
>>> record[0]["IdList"]
['19304878']
```

The "LinkSetDb" key contains the search results, stored as a list consisting of one item for each target database. In our search results, we only find hits in the PubMed database (although sub-divided into categories):

```
>>> len(record[0]["LinkSetDb"])
8
```

The exact numbers should increase over time:

```
>>> for linksetdb in record[0]["LinkSetDb"]:
...     print(linksetdb["DbTo"], linksetdb["LinkName"], len(linksetdb["Link"]))
...
pubmed pubmed_pubmed 284
pubmed pubmed_pubmed_alsoviewed 7
pubmed pubmed_pubmed_citedin 926
pubmed pubmed_pubmed_combined 6
pubmed pubmed_pubmed_five 6
pubmed pubmed_pubmed_refs 17
pubmed pubmed_pubmed_reviews 12
pubmed pubmed_pubmed_reviews_five 6
```

The actual search results are stored as under the "Link" key.

Let's now at the first search result:

```
>>> record[0]["LinkSetDb"][0]["Link"][0]
{'Id': '19304878'}
```

This is the article we searched for, which doesn't help us much, so let's look at the second search result:

```
>>> record[0]["LinkSetDb"][0]["Link"][1]
{'Id': '14630660'}
```

This paper, with PubMed ID 14630660, is about the Biopython PDB parser.

We can use a loop to print out all PubMed IDs:

```
>>> for link in record[0]["LinkSetDb"][0]["Link"]:
...     print(link["Id"])
...
19304878
14630660
18689808
17121776
16377612
12368254
......
```

Now that was nice, but personally I am often more interested to find out if a paper has been cited. Well, ELink can do that too – at least for journals in Pubmed Central (see Section *Searching for citations*).

For help on ELink, see the ELink help page. There is an entire sub-page just for the link names, describing how different databases can be cross referenced.

13.8 EGQuery: Global Query - counts for search terms

EGQuery provides counts for a search term in each of the Entrez databases (i.e. a global query). This is particularly useful to find out how many items your search terms would find in each database without actually performing lots of separate searches with ESearch (see the example in *Searching, downloading, and parsing Entrez Nucleotide records* below).

In this example, we use `Bio.Entrez.egquery()` to obtain the counts for "Biopython":

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.egquery(term="biopython")
>>> record = Entrez.read(stream)
>>> for row in record["eGQueryResult"]:
...     print(row["DbName"], row["Count"])
...
pubmed 6
pmc 62
journals 0
...
```

See the EGQuery help page for more information.

13.9 ESpell: Obtaining spelling suggestions

ESpell retrieves spelling suggestions. In this example, we use `Bio.Entrez.espell()` to obtain the correct spelling of Biopython:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.espell(term="biopythooon")
>>> record = Entrez.read(stream)
>>> record["Query"]
'biopythooon'
>>> record["CorrectedQuery"]
'biopython'
```

See the ESpell help page for more information. The main use of this is for GUI tools to provide automatic suggestions for search terms.

13.10 Parsing huge Entrez XML files

The `Entrez.read` function reads the entire XML file returned by Entrez into a single Python object, which is kept in memory. To parse Entrez XML files too large to fit in memory, you can use the function `Entrez.parse`. This is a generator function that reads records in the XML file one by one. This function is only useful if the XML file reflects a Python list object (in other words, if `Entrez.read` on a computer with infinite memory resources would return a Python list).

For example, you can download the entire Entrez Gene database for a given organism as a file from NCBI's ftp site. These files can be very large. As an example, on September 4, 2009, the file `Homo_sapiens.ags.gz`, containing the Entrez Gene database for human, had a size of 116576 kB. This file, which is in the `ASN` format, can be converted into an XML file using NCBI's `gene2xml` program (see NCBI's ftp site for more information):

```
$ gene2xml -b T -i Homo_sapiens.ags -o Homo_sapiens.xml
```

The resulting XML file has a size of 6.1 GB. Attempting `Entrez.read` on this file will result in a `MemoryError` on many computers.

The XML file `Homo_sapiens.xml` consists of a list of Entrez gene records, each corresponding to one Entrez gene in human. `Entrez.parse` retrieves these gene records one by one. You can then print out or store the relevant information in each record by iterating over the records. For example, this script iterates over the Entrez gene records and prints out the gene numbers and names for all current genes:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = open("Homo_sapiens.xml", "rb")
>>> records = Entrez.parse(stream)
```

Alternatively, you can use

```
>>> records = Entrez.parse("Homo_sapiens.xml")
```

and let `Bio.Entrez` take care of opening and closing the file. This is safer, as the file will then automatically be closed after parsing it, or if an error occurs.

```
>>> for record in records:
...     status = record["Entrezgene_track-info"]["Gene-track"]["Gene-track_status"]
...     if status.attributes["value"] == "discontinued":
...         continue
...     geneid = record["Entrezgene_track-info"]["Gene-track"]["Gene-track_geneid"]
...     genename = record["Entrezgene_gene"]["Gene-ref"]["Gene-ref_locus"]
...     print(geneid, genename)
...
1 A1BG
2 A2M
3 A2MP
8 AA
9 NAT1
10 NAT2
11 AACP
12 SERPINA3
13 AADAC
14 AAMP
15 AANAT
16 AARS
17 AAVS1
...
```

13.11 HTML escape characters

Pubmed records may contain HTML tags to indicate e.g. subscripts, superscripts, or italic text, as well as mathematical symbols via MathML. By default, the `Bio.Entrez` parser treats all text as plain text without markup; for example, the fragment "$P < 0.05$" in the abstract of a Pubmed record, which is encoded as

```
<i>P</i> &lt; 0.05
```

in the XML returned by Entrez, is converted to the Python string

```
'<i>P</i> < 0.05'
```

by the `Bio.Entrez` parser. While this is more human-readable, it is not valid HTML due to the less-than sign, and makes further processing of the text e.g. by an HTML parser impractical. To ensure that all strings returned by the parser are valid HTML, call `Entrez.read` or `Entrez.parse` with the `escape` argument set to `True`:

```
>>> record = Entrez.read(stream, escape=True)
```

The parser will then replace all characters disallowed in HTML by their HTML-escaped equivalent; in the example above, the parser will generate

```
'<i>P</i> &lt; 0.05'
```

which is a valid HTML fragment. By default, `escape` is `False`.

13.12 Handling errors

13.12.1 The file is not an XML file

For example, this error occurs if you try to parse a Fasta file as if it were an XML file:

```
>>> from Bio import Entrez
>>> stream = open("NC_005816.fna", "rb")  # a Fasta file
>>> record = Entrez.read(stream)
Traceback (most recent call last):
  ...
Bio.Entrez.Parser.NotXMLError: Failed to parse the XML data (syntax error: line 1,␣
↪column 0). Please make sure that the input data are in XML format.
```

Here, the parser didn't find the `<?xml ...` tag with which an XML file is supposed to start, and therefore decides (correctly) that the file is not an XML file.

13.12.2 The file ends prematurely or is otherwise corrupted

When your file is in the XML format but is corrupted (for example, by ending prematurely), the parser will raise a CorruptedXMLError.

Here is an example of an XML file that ends prematurely:

```
<?xml version="1.0"?>
<!DOCTYPE eInfoResult PUBLIC "-//NLM//DTD eInfoResult, 11 May 2002//EN" "https://www.
```

(continues on next page)

(continued from previous page)

```
↪ncbi.nlm.nih.gov/entrez/query/DTD/eInfo_020511.dtd">
<eInfoResult>
<DbList>
        <DbName>pubmed</DbName>
        <DbName>protein</DbName>
        <DbName>nucleotide</DbName>
        <DbName>nuccore</DbName>
        <DbName>nucgss</DbName>
        <DbName>nucest</DbName>
        <DbName>structure</DbName>
        <DbName>genome</DbName>
        <DbName>books</DbName>
        <DbName>cancerchromosomes</DbName>
        <DbName>cdd</DbName>
```

which will generate the following traceback:

```
>>> Entrez.read(stream)
Traceback (most recent call last):
  ...
Bio.Entrez.Parser.CorruptedXMLError: Failed to parse the XML data (no element found:␣
↪line 16, column 0). Please make sure that the input data are not corrupted.
```

Note that the error message tells you at what point in the XML file the error was detected.

13.12.3 The file contains items that are missing from the associated DTD

This is an example of an XML file containing tags that do not have a description in the corresponding DTD file:

```
<?xml version="1.0"?>
<!DOCTYPE eInfoResult PUBLIC "-//NLM//DTD eInfoResult, 11 May 2002//EN" "https://www.
↪ncbi.nlm.nih.gov/entrez/query/DTD/eInfo_020511.dtd">
<eInfoResult>
        <DbInfo>
        <DbName>pubmed</DbName>
        <MenuName>PubMed</MenuName>
        <Description>PubMed bibliographic record</Description>
        <Count>20161961</Count>
        <LastUpdate>2010/09/10 04:52</LastUpdate>
        <FieldList>
                <Field>
...
                </Field>
        </FieldList>
        <DocsumList>
                <Docsum>
                        <DsName>PubDate</DsName>
                        <DsType>4</DsType>
                        <DsTypeName>string</DsTypeName>
                </Docsum>
                <Docsum>
                        <DsName>EPubDate</DsName>
```

(continues on next page)

(continued from previous page)

```
...
        </DbInfo>
</eInfoResult>
```

In this file, for some reason the tag <DocsumList> (and several others) are not listed in the DTD file eInfo_020511.dtd, which is specified on the second line as the DTD for this XML file. By default, the parser will stop and raise a ValidationError if it cannot find some tag in the DTD:

```
>>> from Bio import Entrez
>>> stream = open("einfo3.xml", "rb")
>>> record = Entrez.read(stream)
Traceback (most recent call last):
  ...
Bio.Entrez.Parser.ValidationError: Failed to find tag 'DocsumList' in the DTD. To skip␣
↪all tags that are not represented in the DTD, please call Bio.Entrez.read or Bio.
↪Entrez.parse with validate=False.
```

Optionally, you can instruct the parser to skip such tags instead of raising a ValidationError. This is done by calling Entrez.read or Entrez.parse with the argument validate equal to False:

```
>>> from Bio import Entrez
>>> stream = open("einfo3.xml", "rb")
>>> record = Entrez.read(stream, validate=False)
>>> stream.close()
```

Of course, the information contained in the XML tags that are not in the DTD are not present in the record returned by Entrez.read.

13.12.4 The file contains an error message

This may occur, for example, when you attempt to access a PubMed record for a nonexistent PubMed ID. By default, this will raise a RuntimeError:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esummary(db="pubmed", id="99999999")
>>> record = Entrez.read(stream)
Traceback (most recent call last):
...
RuntimeError: UID=99999999: cannot get document summary
```

If you are accessing multiple PubMed records, the RuntimeError would prevent you from receiving results for any of the PubMed records if one of the PubMed IDs is incorrect. To circumvent this, you can set the ignore_errors argument to True. This will return the requested results for the valid PubMed IDs, and an ErrorElement for the incorrect ID:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esummary(db="pubmed", id="19304878,99999999,31278684")
>>> record = Entrez.read(stream, ignore_errors=True)
>>> len(record)
3
```

(continues on next page)

(continued from previous page)

```
>>> record[0].tag
'DocSum'
>>> record[0]["Title"]
'Biopython: freely available Python tools for computational molecular biology and␣
↪bioinformatics.'
>>> record[1].tag
'ERROR'
>>> record[1]
ErrorElement('UID=99999999: cannot get document summary')
>>> record[2].tag
'DocSum'
>>> record[2]["Title"]
'Sharing Programming Resources Between Bio* Projects.'
```

13.13 Specialized parsers

The `Bio.Entrez.read()` function can parse most (if not all) XML output returned by Entrez. Entrez typically allows you to retrieve records in other formats, which may have some advantages compared to the XML format in terms of readability (or download size).

To request a specific file format from Entrez using `Bio.Entrez.efetch()` requires specifying the `rettype` and/or `retmode` optional arguments. The different combinations are described for each database type on the NCBI efetch webpage.

One obvious case is you may prefer to download sequences in the FASTA or GenBank/GenPept plain text formats (which can then be parsed with `Bio.SeqIO`, see Sections *Parsing GenBank records from the net* and *EFetch: Downloading full records from Entrez*). For the literature databases, Biopython contains a parser for the `MEDLINE` format used in PubMed.

13.13.1 Parsing Medline records

You can find the Medline parser in `Bio.Medline`. Suppose we want to parse the file `pubmed_result1.txt`, containing one Medline record. You can find this file in Biopython's `Tests\Medline` directory. The file looks like this:

```
PMID- 12230038
OWN - NLM
STAT- MEDLINE
DA  - 20020916
DCOM- 20030606
LR  - 20041117
PUBM- Print
IS  - 1467-5463 (Print)
VI  - 3
IP  - 3
DP  - 2002 Sep
TI  - The Bio* toolkits--a brief overview.
PG  - 296-302
AB  - Bioinformatics research is often difficult to do with commercial software. The
      Open Source BioPerl, BioPython and Biojava projects provide toolkits with
...
```

We first open the file and then parse it:

```
>>> from Bio import Medline
>>> with open("pubmed_result1.txt") as stream:
...     record = Medline.read(stream)
...
```

The `record` now contains the Medline record as a Python dictionary:

```
>>> record["PMID"]
'12230038'
```

```
>>> record["AB"]
'Bioinformatics research is often difficult to do with commercial software.
The Open Source BioPerl, BioPython and Biojava projects provide toolkits with
multiple functionality that make it easier to create customized pipelines or
analysis. This review briefly compares the quirks of the underlying languages
and the functionality, documentation, utility and relative advantages of the
Bio counterparts, particularly from the point of view of the beginning
biologist programmer.'
```

The key names used in a Medline record can be rather obscure; use

```
>>> help(record)
```

for a brief summary.

To parse a file containing multiple Medline records, you can use the `parse` function instead:

```
>>> from Bio import Medline
>>> with open("pubmed_result2.txt") as stream:
...     for record in Medline.parse(stream):
...         print(record["TI"])
...
A high level interface to SCOP and ASTRAL implemented in python.
GenomeDiagram: a python package for the visualization of large-scale genomic data.
Open source clustering software.
PDB file parser and structure class implemented in Python.
```

Instead of parsing Medline records stored in files, you can also parse Medline records downloaded by `Bio.Entrez.efetch`. For example, let's look at all Medline records in PubMed related to Biopython:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="pubmed", term="biopython")
>>> record = Entrez.read(stream)
>>> record["IdList"]
['19304878', '18606172', '16403221', '16377612', '14871861', '14630660', '12230038']
```

We now use `Bio.Entrez.efetch` to download these Medline records:

```
>>> idlist = record["IdList"]
>>> stream = Entrez.efetch(db="pubmed", id=idlist, rettype="medline", retmode="text")
```

Here, we specify `rettype="medline", retmode="text"` to obtain the Medline records in plain-text Medline format. Now we use `Bio.Medline` to parse these records:

```
>>> from Bio import Medline
>>> records = Medline.parse(stream)
>>> for record in records:
...     print(record["AU"])
...
['Cock PJ', 'Antao T', 'Chang JT', 'Chapman BA', 'Cox CJ', 'Dalke A', ..., 'de Hoon MJ']
['Munteanu CR', 'Gonzalez-Diaz H', 'Magalhaes AL']
['Casbon JA', 'Crooks GE', 'Saqi MA']
['Pritchard L', 'White JA', 'Birch PR', 'Toth IK']
['de Hoon MJ', 'Imoto S', 'Nolan J', 'Miyano S']
['Hamelryck T', 'Manderick B']
['Mangalam H']
```

For comparison, here we show an example using the XML format:

```
>>> stream = Entrez.efetch(db="pubmed", id=idlist, rettype="medline", retmode="xml")
>>> records = Entrez.read(stream)
>>> for record in records["PubmedArticle"]:
...     print(record["MedlineCitation"]["Article"]["ArticleTitle"])
...
Biopython: freely available Python tools for computational molecular biology and
 bioinformatics.
Enzymes/non-enzymes classification model complexity based on composition, sequence,
 3D and topological indices.
A high level interface to SCOP and ASTRAL implemented in python.
GenomeDiagram: a python package for the visualization of large-scale genomic data.
Open source clustering software.
PDB file parser and structure class implemented in Python.
The Bio* toolkits--a brief overview.
```

Note that in both of these examples, for simplicity we have naively combined ESearch and EFetch. In this situation, the NCBI would expect you to use their history feature, as illustrated in Section *Using the history and WebEnv*.

13.13.2 Parsing GEO records

GEO (Gene Expression Omnibus) is a data repository of high-throughput gene expression and hybridization array data. The `Bio.Geo` module can be used to parse GEO-formatted data.

The following code fragment shows how to parse the example GEO file `GSE16.txt` into a record and print the record:

```
>>> from Bio import Geo
>>> stream = open("GSE16.txt")
>>> records = Geo.parse(stream)
>>> for record in records:
...     print(record)
...
```

You can search the "gds" database (GEO datasets) with ESearch:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="gds", term="GSE16")
>>> record = Entrez.read(stream)
```

(continues on next page)

(continued from previous page)

```
>>> stream.close()
>>> record["Count"]
'27'
```

```
>>> record["IdList"]
['200000016', '100000028', ...]
```

From the Entrez website, UID "200000016" is GDS16 while the other hit "100000028" is for the associated platform, GPL28. Unfortunately, at the time of writing the NCBI don't seem to support downloading GEO files using Entrez (not as XML, nor in the *Simple Omnibus Format in Text* (SOFT) format).

However, it is actually pretty straight forward to download the GEO files by FTP from ftp://ftp.ncbi.nih.gov/pub/geo/ instead. In this case you might want ftp://ftp.ncbi.nih.gov/pub/geo/DATA/SOFT/by_series/GSE16/GSE16_family.soft.gz (a compressed file, see the Python module gzip).

13.13.3 Parsing UniGene records

UniGene is an NCBI database of the transcriptome, with each UniGene record showing the set of transcripts that are associated with a particular gene in a specific organism. A typical UniGene record looks like this:

```
ID          Hs.2
TITLE       N-acetyltransferase 2 (arylamine N-acetyltransferase)
GENE        NAT2
CYTOBAND    8p22
GENE_ID     10
LOCUSLINK   10
HOMOL       YES
EXPRESS      bone| connective tissue| intestine| liver| liver tumor| normal| soft tissue/
↪muscle tissue tumor| adult
RESTR_EXPR   adult
CHROMOSOME  8
STS         ACC=PMC310725P3 UNISTS=272646
STS         ACC=WIAF-2120 UNISTS=44576
STS         ACC=G59899 UNISTS=137181
...
STS         ACC=GDB:187676 UNISTS=155563
PROTSIM     ORG=10090; PROTGI=6754794; PROTID=NP_035004.1; PCT=76.55; ALN=288
PROTSIM     ORG=9796; PROTGI=149742490; PROTID=XP_001487907.1; PCT=79.66; ALN=288
PROTSIM     ORG=9986; PROTGI=126722851; PROTID=NP_001075655.1; PCT=76.90; ALN=288
...
PROTSIM     ORG=9598; PROTGI=114619004; PROTID=XP_519631.2; PCT=98.28; ALN=288

SCOUNT      38
SEQUENCE    ACC=BC067218.1; NID=g45501306; PID=g45501307; SEQTYPE=mRNA
SEQUENCE    ACC=NM_000015.2; NID=g116295259; PID=g116295260; SEQTYPE=mRNA
SEQUENCE    ACC=D90042.1; NID=g219415; PID=g219416; SEQTYPE=mRNA
SEQUENCE    ACC=D90040.1; NID=g219411; PID=g219412; SEQTYPE=mRNA
SEQUENCE    ACC=BC015878.1; NID=g16198419; PID=g16198420; SEQTYPE=mRNA
SEQUENCE    ACC=CR407631.1; NID=g47115198; PID=g47115199; SEQTYPE=mRNA
SEQUENCE    ACC=BG569293.1; NID=g13576946; CLONE=IMAGE:4722596; END=5'; LID=6989;␣
↪SEQTYPE=EST; TRACE=44157214
...
```

(continues on next page)

(continued from previous page)

```
SEQUENCE    ACC=AU099534.1; NID=g13550663; CLONE=HSI08034; END=5'; LID=8800; SEQTYPE=EST
//
```

This particular record shows the set of transcripts (shown in the SEQUENCE lines) that originate from the human gene NAT2, encoding en N-acetyltransferase. The PROTSIM lines show proteins with significant similarity to NAT2, whereas the STS lines show the corresponding sequence-tagged sites in the genome.

To parse UniGene files, use the Bio.UniGene module:

```
>>> from Bio import UniGene
>>> input = open("myunigenefile.data")
>>> record = UniGene.read(input)
```

The record returned by UniGene.read is a Python object with attributes corresponding to the fields in the UniGene record. For example,

```
>>> record.ID
"Hs.2"
>>> record.title
"N-acetyltransferase 2 (arylamine N-acetyltransferase)"
```

The EXPRESS and RESTR_EXPR lines are stored as Python lists of strings:

```
[
    "bone",
    "connective tissue",
    "intestine",
    "liver",
    "liver tumor",
    "normal",
    "soft tissue/muscle tissue tumor",
    "adult",
]
```

Specialized objects are returned for the STS, PROTSIM, and SEQUENCE lines, storing the keys shown in each line as attributes:

```
>>> record.sts[0].acc
'PMC310725P3'
>>> record.sts[0].unists
'272646'
```

and similarly for the PROTSIM and SEQUENCE lines.

To parse a file containing more than one UniGene record, use the parse function in Bio.UniGene:

```
>>> from Bio import UniGene
>>> input = open("unigenerecords.data")
>>> records = UniGene.parse(input)
>>> for record in records:
...     print(record.ID)
...
```

13.14 Using a proxy

Normally you won't have to worry about using a proxy, but if this is an issue on your network here is how to deal with it. Internally, `Bio.Entrez` uses the standard Python library `urllib` for accessing the NCBI servers. This will check an environment variable called `http_proxy` to configure any simple proxy automatically. Unfortunately this module does not support the use of proxies which require authentication.

You may choose to set the `http_proxy` environment variable once (how you do this will depend on your operating system). Alternatively you can set this within Python at the start of your script, for example:

```
import os

os.environ["http_proxy"] = "http://proxyhost.example.com:8080"
```

See the urllib documentation for more details.

13.15 Examples

13.15.1 PubMed and Medline

If you are in the medical field or interested in human issues (and many times even if you are not!), PubMed (https://www.ncbi.nlm.nih.gov/PubMed/) is an excellent source of all kinds of goodies. So like other things, we'd like to be able to grab information from it and use it in Python scripts.

In this example, we will query PubMed for all articles having to do with orchids (see section *A usage example* for our motivation). We first check how many of such articles there are:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.egquery(term="orchid")
>>> record = Entrez.read(stream)
>>> for row in record["eGQueryResult"]:
...     if row["DbName"] == "pubmed":
...         print(row["Count"])
...
463
```

Now we use the `Bio.Entrez.efetch` function to download the PubMed IDs of these 463 articles:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="pubmed", term="orchid", retmax=463)
>>> record = Entrez.read(stream)
>>> stream.close()
>>> idlist = record["IdList"]
```

This returns a Python list containing all of the PubMed IDs of articles related to orchids:

```
>>> print(idlist)
['18680603', '18665331', '18661158', '18627489', '18627452', '18612381',
'18594007', '18591784', '18589523', '18579475', '18575811', '18575690',
...
```

Now that we've got them, we obviously want to get the corresponding Medline records and extract the information from them. Here, we'll download the Medline records in the Medline flat-file format, and use the `Bio.Medline` module to parse them:

```
>>> from Bio import Medline
>>> stream = Entrez.efetch(db="pubmed", id=idlist, rettype="medline", retmode="text")
>>> records = Medline.parse(stream)
```

NOTE - We've just done a separate search and fetch here, the NCBI much prefer you to take advantage of their history support in this situation. See Section *Using the history and WebEnv*.

Keep in mind that `records` is an iterator, so you can iterate through the records only once. If you want to save the records, you can convert them to a list:

```
>>> records = list(records)
```

Let's now iterate over the records to print out some information about each record:

```
>>> for record in records:
...     print("title:", record.get("TI", "?"))
...     print("authors:", record.get("AU", "?"))
...     print("source:", record.get("SO", "?"))
...     print("")
...
```

The output for this looks like:

```
title: Sex pheromone mimicry in the early spider orchid (ophrys sphegodes):
patterns of hydrocarbons as the key mechanism for pollination by sexual
deception [In Process Citation]
authors: ['Schiestl FP', 'Ayasse M', 'Paulus HF', 'Lofstedt C', 'Hansson BS',
'Ibarra F', 'Francke W']
source: J Comp Physiol [A] 2000 Jun;186(6):567-74
```

Especially interesting to note is the list of authors, which is returned as a standard Python list. This makes it easy to manipulate and search using standard Python tools. For instance, we could loop through a whole bunch of entries searching for a particular author with code like the following:

```
>>> search_author = "Waits T"
>>> for record in records:
...     if not "AU" in record:
...         continue
...     if search_author in record["AU"]:
...         print("Author %s found: %s" % (search_author, record["SO"]))
...
```

Hopefully this section gave you an idea of the power and flexibility of the Entrez and Medline interfaces and how they can be used together.

13.15.2 Searching, downloading, and parsing Entrez Nucleotide records

Here we'll show a simple example of performing a remote Entrez query. In section *A usage example* of the parsing examples, we talked about using NCBI's Entrez website to search the NCBI nucleotide databases for info on Cypripedioideae, our friends the lady slipper orchids. Now, we'll look at how to automate that process using a Python script. In this example, we'll just show how to connect, get the results, and parse them, with the Entrez module doing all of the work.

First, we use EGQuery to find out the number of results we will get before actually downloading them. EGQuery will tell us how many search results were found in each of the databases, but for this example we are only interested in nucleotides:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.egquery(term="Cypripedioideae")
>>> record = Entrez.read(stream)
>>> for row in record["eGQueryResult"]:
...     if row["DbName"] == "nuccore":
...         print(row["Count"])
...
4457
```

So, we expect to find 4457 Entrez Nucleotide records (this increased from 814 records in 2008; it is likely to continue to increase in the future). If you find some ridiculously high number of hits, you may want to reconsider if you really want to download all of them, which is our next step. Let's use the `retmax` argument to restrict the maximum number of records retrieved to the number available in 2008:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(
...     db="nucleotide", term="Cypripedioideae", retmax=814, idtype="acc"
... )
>>> record = Entrez.read(stream)
>>> stream.close()
```

Here, `record` is a Python dictionary containing the search results and some auxiliary information. Just for information, let's look at what is stored in this dictionary:

```
>>> print(record.keys())
['Count', 'RetMax', 'IdList', 'TranslationSet', 'RetStart', 'QueryTranslation']
```

First, let's check how many results were found:

```
>>> print(record["Count"])
'4457'
```

You might have expected this to be 814, the maximum number of records we asked to retrieve. However, `Count` represents the total number of records available for that search, not how many were retrieved. The retrieved records are stored in `record['IdList']`, which should contain the total number we asked for:

```
>>> len(record["IdList"])
814
```

Let's look at the first five results:

```
>>> record["IdList"][:5]
['KX265015.1', 'KX265014.1', 'KX265013.1', 'KX265012.1', 'KX265011.1']
```

We can download these records using `efetch`. While you could download these records one by one, to reduce the load on NCBI's servers, it is better to fetch a bunch of records at the same time, shown below. However, in this situation you should ideally be using the history feature described later in Section *Using the history and WebEnv*.

```
>>> idlist = ",".join(record["IdList"][:5])
>>> print(idlist)
KX265015.1, KX265014.1, KX265013.1, KX265012.1, KX265011.1]
>>> stream = Entrez.efetch(db="nucleotide", id=idlist, retmode="xml")
>>> records = Entrez.read(stream)
>>> len(records)
5
```

Each of these records corresponds to one GenBank record.

```
>>> print(records[0].keys())
['GBSeq_moltype', 'GBSeq_source', 'GBSeq_sequence',
 'GBSeq_primary-accession', 'GBSeq_definition', 'GBSeq_accession-version',
 'GBSeq_topology', 'GBSeq_length', 'GBSeq_feature-table',
 'GBSeq_create-date', 'GBSeq_other-seqids', 'GBSeq_division',
 'GBSeq_taxonomy', 'GBSeq_references', 'GBSeq_update-date',
 'GBSeq_organism', 'GBSeq_locus', 'GBSeq_strandedness']

>>> print(records[0]["GBSeq_primary-accession"])
DQ110336

>>> print(records[0]["GBSeq_other-seqids"])
['gb|DQ110336.1|', 'gi|187237168']

>>> print(records[0]["GBSeq_definition"])
Cypripedium calceolus voucher Davis 03-03 A maturase (matR) gene, partial cds;
mitochondrial

>>> print(records[0]["GBSeq_organism"])
Cypripedium calceolus
```

You could use this to quickly set up searches – but for heavy usage, see Section *Using the history and WebEnv*.

13.15.3 Searching, downloading, and parsing GenBank records

The GenBank record format is a very popular method of holding information about sequences, sequence features, and other associated sequence information. The format is a good way to get information from the NCBI databases at https://www.ncbi.nlm.nih.gov/.

In this example we'll show how to query the NCBI databases,to retrieve the records from the query, and then parse them using `Bio.SeqIO` - something touched on in Section *Parsing GenBank records from the net*. For simplicity, this example *does not* take advantage of the WebEnv history feature – see Section *Using the history and WebEnv* for this.

First, we want to make a query and find out the ids of the records to retrieve. Here we'll do a quick search for one of our favorite organisms, *Opuntia* (prickly-pear cacti). We can do quick search and get back the GIs (GenBank identifiers) for all of the corresponding records. First we check how many records there are:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.egquery(term="Opuntia AND rpl16")
>>> record = Entrez.read(stream)
>>> for row in record["eGQueryResult"]:
...     if row["DbName"] == "nuccore":
...         print(row["Count"])
...
9
```

Now we download the list of GenBank identifiers:

```
>>> stream = Entrez.esearch(db="nuccore", term="Opuntia AND rpl16")
>>> record = Entrez.read(stream)
>>> gi_list = record["IdList"]
>>> gi_list
['57240072', '57240071', '6273287', '6273291', '6273290', '6273289', '6273286',
'6273285', '6273284']
```

Now we use these GIs to download the GenBank records - note that with older versions of Biopython you had to supply a comma separated list of GI numbers to Entrez, as of Biopython 1.59 you can pass a list and this is converted for you:

```
>>> gi_str = ",".join(gi_list)
>>> stream = Entrez.efetch(db="nuccore", id=gi_str, rettype="gb", retmode="text")
```

If you want to look at the raw GenBank files, you can read from this stream and print out the result:

```
>>> text = stream.read()
>>> print(text)
LOCUS       AY851612                 892 bp    DNA     linear   PLN 10-APR-2007
DEFINITION  Opuntia subulata rpl16 gene, intron; chloroplast.
ACCESSION   AY851612
VERSION     AY851612.1  GI:57240072
KEYWORDS    .
SOURCE      chloroplast Austrocylindropuntia subulata
  ORGANISM  Austrocylindropuntia subulata
            Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;
            Spermatophyta; Magnoliophyta; eudicotyledons; core eudicotyledons;
            Caryophyllales; Cactaceae; Opuntioideae; Austrocylindropuntia.
REFERENCE   1  (bases 1 to 892)
  AUTHORS   Butterworth,C.A. and Wallace,R.S.
...
```

In this case, we are just getting the raw records. To get the records in a more Python-friendly form, we can use `Bio.SeqIO` to parse the GenBank data into `SeqRecord` objects, including `SeqFeature` objects (see Chapter *Sequence Input/Output*):

```
>>> from Bio import SeqIO
>>> stream = Entrez.efetch(db="nuccore", id=gi_str, rettype="gb", retmode="text")
>>> records = SeqIO.parse(stream, "gb")
```

We can now step through the records and look at the information we are interested in:

```
>>> for record in records:
...     print(f"{record.name}, length {len(record)}, with {len(record.features)} features
↪")
...
AY851612, length 892, with 3 features
AY851611, length 881, with 3 features
AF191661, length 895, with 3 features
AF191665, length 902, with 3 features
AF191664, length 899, with 3 features
AF191663, length 899, with 3 features
AF191660, length 893, with 3 features
AF191659, length 894, with 3 features
AF191658, length 896, with 3 features
```

Using these automated query retrieval functionality is a big plus over doing things by hand. Although the module should obey the NCBI's max three queries per second rule, the NCBI have other recommendations like avoiding peak hours. See Section *Entrez Guidelines*. In particular, please note that for simplicity, this example does not use the WebEnv history feature. You should use this for any non-trivial search and download work, see Section *Using the history and WebEnv*.

Finally, if plan to repeat your analysis, rather than downloading the files from the NCBI and parsing them immediately (as shown in this example), you should just download the records *once* and save them to your hard disk, and then parse the local file.

13.15.4 Finding the lineage of an organism

Staying with a plant example, let's now find the lineage of the Cypripedioideae orchid family. First, we search the Taxonomy database for Cypripedioideae, which yields exactly one NCBI taxonomy identifier:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="Taxonomy", term="Cypripedioideae")
>>> record = Entrez.read(stream)
>>> record["IdList"]
['158330']
>>> record["IdList"][0]
'158330'
```

Now, we use `efetch` to download this entry in the Taxonomy database, and then parse it:

```
>>> stream = Entrez.efetch(db="Taxonomy", id="158330", retmode="xml")
>>> records = Entrez.read(stream)
```

Again, this record stores lots of information:

```
>>> records[0].keys()
['Lineage', 'Division', 'ParentTaxId', 'PubDate', 'LineageEx',
 'CreateDate', 'TaxId', 'Rank', 'GeneticCode', 'ScientificName',
 'MitoGeneticCode', 'UpdateDate']
```

We can get the lineage directly from this record:

```
>>> records[0]["Lineage"]
'cellular organisms; Eukaryota; Viridiplantae; Streptophyta; Streptophytina;
 Embryophyta; Tracheophyta; Euphyllophyta; Spermatophyta; Magnoliopsida;
 Liliopsida; Asparagales; Orchidaceae'
```

The record data contains much more than just the information shown here - for example look under `"LineageEx"` instead of `"Lineage"` and you'll get the NCBI taxon identifiers of the lineage entries too.

13.16 Using the history and WebEnv

Often you will want to make a series of linked queries. Most typically, running a search, perhaps refining the search, and then retrieving detailed search results. You *can* do this by making a series of separate calls to Entrez. However, the NCBI prefer you to take advantage of their history support - for example combining ESearch and EFetch.

Another typical use of the history support would be to combine EPost and EFetch. You use EPost to upload a list of identifiers, which starts a new history session. You then download the records with EFetch by referring to the session (instead of the identifiers).

13.16.1 Searching for and downloading sequences using the history

Suppose we want to search and download all the *Opuntia* rpl16 nucleotide sequences, and store them in a FASTA file. As shown in Section *Searching, downloading, and parsing GenBank records*, we can naively combine `Bio.Entrez.esearch()` to get a list of Accession numbers, and then call `Bio.Entrez.efetch()` to download them all.

However, the approved approach is to run the search with the history feature. Then, we can fetch the results by reference to the search results - which the NCBI can anticipate and cache.

To do this, call `Bio.Entrez.esearch()` as normal, but with the additional argument of `usehistory="y"`,

```
>>> from Bio import Entrez
>>> Entrez.email = "history.user@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(
...     db="nucleotide", term="Opuntia[orgn] and rpl16", usehistory="y", idtype="acc"
... )
>>> search_results = Entrez.read(stream)
>>> stream.close()
```

As before (see Section *Searching, downloading, and parsing Entrez Nucleotide records*), the XML output includes the first `retmax` search results, with `retmax` defaulting to 20:

```
>>> acc_list = search_results["IdList"]
>>> count = int(search_results["Count"])
>>> len(acc_list)
20
```

```
>>> count
28
```

You also get given two additional pieces of information, the `WebEnv` session cookie, and the `QueryKey`:

```
>>> webenv = search_results["WebEnv"]
>>> query_key = search_results["QueryKey"]
```

Having stored these values in variables `session_cookie` and `query_key` we can use them as parameters to `Bio.Entrez.efetch()` instead of giving the GI numbers as identifiers.

While for small searches you might be OK downloading everything at once, it is better to download in batches. You use the `retstart` and `retmax` parameters to specify which range of search results you want returned (starting entry using zero-based counting, and maximum number of results to return). Note that if Biopython encounters a transient failure like a HTTP 500 response when communicating with NCBI, it will automatically try again a couple of times. For example,

```
# This assumes you have already run a search as shown above,
# and set the variables count, webenv, query_key

batch_size = 3
output = open("orchid_rpl16.fasta", "w")
for start in range(0, count, batch_size):
    end = min(count, start + batch_size)
    print("Going to download record %i to %i" % (start + 1, end))
    stream = Entrez.efetch(
        db="nucleotide",
        rettype="fasta",
        retmode="text",
        retstart=start,
        retmax=batch_size,
        webenv=webenv,
        query_key=query_key,
        idtype="acc",
    )
    data = stream.read()
    stream.close()
    output.write(data)
output.close()
```

For illustrative purposes, this example downloaded the FASTA records in batches of three. Unless you are downloading genomes or chromosomes, you would normally pick a larger batch size.

13.16.2 Searching for and downloading abstracts using the history

Here is another history example, searching for papers published in the last year about the *Opuntia*, and then downloading them into a file in MedLine format:

```
from Bio import Entrez

Entrez.email = "history.user@example.com"
search_results = Entrez.read(
    Entrez.esearch(
        db="pubmed", term="Opuntia[ORGN]", reldate=365, datetype="pdat", usehistory="y"
    )
)
count = int(search_results["Count"])
print("Found %i results" % count)

batch_size = 10
output = open("recent_orchid_papers.txt", "w")
```

(continues on next page)

(continued from previous page)

```
for start in range(0, count, batch_size):
    end = min(count, start + batch_size)
    print("Going to download record %i to %i" % (start + 1, end))
    stream = Entrez.efetch(
        db="pubmed",
        rettype="medline",
        retmode="text",
        retstart=start,
        retmax=batch_size,
        webenv=search_results["WebEnv"],
        query_key=search_results["QueryKey"],
    )
    data = stream.read()
    stream.close()
    output.write(data)
output.close()
```

At the time of writing, this gave 28 matches - but because this is a date dependent search, this will of course vary. As described in Section *Parsing Medline records* above, you can then use `Bio.Medline` to parse the saved records.

13.16.3 Searching for citations

Back in Section *ELink: Searching for related items in NCBI Entrez* we mentioned ELink can be used to search for citations of a given paper. Unfortunately this only covers journals indexed for PubMed Central (doing it for all the journals in PubMed would mean a lot more work for the NIH). Let's try this for the Biopython PDB parser paper, PubMed ID 14630660:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> pmid = "14630660"
>>> results = Entrez.read(
...     Entrez.elink(dbfrom="pubmed", db="pmc", LinkName="pubmed_pmc_refs", id=pmid)
... )
>>> pmc_ids = [link["Id"] for link in results[0]["LinkSetDb"][0]["Link"]]
>>> pmc_ids
['2744707', '2705363', '2682512', ..., '1190160']
```

Great - eleven articles. But why hasn't the Biopython application note been found (PubMed ID 19304878)? Well, as you might have guessed from the variable names, there are not actually PubMed IDs, but PubMed Central IDs. Our application note is the third citing paper in that list, PMCID 2682512.

So, what if (like me) you'd rather get back a list of PubMed IDs? Well we can call ELink again to translate them. This becomes a two step process, so by now you should expect to use the history feature to accomplish it (Section *Using the history and WebEnv*).

But first, taking the more straightforward approach of making a second (separate) call to ELink:

```
>>> results2 = Entrez.read(
...     Entrez.elink(dbfrom="pmc", db="pubmed", LinkName="pmc_pubmed", id=",".join(pmc_
↪ids))
... )
>>> pubmed_ids = [link["Id"] for link in results2[0]["LinkSetDb"][0]["Link"]]
```

(continues on next page)

(continued from previous page)

```
>>> pubmed_ids
['19698094', '19450287', '19304878', ..., '15985178']
```

This time you can immediately spot the Biopython application note as the third hit (PubMed ID 19304878).

Now, let's do that all again but with the history … *TODO*.

And finally, don't forget to include your *own* email address in the Entrez calls.

CHAPTER

FOURTEEN

SWISS-PROT AND EXPASY

14.1 Parsing Swiss-Prot files

Swiss-Prot (https://web.expasy.org/docs/swiss-prot_guideline.html) is a hand-curated database of protein sequences. Biopython can parse the "plain text" Swiss-Prot file format, which is still used for the UniProt Knowledgebase which combined Swiss-Prot, TrEMBL and PIR-PSD.

Although in the following we focus on the older human readable plain text format, `Bio.SeqIO` can read both this and the newer UniProt XML file format for annotated protein sequences.

14.1.1 Parsing Swiss-Prot records

In Section *Parsing SwissProt sequences from the net*, we described how to extract the sequence of a Swiss-Prot record as a `SeqRecord` object. Alternatively, you can store the Swiss-Prot record in a `Bio.SwissProt.Record` object, which in fact stores the complete information contained in the Swiss-Prot record. In this section, we describe how to extract `Bio.SwissProt.Record` objects from a Swiss-Prot file.

To parse a Swiss-Prot record, we first get a handle to a Swiss-Prot record. There are several ways to do so, depending on where and how the Swiss-Prot record is stored:

- Open a Swiss-Prot file locally:

```
>>> handle = open("SwissProt/F2CXE6.txt")
```

- Open a gzipped Swiss-Prot file:

```
>>> import gzip
>>> handle = gzip.open("myswissprotfile.dat.gz", "rt")
```

- Open a Swiss-Prot file over the internet:

```
>>> from urllib.request import urlopen
>>> url = "https://raw.githubusercontent.com/biopython/biopython/master/Tests/
↪SwissProt/F2CXE6.txt"
>>> handle = urlopen(url)
```

  to open the file stored on the Internet before calling `read`.

- Open a Swiss-Prot file over the internet from the ExPASy database (see section *Retrieving a Swiss-Prot record*):

```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_sprot_raw("F2CXE6")
```

The key point is that for the parser, it doesn't matter how the handle was created, as long as it points to data in the Swiss-Prot format. The parser will automatically decode the data as ASCII (the encoding used by Swiss-Prot) if the handle was opened in binary mode.

We can use `Bio.SeqIO` as described in Section *Parsing SwissProt sequences from the net* to get file format agnostic `SeqRecord` objects. Alternatively, we can use `Bio.SwissProt` get `Bio.SwissProt.Record` objects, which are a much closer match to the underlying file format.

To read one Swiss-Prot record from the handle, we use the function `read()`:

```
>>> from Bio import SwissProt
>>> record = SwissProt.read(handle)
```

This function should be used if the handle points to exactly one Swiss-Prot record. It raises a `ValueError` if no Swiss-Prot record was found, and also if more than one record was found.

We can now print out some information about this record:

```
>>> print(record.description)
SubName: Full=Plasma membrane intrinsic protein {ECO:0000313|EMBL:BAN04711.1}; SubName:␣
↪Full=Predicted protein {ECO:0000313|EMBL:BAJ87517.1};
>>> for ref in record.references:
...     print("authors:", ref.authors)
...     print("title:", ref.title)
...
authors: Matsumoto T., Tanaka T., Sakai H., Amano N., Kanamori H., Kurita K., Kikuta A.,␣
↪Kamiya K., Yamamoto M., Ikawa H., Fujii N., Hori K., Itoh T., Sato K.
title: Comprehensive sequence analysis of 24,783 barley full-length cDNAs derived from␣
↪12 clone libraries.
authors: Shibasaka M., Sasano S., Utsugi S., Katsuhara M.
title: Functional characterization of a novel plasma membrane intrinsic protein2 in␣
↪barley.
authors: Shibasaka M., Katsuhara M., Sasano S.
title:
>>> print(record.organism_classification)
['Eukaryota', 'Viridiplantae', 'Streptophyta', 'Embryophyta', 'Tracheophyta',
↪'Spermatophyta', 'Magnoliophyta', 'Liliopsida', 'Poales', 'Poaceae', 'BEP clade',
↪'Pooideae', 'Triticeae', 'Hordeum']
```

To parse a file that contains more than one Swiss-Prot record, we use the `parse` function instead. This function allows us to iterate over the records in the file.

For example, let's parse the full Swiss-Prot database and collect all the descriptions. You can download this from the ExPASy FTP site as a single gzipped-file `uniprot_sprot.dat.gz` (about 300MB). This is a compressed file containing a single file, `uniprot_sprot.dat` (over 1.5GB).

As described at the start of this section, you can use the Python library `gzip` to open and uncompress a `.gz` file, like this:

```
>>> import gzip
>>> handle = gzip.open("uniprot_sprot.dat.gz", "rt")
```

However, uncompressing a large file takes time, and each time you open the file for reading in this way, it has to be decompressed on the fly. So, if you can spare the disk space you'll save time in the long run if you first decompress the file to disk, to get the `uniprot_sprot.dat` file inside. Then you can open the file for reading as usual:

```
>>> handle = open("uniprot_sprot.dat")
```

As of June 2009, the full Swiss-Prot database downloaded from ExPASy contained 468851 Swiss-Prot records. One concise way to build up a list of the record descriptions is with a list comprehension:

```
>>> from Bio import SwissProt
>>> handle = open("uniprot_sprot.dat")
>>> descriptions = [record.description for record in SwissProt.parse(handle)]
>>> len(descriptions)
468851
>>> descriptions[:5]
['RecName: Full=Protein MGF 100-1R;',
 'RecName: Full=Protein MGF 100-1R;',
 'RecName: Full=Protein MGF 100-1R;',
 'RecName: Full=Protein MGF 100-1R;',
 'RecName: Full=Protein MGF 100-2L;']
```

Or, using a for loop over the record iterator:

```
>>> from Bio import SwissProt
>>> descriptions = []
>>> handle = open("uniprot_sprot.dat")
>>> for record in SwissProt.parse(handle):
...     descriptions.append(record.description)
...
>>> len(descriptions)
468851
```

Because this is such a large input file, either way takes about eleven minutes on my new desktop computer (using the uncompressed `uniprot_sprot.dat` file as input).

It is equally easy to extract any kind of information you'd like from Swiss-Prot records. To see the members of a Swiss-Prot record, use

```
>>> dir(record)
['__doc__', '__init__', '__module__', 'accessions', 'annotation_update',
'comments', 'created', 'cross_references', 'data_class', 'description',
'entry_name', 'features', 'gene_name', 'host_organism', 'keywords',
'molecule_type', 'organelle', 'organism', 'organism_classification',
'references', 'seqinfo', 'sequence', 'sequence_length',
'sequence_update', 'taxonomy_id']
```

14.1.2 Parsing the Swiss-Prot keyword and category list

Swiss-Prot also distributes a file `keywlist.txt`, which lists the keywords and categories used in Swiss-Prot. The file contains entries in the following form:

```
ID   2Fe-2S.
AC   KW-0001
DE   Protein which contains at least one 2Fe-2S iron-sulfur cluster: 2 iron
DE   atoms complexed to 2 inorganic sulfides and 4 sulfur atoms of
DE   cysteines from the protein.
SY   Fe2S2; [2Fe-2S] cluster; [Fe2S2] cluster; Fe2/S2 (inorganic) cluster;
```

(continues on next page)

(continued from previous page)

```
SY   Di-mu-sulfido-diiron; 2 iron, 2 sulfur cluster binding.
GO   GO:0051537; 2 iron, 2 sulfur cluster binding
HI   Ligand: Iron; Iron-sulfur; 2Fe-2S.
HI   Ligand: Metal-binding; 2Fe-2S.
CA   Ligand.
//
ID   3D-structure.
AC   KW-0002
DE   Protein, or part of a protein, whose three-dimensional structure has
DE   been resolved experimentally (for example by X-ray crystallography or
DE   NMR spectroscopy) and whose coordinates are available in the PDB
DE   database. Can also be used for theoretical models.
HI   Technical term: 3D-structure.
CA   Technical term.
//
ID   3Fe-4S.
...
```

The entries in this file can be parsed by the `parse` function in the `Bio.SwissProt.KeyWList` module. Each entry is then stored as a `Bio.SwissProt.KeyWList.Record`, which is a Python dictionary.

```
>>> from Bio.SwissProt import KeyWList
>>> handle = open("keywlist.txt")
>>> records = KeyWList.parse(handle)
>>> for record in records:
...     print(record["ID"])
...     print(record["DE"])
...
```

This prints

```
2Fe-2S.
Protein which contains at least one 2Fe-2S iron-sulfur cluster: 2 iron atoms
complexed to 2 inorganic sulfides and 4 sulfur atoms of cysteines from the
protein.
...
```

14.2 Parsing Prosite records

Prosite is a database containing protein domains, protein families, functional sites, as well as the patterns and profiles to recognize them. Prosite was developed in parallel with Swiss-Prot. In Biopython, a Prosite record is represented by the `Bio.ExPASy.Prosite.Record` class, whose members correspond to the different fields in a Prosite record.

In general, a Prosite file can contain more than one Prosite records. For example, the full set of Prosite records, which can be downloaded as a single file (`prosite.dat`) from the ExPASy FTP site, contains 2073 records (version 20.24 released on 4 December 2007). To parse such a file, we again make use of an iterator:

```
>>> from Bio.ExPASy import Prosite
>>> handle = open("myprositefile.dat")
>>> records = Prosite.parse(handle)
```

We can now take the records one at a time and print out some information. For example, using the file containing the complete Prosite database, we'd find

```
>>> from Bio.ExPASy import Prosite
>>> handle = open("prosite.dat")
>>> records = Prosite.parse(handle)
>>> record = next(records)
>>> record.accession
'PS00001'
>>> record.name
'ASN_GLYCOSYLATION'
>>> record.pdoc
'PDOC00001'
>>> record = next(records)
>>> record.accession
'PS00004'
>>> record.name
'CAMP_PHOSPHO_SITE'
>>> record.pdoc
'PDOC00004'
>>> record = next(records)
>>> record.accession
'PS00005'
>>> record.name
'PKC_PHOSPHO_SITE'
>>> record.pdoc
'PDOC00005'
```

and so on. If you're interested in how many Prosite records there are, you could use

```
>>> from Bio.ExPASy import Prosite
>>> handle = open("prosite.dat")
>>> records = Prosite.parse(handle)
>>> n = 0
>>> for record in records:
...     n += 1
...
>>> n
2073
```

To read exactly one Prosite from the handle, you can use the `read` function:

```
>>> from Bio.ExPASy import Prosite
>>> handle = open("mysingleprositerecord.dat")
>>> record = Prosite.read(handle)
```

This function raises a ValueError if no Prosite record is found, and also if more than one Prosite record is found.

14.3 Parsing Prosite documentation records

In the Prosite example above, the `record.pdoc` accession numbers `'PDOC00001'`, `'PDOC00004'`, `'PDOC00005'` and so on refer to Prosite documentation. The Prosite documentation records are available from ExPASy as individual files, and as one file (`prosite.doc`) containing all Prosite documentation records.

We use the parser in `Bio.ExPASy.Prodoc` to parse Prosite documentation records. For example, to create a list of all accession numbers of Prosite documentation record, you can use

```
>>> from Bio.ExPASy import Prodoc
>>> handle = open("prosite.doc")
>>> records = Prodoc.parse(handle)
>>> accessions = [record.accession for record in records]
```

Again a `read()` function is provided to read exactly one Prosite documentation record from the handle.

14.4 Parsing Enzyme records

ExPASy's Enzyme database is a repository of information on enzyme nomenclature. A typical Enzyme record looks as follows:

```
ID   3.1.1.34
DE   Lipoprotein lipase.
AN   Clearing factor lipase.
AN   Diacylglycerol lipase.
AN   Diglyceride lipase.
CA   Triacylglycerol + H(2)O = diacylglycerol + a carboxylate.
CC   -!- Hydrolyzes triacylglycerols in chylomicrons and very low-density
CC       lipoproteins (VLDL).
CC   -!- Also hydrolyzes diacylglycerol.
PR   PROSITE; PDOC00110;
DR   P11151, LIPL_BOVIN ;  P11153, LIPL_CAVPO ;  P11602, LIPL_CHICK ;
DR   P55031, LIPL_FELCA ;  P06858, LIPL_HUMAN ;  P11152, LIPL_MOUSE ;
DR   O46647, LIPL_MUSVI ;  P49060, LIPL_PAPAN ;  P49923, LIPL_PIG   ;
DR   Q06000, LIPL_RAT   ;  Q29524, LIPL_SHEEP ;
//
```

In this example, the first line shows the EC (Enzyme Commission) number of lipoprotein lipase (second line). Alternative names of lipoprotein lipase are "clearing factor lipase", "diacylglycerol lipase", and "diglyceride lipase" (lines 3 through 5). The line starting with "CA" shows the catalytic activity of this enzyme. Comment lines start with "CC". The "PR" line shows references to the Prosite Documentation records, and the "DR" lines show references to Swiss-Prot records. Not of these entries are necessarily present in an Enzyme record.

In Biopython, an Enzyme record is represented by the `Bio.ExPASy.Enzyme.Record` class. This record derives from a Python dictionary and has keys corresponding to the two-letter codes used in Enzyme files. To read an Enzyme file containing one Enzyme record, use the `read` function in `Bio.ExPASy.Enzyme`:

```
>>> from Bio.ExPASy import Enzyme
>>> with open("lipoprotein.txt") as handle:
...     record = Enzyme.read(handle)
...
>>> record["ID"]
```

(continues on next page)

(continued from previous page)

```
'3.1.1.34'
>>> record["DE"]
'Lipoprotein lipase.'
>>> record["AN"]
['Clearing factor lipase.', 'Diacylglycerol lipase.', 'Diglyceride lipase.']
>>> record["CA"]
'Triacylglycerol + H(2)O = diacylglycerol + a carboxylate.'
>>> record["PR"]
['PDOC00110']
```

```
>>> record["CC"]
['Hydrolyzes triacylglycerols in chylomicrons and very low-density lipoproteins
(VLDL).', 'Also hydrolyzes diacylglycerol.']
>>> record["DR"]
[['P11151', 'LIPL_BOVIN'], ['P11153', 'LIPL_CAVPO'], ['P11602', 'LIPL_CHICK'],
['P55031', 'LIPL_FELCA'], ['P06858', 'LIPL_HUMAN'], ['P11152', 'LIPL_MOUSE'],
['O46647', 'LIPL_MUSVI'], ['P49060', 'LIPL_PAPAN'], ['P49923', 'LIPL_PIG'],
['Q06000', 'LIPL_RAT'], ['Q29524', 'LIPL_SHEEP']]
```

The `read` function raises a ValueError if no Enzyme record is found, and also if more than one Enzyme record is found.

The full set of Enzyme records can be downloaded as a single file (`enzyme.dat`) from the ExPASy FTP site, containing 4877 records (release of 3 March 2009). To parse such a file containing multiple Enzyme records, use the `parse` function in `Bio.ExPASy.Enzyme` to obtain an iterator:

```
>>> from Bio.ExPASy import Enzyme
>>> handle = open("enzyme.dat")
>>> records = Enzyme.parse(handle)
```

We can now iterate over the records one at a time. For example, we can make a list of all EC numbers for which an Enzyme record is available:

```
>>> ecnumbers = [record["ID"] for record in records]
```

14.5 Accessing the ExPASy server

Swiss-Prot, Prosite, and Prosite documentation records can be downloaded from the ExPASy web server at https://www.expasy.org. Four kinds of queries are available from ExPASy:

get_prodoc_entry
: To download a Prosite documentation record in HTML format

get_prosite_entry
: To download a Prosite record in HTML format

get_prosite_raw
: To download a Prosite or Prosite documentation record in raw format

get_sprot_raw
: To download a Swiss-Prot record in raw format

To access this web server from a Python script, we use the `Bio.ExPASy` module.

14.5.1 Retrieving a Swiss-Prot record

Let's say we are looking at chalcone synthases for Orchids (see section *A usage example* for some justification for looking for interesting things about orchids). Chalcone synthase is involved in flavanoid biosynthesis in plants, and flavanoids make lots of cool things like pigment colors and UV protectants.

If you do a search on Swiss-Prot, you can find three orchid proteins for Chalcone Synthase, id numbers O23729, O23730, O23731. Now, let's write a script which grabs these, and parses out some interesting information.

First, we grab the records, using the `get_sprot_raw()` function of `Bio.ExPASy`. This function is very nice since you can feed it an id and get back a handle to a raw text record (no HTML to mess with!). We can the use `Bio.SwissProt.read` to pull out the Swiss-Prot record, or `Bio.SeqIO.read` to get a SeqRecord. The following code accomplishes what I just wrote:

```
>>> from Bio import ExPASy
>>> from Bio import SwissProt

>>> accessions = ["O23729", "O23730", "O23731"]
>>> records = []

>>> for accession in accessions:
...     handle = ExPASy.get_sprot_raw(accession)
...     record = SwissProt.read(handle)
...     records.append(record)
...
```

If the accession number you provided to `ExPASy.get_sprot_raw` does not exist, then `SwissProt.read(handle)` will raise a `ValueError`. You can catch `ValueException` exceptions to detect invalid accession numbers:

```
>>> for accession in accessions:
...     handle = ExPASy.get_sprot_raw(accession)
...     try:
...         record = SwissProt.read(handle)
...     except ValueException:
...         print("WARNING: Accession %s not found" % accession)
...     records.append(record)
...
```

14.5.2 Searching with UniProt

Now, you may remark that I knew the records' accession numbers beforehand. Indeed, `get_sprot_raw()` needs either the entry name or an accession number. When you don't have them handy, you can use UniProt to search for them. You can also use the UniProt package to programmatically search for proteins.

For example, let's search for proteins from a specific organism (organism ID: 2697049) that have been reviewed. We can do this with the following code:

```
>>> from Bio import UniProt

>>> query = "(organism_id:2697049) AND (reviewed:true)"
>>> results = list(UniProt.search(query))
```

The `UniProt.search` method returns an iterator over the search results. The iterator returns one result at a time, fetching more results from UniProt as needed until all results are returned. We can efficiently create a list from this iterator for this specific query because this query only returns a few results (17 at the time of writing).

Let's try a search that returns more results. At the time of writing, there are 5,147 results for the query "Insulin AND (reviewed:true)". We can use slicing to get a list of the first 50 results.

```
>>> from Bio import UniProt
>>> from itertools import islice

>>> query = "Insulin AND (reviewed:true)"
>>> results = UniProt.search(query, batch_size=50)[:50]
```

You can get the total number of search results (regardless of the batch size) with the `len` method:

```
>>> from Bio import UniProt

>>> query = "Insulin AND (reviewed:true)"
>>> result_iterator = UniProt.search(query, batch_size=0)
>>> len(result_iterator)
5147
```

14.5.3 Retrieving Prosite and Prosite documentation records

Prosite and Prosite documentation records can be retrieved either in HTML format, or in raw format. To parse Prosite and Prosite documentation records with Biopython, you should retrieve the records in raw format. For other purposes, however, you may be interested in these records in HTML format.

To retrieve a Prosite or Prosite documentation record in raw format, use `get_prosite_raw()`. For example, to download a Prosite record and print it out in raw text format, use

```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prosite_raw("PS00001")
>>> text = handle.read()
>>> print(text)
```

To retrieve a Prosite record and parse it into a `Bio.Prosite.Record` object, use

```
>>> from Bio import ExPASy
>>> from Bio import Prosite
>>> handle = ExPASy.get_prosite_raw("PS00001")
>>> record = Prosite.read(handle)
```

The same function can be used to retrieve a Prosite documentation record and parse it into a `Bio.ExPASy.Prodoc.Record` object:

```
>>> from Bio import ExPASy
>>> from Bio.ExPASy import Prodoc
>>> handle = ExPASy.get_prosite_raw("PDOC00001")
>>> record = Prodoc.read(handle)
```

For non-existing accession numbers, `ExPASy.get_prosite_raw` returns a handle to an empty string. When faced with an empty string, `Prosite.read` and `Prodoc.read` will raise a ValueError. You can catch these exceptions to detect invalid accession numbers.

The functions `get_prosite_entry()` and `get_prodoc_entry()` are used to download Prosite and Prosite documentation records in HTML format. To create a web page showing one Prosite record, you can use

```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prosite_entry("PS00001")
>>> html = handle.read()
>>> with open("myprositerecord.html", "w") as out_handle:
...     out_handle.write(html)
...
```

and similarly for a Prosite documentation record:

```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prodoc_entry("PDOC00001")
>>> html = handle.read()
>>> with open("myprodocrecord.html", "w") as out_handle:
...     out_handle.write(html)
...
```

For these functions, an invalid accession number returns an error message in HTML format.

14.6 Scanning the Prosite database

ScanProsite allows you to scan protein sequences online against the Prosite database by providing a UniProt or PDB sequence identifier or the sequence itself. For more information about ScanProsite, please see the ScanProsite documentation as well as the documentation for programmatic access of ScanProsite.

You can use Biopython's `Bio.ExPASy.ScanProsite` module to scan the Prosite database from Python. This module both helps you to access ScanProsite programmatically, and to parse the results returned by ScanProsite. To scan for Prosite patterns in the following protein sequence:

```
MEHKEVVLLLLLFLKSGQGEPLDDYVNTQGASLFSVTKKQLGAGSIEECAAKCEEDEEFT
CRAFQYHSKEQQCVIMAENRKSSIIIRMRDVVLFEKKVYLSECKTGNGKNYRGTMSKTKN
```

you can use the following code:

```
>>> sequence = (
...     "MEHKEVVLLLLLFLKSGQGEPLDDYVNTQGASLFSVTKKQLGAGSIEECAAKCEEDEEFT"
...     "CRAFQYHSKEQQCVIMAENRKSSIIIRMRDVVLFEKKVYLSECKTGNGKNYRGTMSKTKN"
... )
>>> from Bio.ExPASy import ScanProsite
>>> handle = ScanProsite.scan(seq=sequence)
```

By executing `handle.read()`, you can obtain the search results in raw XML format. Instead, let's use `Bio.ExPASy.ScanProsite.read` to parse the raw XML into a Python object:

```
>>> result = ScanProsite.read(handle)
>>> type(result)
<class 'Bio.ExPASy.ScanProsite.Record'>
```

A `Bio.ExPASy.ScanProsite.Record` object is derived from a list, with each element in the list storing one ScanProsite hit. This object also stores the number of hits, as well as the number of search sequences, as returned by ScanProsite. This ScanProsite search resulted in six hits:

```
>>> result.n_seq
1
>>> result.n_match
1
>>> len(result)
1
>>> result[0]
{'sequence_ac': 'USERSEQ1', 'start': 16, 'stop': 98, 'signature_ac': 'PS50948', 'score':
↪'8.873', 'level': '0'}
```

Other ScanProsite parameters can be passed as keyword arguments; see the documentation for programmatic access of ScanProsite for more information. As an example, passing `lowscore=1` to include matches with low level scores lets use find one additional hit:

```
>>> handle = ScanProsite.scan(seq=sequence, lowscore=1)
>>> result = ScanProsite.read(handle)
>>> result.n_match
2
```

CHAPTER

FIFTEEN

GOING 3D: THE PDB MODULE

Bio.PDB is a Biopython module that focuses on working with crystal structures of biological macromolecules. Among other things, Bio.PDB includes a PDBParser class that produces a Structure object, which can be used to access the atomic data in the file in a convenient manner. There is limited support for parsing the information contained in the PDB header. PDB file format is no longer being modified or extended to support new content and PDBx/mmCIF became the standard PDB archive format in 2014. All the Worldwide Protein Data Bank (wwPDB) sites uses the macromolecular Crystallographic Information File (mmCIF) data dictionaries to describe the information content of PDB entries. mmCIF uses a flexible and extensible key-value pair format for representing macromolecular structural data and imposes no limitations for the number of atoms, residues or chains that can be represented in a single PDB entry (no split entries!).

15.1 Reading and writing crystal structure files

15.1.1 Reading an mmCIF file

First create an `MMCIFParser` object:

```
>>> from Bio.PDB.MMCIFParser import MMCIFParser
>>> parser = MMCIFParser()
```

Then use this parser to create a structure object from the mmCIF file:

```
>>> structure = parser.get_structure("1fat", "1fat.cif")
```

To have some more low level access to an mmCIF file, you can use the `MMCIF2Dict` class to create a Python dictionary that maps all mmCIF tags in an mmCIF file to their values. Whether there are multiple values (like in the case of tag `_atom_site.Cartn_y`, which holds the y coordinates of all atoms) or a single value (like the initial deposition date), the tag is mapped to a list of values. The dictionary is created from the mmCIF file as follows:

```
>>> from Bio.PDB.MMCIF2Dict import MMCIF2Dict
>>> mmcif_dict = MMCIF2Dict("1FAT.cif")
```

Example: get the solvent content from an mmCIF file:

```
>>> sc = mmcif_dict["_exptl_crystal.density_percent_sol"]
```

Example: get the list of the y coordinates of all atoms

```
>>> y_list = mmcif_dict["_atom_site.Cartn_y"]
```

15.1.2 Reading a BinaryCIF file

Create a `BinaryCIFParser` object:

```
>>> from Bio.PDB.binary_cif import BinaryCIFParser
>>> parser = BinaryCIFParser()
```

Call `get_structure` with the path to the BinaryCIF file:

```
>>> parser.get_structure("1GBT", "1gbt.bcif.gz")
<Structure id=1GBT>
```

15.1.3 Reading files in the MMTF format

You can use the direct MMTFParser to read a structure from a file:

```
>>> from Bio.PDB.mmtf import MMTFParser
>>> structure = MMTFParser.get_structure("PDB/4CUP.mmtf")
```

Or you can use the same class to get a structure by its PDB ID:

```
>>> structure = MMTFParser.get_structure_from_url("4CUP")
```

This gives you a Structure object as if read from a PDB or mmCIF file.

You can also have access to the underlying data using the external MMTF library which Biopython is using internally:

```
>>> from mmtf import fetch
>>> decoded_data = fetch("4CUP")
```

For example you can access just the X-coordinate.

```
>>> print(decoded_data.x_coord_list)
```

15.1.4 Reading a PDB file

First we create a `PDBParser` object:

```
>>> from Bio.PDB.PDBParser import PDBParser
>>> parser = PDBParser(PERMISSIVE=1)
```

The `PERMISSIVE` flag indicates that a number of common problems (see *Examples*) associated with PDB files will be ignored (but note that some atoms and/or residues will be missing). If the flag is not present a `PDBConstructionException` will be generated if any problems are detected during the parse operation.

The Structure object is then produced by letting the `PDBParser` object parse a PDB file (the PDB file in this case is called `pdb1fat.ent`, `1fat` is a user defined name for the structure):

```
>>> structure_id = "1fat"
>>> filename = "pdb1fat.ent"
>>> structure = parser.get_structure(structure_id, filename)
```

You can extract the header and trailer (simple lists of strings) of the PDB file from the PDBParser object with the `get_header` and `get_trailer` methods. Note however that many PDB files contain headers with incomplete or erroneous information. Many of the errors have been fixed in the equivalent mmCIF files. *Hence, if you are interested in the header information, it is a good idea to extract information from mmCIF files using the ``MMCIF2Dict`` tool described above, instead of parsing the PDB header.*

Now that is clarified, let's return to parsing the PDB header. The structure object has an attribute called `header` which is a Python dictionary that maps header records to their values.

Example:

```
>>> resolution = structure.header["resolution"]
>>> keywords = structure.header["keywords"]
```

The available keys are `name`, `head`, `deposition_date`, `release_date`, `structure_method`, `resolution`, `structure_reference` (which maps to a list of references), `journal_reference`, `author`, `compound` (which maps to a dictionary with various information about the crystallized compound), `has_missing_residues`, `missing_residues`, and `astral` (which maps to dictionary with additional information about the domain if present).

`has_missing_residues` maps to a bool that is True if at least one non-empty `REMARK 465` header line was found. In this case you should assume that the molecule used in the experiment has some residues for which no ATOM coordinates could be determined. `missing_residues` maps to a list of dictionaries with information about the missing residues. *The list of missing residues will be empty or incomplete if the PDB header does not follow the template from the PDB specification.*

The dictionary can also be created without creating a `Structure` object, ie. directly from the PDB file:

```
>>> from Bio.PDB import parse_pdb_header
>>> with open(filename, "r") as handle:
...     header_dict = parse_pdb_header(handle)
...
```

15.1.5 Reading a PQR file

In order to parse a PQR file, proceed in a similar manner as in the case of PDB files:

Create a `PDBParser` object, using the `is_pqr` flag:

```
>>> from Bio.PDB.PDBParser import PDBParser
>>> pqr_parser = PDBParser(PERMISSIVE=1, is_pqr=True)
```

The `is_pqr` flag set to `True` indicates that the file to be parsed is a PQR file, and that the parser should read the atomic charge and radius fields for each atom entry. Following the same procedure as for PQR files, a Structure object is then produced, and the PQR file is parsed.

```
>>> structure_id = "1fat"
>>> filename = "pdb1fat.ent"
>>> structure = parser.get_structure(structure_id, filename, is_pqr=True)
```

15.1.6 Reading a PDBML (PDB XML) file

Create a PDBMLParser object:

```
>>> from Bio.PDB.PDBMLParser import PDBMLParser
>>> pdbml_parser = PDBMLParser()
```

Call get_structure with a file path or file object containing the PDB structure in XML format:

```
>>> structure = pdbml_parser.get_structure("1GBT.xml")
```

15.1.7 Writing mmCIF files

The MMCIFIO class can be used to write structures to the mmCIF file format:

```
>>> io = MMCIFIO()
>>> io.set_structure(s)
>>> io.save("out.cif")
```

The Select class can be used in a similar way to PDBIO below. mmCIF dictionaries read using MMCIF2Dict can also be written:

```
>>> io = MMCIFIO()
>>> io.set_dict(d)
>>> io.save("out.cif")
```

15.1.8 Writing PDB files

Use the PDBIO class for this. It's easy to write out specific parts of a structure too, of course.

Example: saving a structure

```
>>> io = PDBIO()
>>> io.set_structure(s)
>>> io.save("out.pdb")
```

If you want to write out a part of the structure, make use of the Select class (also in PDBIO). Select has four methods:

- accept_model(model)
- accept_chain(chain)
- accept_residue(residue)
- accept_atom(atom)

By default, every method returns 1 (which means the model/chain/residue/atom is included in the output). By subclassing Select and returning 0 when appropriate you can exclude models, chains, etc. from the output. Cumbersome maybe, but very powerful. The following code only writes out glycine residues:

```
>>> class GlySelect(Select):
...     def accept_residue(self, residue):
...         if residue.get_name() == "GLY":
...             return True
...         else:
```

(continues on next page)

(continued from previous page)

```
...             return False
...
>>> io = PDBIO()
>>> io.set_structure(s)
>>> io.save("gly_only.pdb", GlySelect())
```

If this is all too complicated for you, the `Dice` module contains a handy `extract` function that writes out all residues in a chain between a start and end residue.

15.1.9 Writing PQR files

Use the `PDBIO` class as you would for a PDB file, with the flag `is_pqr=True`. The PDBIO methods can be used in the case of PQR files as well.

Example: writing a PQR file

```
>>> io = PDBIO(is_pqr=True)
>>> io.set_structure(s)
>>> io.save("out.pdb")
```

15.1.10 Writing MMTF files

To write structures to the MMTF file format:

```
>>> from Bio.PDB.mmtf import MMTFIO
>>> io = MMTFIO()
>>> io.set_structure(s)
>>> io.save("out.mmtf")
```

The `Select` class can be used as above. Note that the bonding information, secondary structure assignment and some other information contained in standard MMTF files is not written out as it is not easy to determine from the structure object. In addition, molecules that are grouped into the same entity in standard MMTF files are treated as separate entities by `MMTFIO`.

15.2 Structure representation

The overall layout of a `Structure` object follows the so-called SMCRA (Structure/Model/Chain/Residue/Atom) architecture:

- A structure consists of models
- A model consists of chains
- A chain consists of residues
- A residue consists of atoms

This is the way many structural biologists/bioinformaticians think about structure, and provides a simple but efficient way to deal with structure. Additional stuff is essentially added when needed. A UML diagram of the `Structure` object (forget about the `Disordered` classes for now) is shown in Fig. 1. Such a data structure is not necessarily best suited for the representation of the macromolecular content of a structure, but it is absolutely necessary for a good interpretation of the data present in a file that describes the structure (typically a PDB or MMCIF file). If this hierarchy

cannot represent the contents of a structure file, it is fairly certain that the file contains an error or at least does not describe the structure unambiguously. If a SMCRA data structure cannot be generated, there is reason to suspect a problem. Parsing a PDB file can thus be used to detect likely problems. We will give several examples of this in section *Examples*.

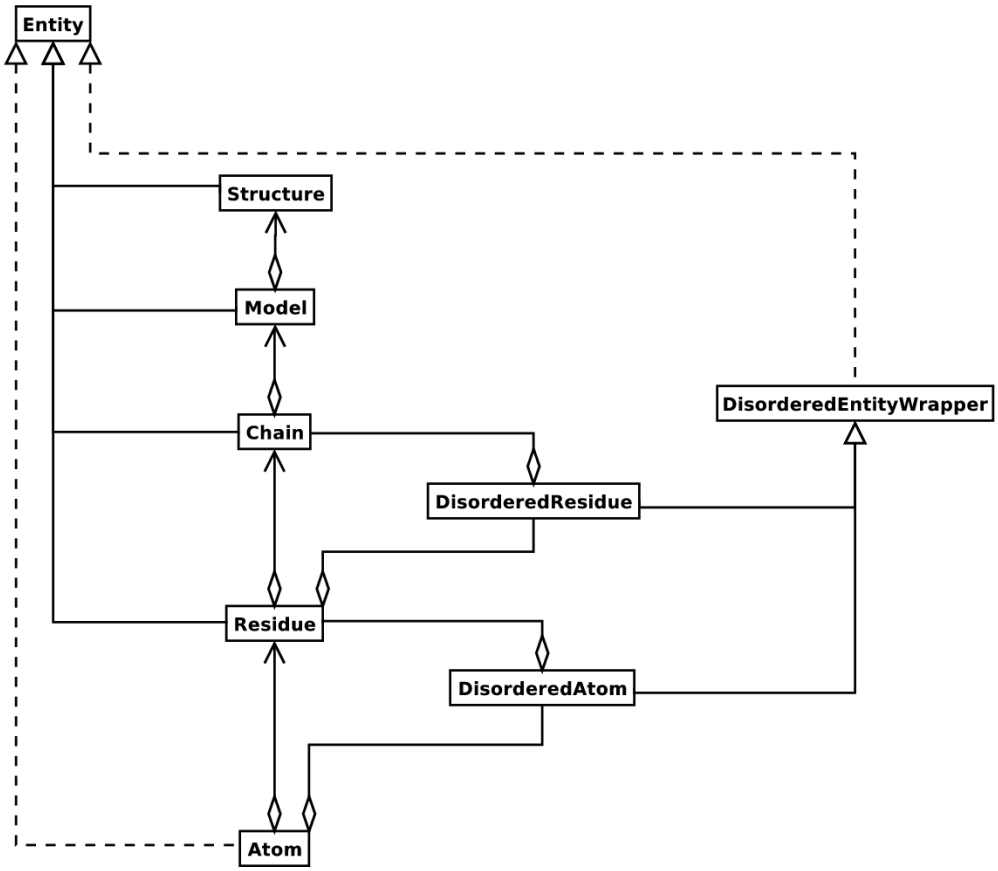


Fig. 1: UML diagram of SMCRA architecture of the `Structure` class.
This is used to represent a macromolecular structure. Full lines with diamonds denote aggregation, full lines with arrows denote referencing, full lines with triangles denote inheritance and dashed lines with triangles denote interface realization.

Structure, Model, Chain and Residue are all subclasses of the Entity base class. The Atom class only (partly) implements the Entity interface (because an Atom does not have children).

For each Entity subclass, you can extract a child by using a unique id for that child as a key (e.g. you can extract an Atom object from a Residue object by using an atom name string as a key, you can extract a Chain object from a Model object by using its chain identifier as a key).

Disordered atoms and residues are represented by DisorderedAtom and DisorderedResidue classes, which are both subclasses of the DisorderedEntityWrapper base class. They hide the complexity associated with disorder and behave exactly as Atom and Residue objects.

In general, a child Entity object (i.e. Atom, Residue, Chain, Model) can be extracted from its parent (i.e. Residue, Chain, Model, Structure, respectively) by using an id as a key.

```
>>> child_entity = parent_entity[child_id]
```

You can also get a list of all child Entities of a parent Entity object. Note that this list is sorted in a specific way (e.g. according to chain identifier for Chain objects in a Model object).

```
>>> child_list = parent_entity.get_list()
```

You can also get the parent from a child:

```
>>> parent_entity = child_entity.get_parent()
```

At all levels of the SMCRA hierarchy, you can also extract a *full id*. The full id is a tuple containing all id's starting from the top object (Structure) down to the current object. A full id for a Residue object e.g. is something like:

```
>>> full_id = residue.get_full_id()
>>> print(full_id)
("1abc", 0, "A", ("", 10, "A"))
```

This corresponds to:

- The Structure with id `"1abc"`
- The Model with id `0`
- The Chain with id `"A"`
- The Residue with id `("", 10, "A")`

The Residue id indicates that the residue is not a hetero-residue (nor a water) because it has a blank hetero field, that its sequence identifier is 10 and that its insertion code is `"A"`.

To get the entity's id, use the `get_id` method:

```
>>> entity.get_id()
```

You can check if the entity has a child with a given id by using the `has_id` method:

```
>>> entity.has_id(entity_id)
```

The length of an entity is equal to its number of children:

```
>>> nr_children = len(entity)
```

It is possible to delete, rename, add, etc. child entities from a parent entity, but this does not include any sanity checks (e.g. it is possible to add two residues with the same id to one chain). This really should be done via a nice Decorator class that includes integrity checking, but you can take a look at the code (Entity.py) if you want to use the raw interface.

15.2.1 Structure

The Structure object is at the top of the hierarchy. Its id is a user given string. The Structure contains a number of Model children. Most crystal structures (but not all) contain a single model, while NMR structures typically consist of several models. Disorder in crystal structures of large parts of molecules can also result in several models.

15.2.2 Model

The id of the Model object is an integer, which is derived from the position of the model in the parsed file (they are automatically numbered starting from 0). Crystal structures generally have only one model (with id 0), while NMR files usually have several models. Whereas many PDB parsers assume that there is only one model, the `Structure` class in `Bio.PDB` is designed such that it can easily handle PDB files with more than one model.

As an example, to get the first model from a Structure object, use

```
>>> first_model = structure[0]
```

The Model object stores a list of Chain children.

15.2.3 Chain

The id of a Chain object is derived from the chain identifier in the PDB/mmCIF file, and is a single character (typically a letter). Each Chain in a Model object has a unique id. As an example, to get the Chain object with identifier "A" from a Model object, use

```
>>> chain_A = model["A"]
```

The Chain object stores a list of Residue children.

15.2.4 Residue

A residue id is a tuple with three elements:

- The **hetero-field** (hetfield): this is
  - `'W'` in the case of a water molecule;
  - `'H_'` followed by the residue name for other hetero residues (e.g. `'H_GLC'` in the case of a glucose molecule);
  - blank for standard amino and nucleic acids.

  This scheme is adopted for reasons described in section *Associated problems*.
- The **sequence identifier** (resseq), an integer describing the position of the residue in the chain (e.g., 100);
- The **insertion code** (icode); a string, e.g. 'A'. The insertion code is sometimes used to preserve a certain desirable residue numbering scheme. A Ser 80 insertion mutant (inserted e.g. between a Thr 80 and an Asn 81 residue) could e.g. have sequence identifiers and insertion codes as follows: Thr 80 A, Ser 80 B, Asn 81. In this way the residue numbering scheme stays in tune with that of the wild type structure.

The id of the above glucose residue would thus be `('H_GLC', 100, 'A')`. If the hetero-flag and insertion code are blank, the sequence identifier alone can be used:

```
# Full id
>>> residue = chain[(" ", 100, " ")]
# Shortcut id
>>> residue = chain[100]
```

The reason for the hetero-flag is that many, many PDB files use the same sequence identifier for an amino acid and a hetero-residue or a water, which would create obvious problems if the hetero-flag was not used.

Unsurprisingly, a Residue object stores a set of Atom children. It also contains a string that specifies the residue name (e.g. "ASN") and the segment identifier of the residue (well known to X-PLOR users, but not used in the construction of the SMCRA data structure).

Let's look at some examples. Asn 10 with a blank insertion code would have residue id `(' ', 10, ' ')`. Water 10 would have residue id `('W', 10, ' ')`. A glucose molecule (a hetero residue with residue name GLC) with sequence identifier 10 would have residue id `('H_GLC', 10, ' ')`. In this way, the three residues (with the same insertion code and sequence identifier) can be part of the same chain because their residue id's are distinct.

In most cases, the hetflag and insertion code fields will be blank, e.g. `(' ', 10, ' ')`. In these cases, the sequence identifier can be used as a shortcut for the full id:

```
# use full id
>>> res10 = chain[(" ", 10, " ")]
# use shortcut
>>> res10 = chain[10]
```

Each Residue object in a Chain object should have a unique id. However, disordered residues are dealt with in a special way, as described in section *Point mutations*.

A Residue object has a number of additional methods:

```
>>> residue.get_resname()  # returns the residue name, e.g. "ASN"
>>> residue.is_disordered()  # returns 1 if the residue has disordered atoms
>>> residue.get_segid()  # returns the SEGID, e.g. "CHN1"
>>> residue.has_id(name)  # test if a residue has a certain atom
```

You can use `is_aa(residue)` to test if a Residue object is an amino acid.

15.2.5 Atom

The Atom object stores the data associated with an atom, and has no children. The id of an atom is its atom name (e.g. "OG" for the side chain oxygen of a Ser residue). An Atom id needs to be unique in a Residue. Again, an exception is made for disordered atoms, as described in section *Disordered atoms*.

The atom id is simply the atom name (eg. `'CA'`). In practice, the atom name is created by stripping all spaces from the atom name in the PDB file.

However, in PDB files, a space can be part of an atom name. Often, calcium atoms are called `'CA..'` in order to distinguish them from Cα atoms (which are called `'.CA.'`). In cases were stripping the spaces would create problems (ie. two atoms called `'CA'` in the same residue) the spaces are kept.

In a PDB file, an atom name consists of 4 chars, typically with leading and trailing spaces. Often these spaces can be removed for ease of use (e.g. an amino acid Cα atom is labeled ".CA." in a PDB file, where the dots represent spaces). To generate an atom name (and thus an atom id) the spaces are removed, unless this would result in a name collision in a Residue (i.e. two Atom objects with the same atom name and id). In the latter case, the atom name including spaces is tried. This situation can e.g. happen when one residue contains atoms with names ".CA." and "CA..", although this is not very likely.

The atomic data stored includes the atom name, the atomic coordinates (including standard deviation if present), the B factor (including anisotropic B factors and standard deviation if present), the altloc specifier and the full atom name including spaces. Less used items like the atom element number or the atomic charge sometimes specified in a PDB file are not stored.

To manipulate the atomic coordinates, use the `transform` method of the `Atom` object. Use the `set_coord` method to specify the atomic coordinates directly.

An Atom object has the following additional methods:

```
>>> a.get_name()  # atom name (spaces stripped, e.g. "CA")
>>> a.get_id()  # id (equals atom name)
>>> a.get_coord()  # atomic coordinates
>>> a.get_vector()  # atomic coordinates as Vector object
>>> a.get_bfactor()  # isotropic B factor
>>> a.get_occupancy()  # occupancy
>>> a.get_altloc()  # alternative location specifier
>>> a.get_sigatm()  # standard deviation of atomic parameters
>>> a.get_siguij()  # standard deviation of anisotropic B factor
>>> a.get_anisou()  # anisotropic B factor
>>> a.get_fullname()  # atom name (with spaces, e.g. ".CA.")
```

To represent the atom coordinates, siguij, anisotropic B factor and sigatm Numpy arrays are used.

The `get_vector` method returns a `Vector` object representation of the coordinates of the `Atom` object, allowing you to do vector operations on atomic coordinates. `Vector` implements the full set of 3D vector operations, matrix multiplication (left and right) and some advanced rotation-related operations as well.

As an example of the capabilities of Bio.PDB's `Vector` module, suppose that you would like to find the position of a Gly residue's Cβ atom, if it had one. Rotating the N atom of the Gly residue along the Cα-C bond over -120 degrees roughly puts it in the position of a virtual Cβ atom. Here's how to do it, making use of the `rotaxis` method (which can be used to construct a rotation around a certain axis) of the `Vector` module:

```
# get atom coordinates as vectors
>>> n = residue["N"].get_vector()
>>> c = residue["C"].get_vector()
>>> ca = residue["CA"].get_vector()
# center at origin
>>> n = n - ca
>>> c = c - ca
# find rotation matrix that rotates n
# -120 degrees along the ca-c vector
>>> rot = rotaxis(-pi * 120.0 / 180.0, c)
# apply rotation to ca-n vector
>>> cb_at_origin = n.left_multiply(rot)
# put on top of ca atom
>>> cb = cb_at_origin + ca
```

This example shows that it's possible to do some quite nontrivial vector operations on atomic data, which can be quite useful. In addition to all the usual vector operations (cross (use **), and dot (use *) product, angle, norm, etc.) and the above mentioned `rotaxis` function, the `Vector` module also has methods to rotate (`rotmat`) or reflect (`refmat`) one vector on top of another.

15.2.6 Extracting a specific `Atom/Residue/Chain/Model` from a Structure

These are some examples:

```
>>> model = structure[0]
>>> chain = model["A"]
>>> residue = chain[100]
>>> atom = residue["CA"]
```

Note that you can use a shortcut:

```
>>> atom = structure[0]["A"][100]["CA"]
```

15.3 Disorder

Bio.PDB can handle both disordered atoms and point mutations (i.e. a Gly and an Ala residue in the same position).

15.3.1 General approach

Disorder should be dealt with from two points of view: the atom and the residue points of view. In general, we have tried to encapsulate all the complexity that arises from disorder. If you just want to loop over all Cα atoms, you do not care that some residues have a disordered side chain. On the other hand it should also be possible to represent disorder completely in the data structure. Therefore, disordered atoms or residues are stored in special objects that behave as if there is no disorder. This is done by only representing a subset of the disordered atoms or residues. Which subset is picked (e.g. which of the two disordered OG side chain atom positions of a Ser residue is used) can be specified by the user.

15.3.2 Disordered atoms

Disordered atoms are represented by ordinary `Atom` objects, but all `Atom` objects that represent the same physical atom are stored in a `DisorderedAtom` object (see Fig. 1). Each `Atom` object in a `DisorderedAtom` object can be uniquely indexed using its altloc specifier. The `DisorderedAtom` object forwards all uncaught method calls to the selected Atom object, by default the one that represents the atom with the highest occupancy. The user can of course change the selected `Atom` object, making use of its altloc specifier. In this way atom disorder is represented correctly without much additional complexity. In other words, if you are not interested in atom disorder, you will not be bothered by it.

Each disordered atom has a characteristic altloc identifier. You can specify that a `DisorderedAtom` object should behave like the `Atom` object associated with a specific altloc identifier:

```
>>> atom.disordered_select("A")  # select altloc A atom
>>> print(atom.get_altloc())
"A"
>>> atom.disordered_select("B")  # select altloc B atom
>>> print(atom.get_altloc())
"B"
```

15.3.3 Disordered residues

Common case

The most common case is a residue that contains one or more disordered atoms. This is evidently solved by using DisorderedAtom objects to represent the disordered atoms, and storing the DisorderedAtom object in a Residue object just like ordinary Atom objects. The DisorderedAtom will behave exactly like an ordinary atom (in fact the atom with the highest occupancy) by forwarding all uncaught method calls to one of the Atom objects (the selected Atom object) it contains.

Point mutations

A special case arises when disorder is due to a point mutation, i.e. when two or more point mutants of a polypeptide are present in the crystal. An example of this can be found in PDB structure 1EN2.

Since these residues belong to a different residue type (e.g. let's say Ser 60 and Cys 60) they should not be stored in a single `Residue` object as in the common case. In this case, each residue is represented by one `Residue` object, and both `Residue` objects are stored in a single `DisorderedResidue` object (see Fig. 1).

The `DisorderedResidue` object forwards all uncaught methods to the selected `Residue` object (by default the last `Residue` object added), and thus behaves like an ordinary residue. Each `Residue` object in a `DisorderedResidue` object can be uniquely identified by its residue name. In the above example, residue Ser 60 would have id "SER" in the `DisorderedResidue` object, while residue Cys 60 would have id "CYS". The user can select the active `Residue` object in a `DisorderedResidue` object via this id.

Example: suppose that a chain has a point mutation at position 10, consisting of a Ser and a Cys residue. Make sure that residue 10 of this chain behaves as the Cys residue.

```
>>> residue = chain[10]
>>> residue.disordered_select("CYS")
```

In addition, you can get a list of all `Atom` objects (ie. all `DisorderedAtom` objects are 'unpacked' to their individual `Atom` objects) using the `get_unpacked_list` method of a `(Disordered)Residue` object.

15.4 Hetero residues

15.4.1 Associated problems

A common problem with hetero residues is that several hetero and non-hetero residues present in the same chain share the same sequence identifier (and insertion code). Therefore, to generate a unique id for each hetero residue, waters and other hetero residues are treated in a different way.

Remember that Residue object have the tuple (hetfield, resseq, icode) as id. The hetfield is blank (" ") for amino and nucleic acids, and a string for waters and other hetero residues. The content of the hetfield is explained below.

15.4.2 Water residues

The hetfield string of a water residue consists of the letter "W". So a typical residue id for a water is ("W", 1, " ").

15.4.3 Other hetero residues

The hetfield string for other hetero residues starts with "H_" followed by the residue name. A glucose molecule e.g. with residue name "GLC" would have hetfield "H_GLC". Its residue id could e.g. be ("H_GLC", 1, " ").

15.5 Navigating through a Structure object

15.5.1 Parse a PDB file, and extract some Model, Chain, Residue and Atom objects

```
>>> from Bio.PDB.PDBParser import PDBParser
>>> parser = PDBParser()
>>> structure = parser.get_structure("test", "1fat.pdb")
>>> model = structure[0]
>>> chain = model["A"]
>>> residue = chain[1]
>>> atom = residue["CA"]
```

15.5.2 Iterating through all atoms of a structure

```
>>> p = PDBParser()
>>> structure = p.get_structure("X", "pdb1fat.ent")
>>> for model in structure:
...     for chain in model:
...         for residue in chain:
...             for atom in residue:
...                 print(atom)
...
```

There is a shortcut if you want to iterate over all atoms in a structure:

```
>>> atoms = structure.get_atoms()
>>> for atom in atoms:
...     print(atom)
...
```

Similarly, to iterate over all atoms in a chain, use

```
>>> atoms = chain.get_atoms()
>>> for atom in atoms:
...     print(atom)
...
```

15.5.3 Iterating over all residues of a model

or if you want to iterate over all residues in a model:

```
>>> residues = model.get_residues()
>>> for residue in residues:
...     print(residue)
...
```

You can also use the `Selection.unfold_entities` function to get all residues from a structure:

```
>>> res_list = Selection.unfold_entities(structure, "R")
```

or to get all atoms from a chain:

```
>>> atom_list = Selection.unfold_entities(chain, "A")
```

Obviously, `A=atom`, `R=residue`, `C=chain`, `M=model`, `S=structure`. You can use this to go up in the hierarchy, e.g. to get a list of (unique) `Residue` or `Chain` parents from a list of `Atoms`:

```
>>> residue_list = Selection.unfold_entities(atom_list, "R")
>>> chain_list = Selection.unfold_entities(atom_list, "C")
```

For more info, see the API documentation.

15.5.4 Extract hetero residue from chain (e.g. glucose (GLC) moiety with resseq 10)

```
>>> residue_id = ("H_GLC", 10, " ")
>>> residue = chain[residue_id]
```

15.5.5 Print all hetero residues in chain

```
>>> for residue in chain.get_list():
...     residue_id = residue.get_id()
...     hetfield = residue_id[0]
...     if hetfield[0] == "H":
...         print(residue_id)
...
```

15.5.6 Print out coordinates of all CA atoms in structure with B factor over 50

```
>>> for model in structure.get_list():
...     for chain in model.get_list():
...         for residue in chain.get_list():
...             if residue.has_id("CA"):
...                 ca = residue["CA"]
...                 if ca.get_bfactor() > 50.0:
...                     print(ca.get_coord())
...
```

15.5.7 Print out all the residues that contain disordered atoms

```
>>> for model in structure.get_list():
...     for chain in model.get_list():
...         for residue in chain.get_list():
...             if residue.is_disordered():
...                 resseq = residue.get_id()[1]
...                 resname = residue.get_resname()
...                 model_id = model.get_id()
...                 chain_id = chain.get_id()
...                 print(model_id, chain_id, resname, resseq)
...
```

15.5.8 Loop over all disordered atoms, and select all atoms with altloc A (if present)

This will make sure that the SMCRA data structure will behave as if only the atoms with altloc A are present.

```
>>> for model in structure.get_list():
...     for chain in model.get_list():
...         for residue in chain.get_list():
...             if residue.is_disordered():
...                 for atom in residue.get_list():
...                     if atom.is_disordered():
...                         if atom.disordered_has_id("A"):
...                             atom.disordered_select("A")
...
```

15.5.9 Extracting polypeptides from a `Structure` object

To extract polypeptides from a structure, construct a list of `Polypeptide` objects from a `Structure` object using `PolypeptideBuilder` as follows:

```
>>> model_nr = 1
>>> polypeptide_list = build_peptides(structure, model_nr)
>>> for polypeptide in polypeptide_list:
...     print(polypeptide)
...
```

A Polypeptide object is simply a UserList of Residue objects, and is always created from a single Model (in this case model 1). You can use the resulting `Polypeptide` object to get the sequence as a `Seq` object or to get a list of Cα atoms as well. Polypeptides can be built using a C-N or a Cα-Cα distance criterion.

Example:

```
# Using C-N
>>> ppb = PPBuilder()
>>> for pp in ppb.build_peptides(structure):
...     print(pp.get_sequence())
...
# Using CA-CA
>>> ppb = CaPPBuilder()
>>> for pp in ppb.build_peptides(structure):
...     print(pp.get_sequence())
...
```

Note that in the above case only model 0 of the structure is considered by `PolypeptideBuilder`. However, it is possible to use `PolypeptideBuilder` to build `Polypeptide` objects from `Model` and `Chain` objects as well.

15.5.10 Obtaining the sequence of a structure

The first thing to do is to extract all polypeptides from the structure (as above). The sequence of each polypeptide can then easily be obtained from the `Polypeptide` objects. The sequence is represented as a Biopython `Seq` object.

Example:

```
>>> seq = polypeptide.get_sequence()
>>> seq
Seq('SNDIYFNFQRFNETNLILQRDASVSSSGQLRLTNLN')
```

15.6 Analyzing structures

15.6.1 Measuring distances

The minus operator for atoms has been overloaded to return the distance between two atoms.

```
# Get some atoms
>>> ca1 = residue1["CA"]
>>> ca2 = residue2["CA"]
# Simply subtract the atoms to get their distance
>>> distance = ca1 - ca2
```

15.6.2 Measuring angles

Use the vector representation of the atomic coordinates, and the `calc_angle` function from the `Vector` module:

```
>>> vector1 = atom1.get_vector()
>>> vector2 = atom2.get_vector()
>>> vector3 = atom3.get_vector()
>>> angle = calc_angle(vector1, vector2, vector3)
```

15.6.3 Measuring torsion angles

Use the vector representation of the atomic coordinates, and the `calc_dihedral` function from the `Vector` module:

```
>>> vector1 = atom1.get_vector()
>>> vector2 = atom2.get_vector()
>>> vector3 = atom3.get_vector()
>>> vector4 = atom4.get_vector()
>>> angle = calc_dihedral(vector1, vector2, vector3, vector4)
```

15.6.4 Internal coordinates - distances, angles, torsion angles, distance plots, etc

Protein structures are normally supplied in 3D XYZ coordinates relative to a fixed origin, as in a PDB or mmCIF file. The `internal_coords` module facilitates converting this system to and from bond lengths, angles and dihedral angles. In addition to supporting standard *psi, phi, chi*, etc. calculations on protein structures, this representation is invariant to translation and rotation, and the implementation exposes multiple benefits for structure analysis.

First load up some modules here for later examples:

```
>>> from Bio.PDB.PDBParser import PDBParser
>>> from Bio.PDB.Chain import Chain
>>> from Bio.PDB.internal_coords import *
>>> from Bio.PDB.PICIO import write_PIC, read_PIC, read_PIC_seq
>>> from Bio.PDB.ic_rebuild import write_PDB, IC_duplicate, structure_rebuild_test
>>> from Bio.PDB.SCADIO import write_SCAD
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.PDB.PDBIO import PDBIO
>>> import numpy as np
```

Accessing dihedrals, angles and bond lengths

We start with the simple case of computing internal coordinates for a structure:

```
>>> # load a structure as normal, get first chain
>>> parser = PDBParser()
>>> myProtein = parser.get_structure("1a8o", "1A8O.pdb")
>>> myChain = myProtein[0]["A"]
```

```
>>> # compute bond lengths, angles, dihedral angles
>>> myChain.atom_to_internal_coordinates(verbose=True)
chain break at THR  186  due to MaxPeptideBond (1.4 angstroms) exceeded
chain break at THR  216  due to MaxPeptideBond (1.4 angstroms) exceeded
```

The chain break warnings for 1A8O are suppressed by removing the `verbose=True` option above. To avoid the creation of a break and instead allow unrealistically long N-C bonds, override the class variable `MaxPeptideBond`, e.g.:

```
>>> IC_Chain.MaxPeptideBond = 4.0
>>> myChain.internal_coord = None  # force re-loading structure data with new cutoff
>>> myChain.atom_to_internal_coordinates(verbose=True)
```

At this point the values are available at both the chain and residue level. The first residue of 1A8O is HETATM MSE (selenomethionine), so we investigate residue 2 below using either canonical names or atom specifiers. Here we obtain the *chi1* dihedral and *tau* angles by name and by atom sequence, and the Cα-Cβ distance by specifying the atom pair:

```
>>> r2 = myChain.child_list[1]
>>> r2
<Residue ASP het=  resseq=152 icode= >
>>> r2ic = r2.internal_coord
>>> print(r2ic, ":", r2ic.pretty_str(), ":", r2ic.rbase, ":", r2ic.lc)
('1a8o', 0, 'A', (' ', 152, ' ')) : ASP  152  : (152, None, 'D') : D
>>> r2chi1 = r2ic.get_angle("chi1")
>>> print(round(r2chi1, 2))
```

(continues on next page)

(continued from previous page)

```
-144.86
>>> r2ic.get_angle("chi1") == r2ic.get_angle("N:CA:CB:CG")
True
>>> print(round(r2ic.get_angle("tau"), 2))
113.45
>>> r2ic.get_angle("tau") == r2ic.get_angle("N:CA:C")
True
>>> print(round(r2ic.get_length("CA:CB"), 2))
1.53
```

The `Chain.internal_coord` object holds arrays and dictionaries of hedra (3 bonded atoms) and dihedra (4 bonded atoms) objects. The dictionaries are indexed by tuples of `AtomKey` objects; `AtomKey` objects capture residue position, insertion code, 1 or 3-character residue name, atom name, altloc and occupancy.

Below we obtain the same *chi1* and *tau* angles as above by indexing the `Chain` arrays directly, using `AtomKey`s to index the `Chain` arrays:

```
>>> myCic = myChain.internal_coord

>>> r2chi1_object = r2ic.pick_angle("chi1")
>>> # or same thing (as for get_angle() above):
>>> r2chi1_object == r2ic.pick_angle("N:CA:CB:CG")
True
>>> r2chi1_key = r2chi1_object.atomkeys
>>> r2chi1_key  # r2chi1_key is tuple of AtomKeys
(152_D_N, 152_D_CA, 152_D_CB, 152_D_CG)

>>> r2chi1_index = myCic.dihedraNdx[r2chi1_key]
>>> # or same thing:
>>> r2chi1_index == r2chi1_object.ndx
True
>>> print(round(myCic.dihedraAngle[r2chi1_index], 2))
-144.86
>>> # also:
>>> r2chi1_object == myCic.dihedra[r2chi1_key]
True

>>> # hedra angles are similar:
>>> r2tau = r2ic.pick_angle("tau")
>>> print(round(myCic.hedraAngle[r2tau.ndx], 2))
113.45
```

Obtaining bond length data at the `Chain` level is more complicated (and not recommended). As shown here, multiple hedra will share a single bond in different positions:

```
>>> r2CaCb = r2ic.pick_length("CA:CB")  # returns list of hedra containing bond
>>> r2CaCb[0][0].atomkeys
(152_D_CB, 152_D_CA, 152_D_C)
>>> print(round(myCic.hedraL12[r2CaCb[0][0].ndx], 2))  # position 1-2
1.53
>>> r2CaCb[0][1].atomkeys
(152_D_N, 152_D_CA, 152_D_CB)
>>> print(round(myCic.hedraL23[r2CaCb[0][1].ndx], 2))  # position 2-3
```

(continues on next page)

(continued from previous page)

```
1.53
>>> r2CaCb[0][2].atomkeys
(152_D_CA, 152_D_CB, 152_D_CG)
>>> print(round(myCic.hedraL12[r2CaCb[0][2].ndx], 2))  # position 1-2
1.53
```

Please use the `Residue` level `set_length`:math:`` function instead.

Testing structures for completeness

Missing atoms and other issues can cause problems when rebuilding a structure. Use `structure_rebuild_test`:math:`` to determine quickly if a structure has sufficient data for a clean rebuild. Add `verbose=True` and/or inspect the result dictionary for more detail:

```
>>> # check myChain makes sense (can get angles and rebuild same structure)
>>> resultDict = structure_rebuild_test(myChain)
>>> resultDict["pass"]
True
```

Modifying and rebuilding structures

It's preferable to use the residue level `set_angle`:math:`` and `set_length`:math:`` facilities for modifying internal coordinates rather than directly accessing the `Chain` structures. While directly modifying hedra angles is safe, bond lengths appear in multiple overlapping hedra as noted above, and this is handled by `set_length`:math:`. When applied to a dihedral angle, ``set_angle`:math:`` will wrap the result to +/-180 and rotate adjacent dihedra as well (such as both bonds for an isoleucine *chi1* angle - which is probably what you want).

```
>>> # rotate residue 2 chi1 angle by -120 degrees
>>> r2ic.set_angle("chi1", r2chi1 - 120.0)
>>> print(round(r2ic.get_angle("chi1"), 2))
95.14
>>> r2ic.set_length("CA:CB", 1.49)
>>> print(round(myCic.hedraL12[r2CaCb[0][0].ndx], 2))  # Cb-Ca-C position 1-2
1.49
```

Rebuilding a structure from internal coordinates is a simple call to `internal_to_atom_coordinates()`:

```
>>> myChain.internal_to_atom_coordinates()

>>> # just for proof:
>>> myChain.internal_coord = None  # all internal_coord data removed, only atoms left
>>> myChain.atom_to_internal_coordinates()  # re-generate internal coordinates
>>> r2ic = myChain.child_list[1].internal_coord
>>> print(round(r2ic.get_angle("chi1"), 2))  # show measured values match what was set␣
↪above
95.14
>>> print(round(myCic.hedraL23[r2CaCb[0][1].ndx], 2))  # N-Ca-Cb position 2-3
1.49
```

The generated structure can be written with PDBIO, as normal:

```
write_PDB(myProtein, "myChain.pdb")
# or just the ATOM records without headers:
io = PDBIO()
io.set_structure(myProtein)
io.save("myChain2.pdb")
```

Protein Internal Coordinate (.pic) files and default values

A file format is defined in the `PICIO` module to describe protein chains as hedra and dihedra relative to initial coordinates. All parts of the file other than the residue sequence information (e.g. `('1A8O', 0, 'A', (' ', 153, ' ')) ILE`) are optional, and will be filled in with default values if not specified and `read_PIC`:math:`` is called with the `defaults=True` option. Default values are calculated from Sep 2019 Dunbrack cullpdb_pc20_res2.2_R1.0.

Here we write 'myChain' as a `.pic` file of internal coordinate specifications and then read it back in as 'myProtein2'.

```
# write chain as 'protein internal coordinates' (.pic) file
write_PIC(myProtein, "myChain.pic")
# read .pic file
myProtein2 = read_PIC("myChain.pic")
```

As all internal coordinate values can be replaced with defaults, `PICIO.read_PIC_seq`:math:`` is supplied as a utility function to create a valid (mostly helical) default structure from an input sequence:

```
# create default structure for random sequence by reading as .pic file
myProtein3 = read_PIC_seq(
    SeqRecord(
        Seq("GAVLIMFPSTCNQYWDEHKR"),
        id="1RND",
        description="my random sequence",
    )
)
myProtein3.internal_to_atom_coordinates()
write_PDB(myProtein3, "myRandom.pdb")
```

It may be of interest to explore the accuracy required in e.g. *omega* angles (180.0), hedra angles and/or bond lengths when generating structures from internal coordinates. The picFlags option to `write_PIC`:math:`` enables this, allowing the selection of data to be written to the .pic file vs. left unspecified to get default values.

Various combinations are possible and some presets are supplied, for example `classic` will write only *psi, phi, tau*, proline *omega* and sidechain *chi* angles to the .pic file:

```
write_PIC(myProtein, "myChain.pic", picFlags=IC_Residue.pic_flags.classic)
myProtein2 = read_PIC("myChain.pic", defaults=True)
```

Accessing the all-atom AtomArray

All 3D XYZ coordinates in Biopython `Atom` objects are moved to a single large array in the `Chain` class and replaced by Numpy 'views' into this array in an early step of `atom_to_internal_coordinates`:math:`. Software accessing Biopython ``Atom` coordinates is not affected, but the new array may offer efficiencies for future work.

Unlike the `Atom` XYZ coordinates, `AtomArray` coordinates are homogeneous, meaning they are arrays like `[ x y z 1.0]` with 1.0 as the fourth element. This facilitates efficient transformation using combined translation and rotation matrices throughout the `internal_coords` module. There is a corresponding `AtomArrayIndex` dictionary, mapping `AtomKeys` to their coordinates.

Here we demonstrate reading coordinates for a specific Cβ atom from the array, then show that modifying the array value modifies the `Atom` object at the same time:

```
>>> # access the array of all atoms for the chain, e.g. r2 above is residue 152 C-beta
>>> r2_cBeta_index = myChain.internal_coord.atomArrayIndex[AtomKey("152_D_CB")]
>>> r2_cBeta_coords = myChain.internal_coord.atomArray[r2_cBeta_index]
>>> print(np.round(r2_cBeta_coords, 2))
[-0.75 -1.18 -0.51  1.  ]

>>> # the Biopython Atom coord array is now a view into atomArray, so
>>> assert r2_cBeta_coords[1] == r2["CB"].coord[1]
>>> r2_cBeta_coords[1] += 1.0  # change the Y coord 1 angstrom
>>> assert r2_cBeta_coords[1] == r2["CB"].coord[1]
>>> # they are always the same (they share the same memory)
>>> r2_cBeta_coords[1] -= 1.0  # restore
```

Note that it is easy to 'break' the view linkage between the Atom coord arrays and the chain atomArray. When modifying Atom coordinates directly, use syntax for an element-by-element copy to avoid this:

```
# use these:
myAtom1.coord[:] = myAtom2.coord
myAtom1.coord[...] = myAtom2.coord
myAtom1.coord[:] = [1, 2, 3]
for i in range(3):
    myAtom1.coord[i] = myAtom2.coord[i]

# do not use:
myAtom1.coord = myAtom2.coord
myAtom1.coord = [1, 2, 3]
```

Using the `atomArrayIndex` and knowledge of the `AtomKey` class enables us to create Numpy 'selectors', as shown below to extract an array of only the Cα atom coordinates:

```
>>> # create a selector to filter just the C-alpha atoms from the all atom array
>>> atmNameNdx = AtomKey.fields.atm
>>> aaI = myChain.internal_coord.atomArrayIndex
>>> CaSelect = [aaI.get(k) for k in aaI.keys() if k.akl[atmNameNdx] == "CA"]
>>> # now the ordered array of C-alpha atom coordinates is:
>>> CA_coords = myChain.internal_coord.atomArray[CaSelect]
>>> # note this uses Numpy fancy indexing, so CA_coords is a new copy
>>> # (if you modify it, the original atomArray is unaffected)
```

Distance Plots

A benefit of the `atomArray` is that generating a distance plot from it is a single line of `Numpy` code:

```
np.linalg.norm(atomArray[:, None, :] - atomArray[None, :, :], axis=-1)
```

Despite its briefness, the idiom cam be difficult to remember and in the form above generates all-atom distances rather than the classic Cα plot as may be desired. The `distance_plot`:math:`` method wraps the line above and accepts an optional selector like `CaSelect` defined in the previous section. See Fig. 2.

```
# create a C-alpha distance plot
caDistances = myChain.internal_coord.distance_plot(CaSelect)
# display with e.g. MatPlotLib:
import matplotlib.pyplot as plt

plt.imshow(caDistances, cmap="hot", interpolation="nearest")
plt.show()
```

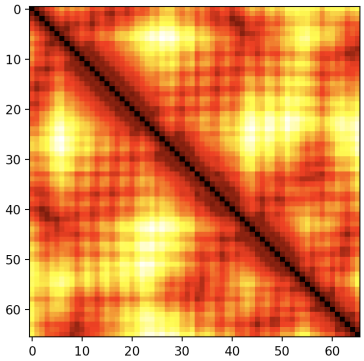


Fig. 2: C-alpha distance plot for PDB file 1A8O (HIV capsid C-terminal domain)

Building a structure from a distance plot

The all-atom distance plot is another representation of a protein structure, also invariant to translation and rotation but lacking in chirality information (a mirror-image structure will generate the same distance plot). By combining the distance matrix with the signs of each dihedral angle, it is possible to regenerate the internal coordinates.

This work uses equations developed by Blue, the Hedronometer, discussed in https://math.stackexchange.com/a/49340/409 and further in http://daylateanddollarshort.com/mathdocs/Heron-like-Results-for-Tetrahedral-Volume.pdf.

To begin, we extract the distances and chirality values from 'myChain':

```
>>> ## create the all-atom distance plot
>>> distances = myCic.distance_plot()
>>> ## get the signs of the dihedral angles
>>> chirality = myCic.dihedral_signs()
```

We need a valid data structure matching 'myChain' to correctly rebuild it; using `read_PIC_seq`:math:`` above would work in the general case, but the 1A8O example used here has some ALTLOC complexity which the sequence alone would not generate. For demonstration the easiest approach is to simply duplicate the 'myChain' structure, but we set all the atom and internal coordinate chain arrays to 0s (only for demonstration) just to be certain there is no data coming through from the original structure:

```
>>> ## get new, empty data structure : copy data structure from myChain
>>> myChain2 = IC_duplicate(myChain)[0]["A"]
>>> cic2 = myChain2.internal_coord

>>> ## clear the new atomArray and di/hedra value arrays, just for proof
>>> cic2.atomArray = np.zeros((cic2.AAsiz, 4), dtype=np.float64)
>>> cic2.dihedraAngle[:] = 0.0
>>> cic2.hedraAngle[:] = 0.0
>>> cic2.hedraL12[:] = 0.0
>>> cic2.hedraL23[:] = 0.0
```

The approach is to regenerate the internal coordinates from the distance plot data, then generate the atom coordinates from the internal coordinates as shown above. To place the final generated structure in the same coordinate space as the starting structure, we copy just the coordinates for the first three N-Cα-C atoms from the chain start of 'myChain' to the 'myChain2' structure (this is only needed to demonstrate equivalence at end):

```
>>> ## copy just the first N-Ca-C coords so structures will superimpose:
>>> cic2.copy_initNCaCs(myChain.internal_coord)
```

The `distance_to_internal_coordinates`:math:`` routine needs arrays of the six inter-atom distances for each dihedron for the target structure. The convenience routine `distplot_to_dh_arrays`:math:`` extracts these values from the previously generated distance matrix as needed, and may be replaced by a user method to write these data to the arrays in the `Chain.internal_coords` object.

```
>>> ## copy distances to chain arrays:
>>> cic2.distplot_to_dh_arrays(distances, chirality)
>>> ## compute angles and dihedral angles from distances:
>>> cic2.distance_to_internal_coordinates()
```

The steps below generate the atom coordinates from the newly generated 'myChain2' internal coordinates, then use the Numpy `allclose`:math:`` routine to confirm that all values match to better than PDB file resolution:

```
>>> ## generate XYZ coordinates from internal coordinates:
>>> myChain2.internal_to_atom_coordinates()
>>> ## confirm result atomArray matches original structure:
>>> np.allclose(cic2.atomArray, myCic.atomArray)
True
```

Note that this procedure does not use the entire distance matrix, but only the six local distances between the four atoms of each dihedral angle.

Superimposing residues and their neighborhoods

The `internal_coords` module relies on transforming atom coordinates between different coordinate spaces for both calculation of torsion angles and reconstruction of structures. Each dihedron has a coordinate space transform placing its first atom on the XZ plane, second atom at the origin, and third atom on the +Z axis, as well as a corresponding reverse transform which will return it to the coordinates in the original structure. These transform matrices are available to use as shown below. By judicious choice of a reference dihedron, pairwise and higher order residue intereactions can be investigated and visualized across multiple protein structures, e.g. Fig. 3.

This example superimposes each PHE residue in a chain on its N-Cα-Cβ atoms, and presents all PHEs in the chain in the respective coordinate space as a simple demonstration. A more realistic exploration of pairwise sidechain interactions would examine a dataset of structures and filter for interaction classes as discussed in the relevant literature.

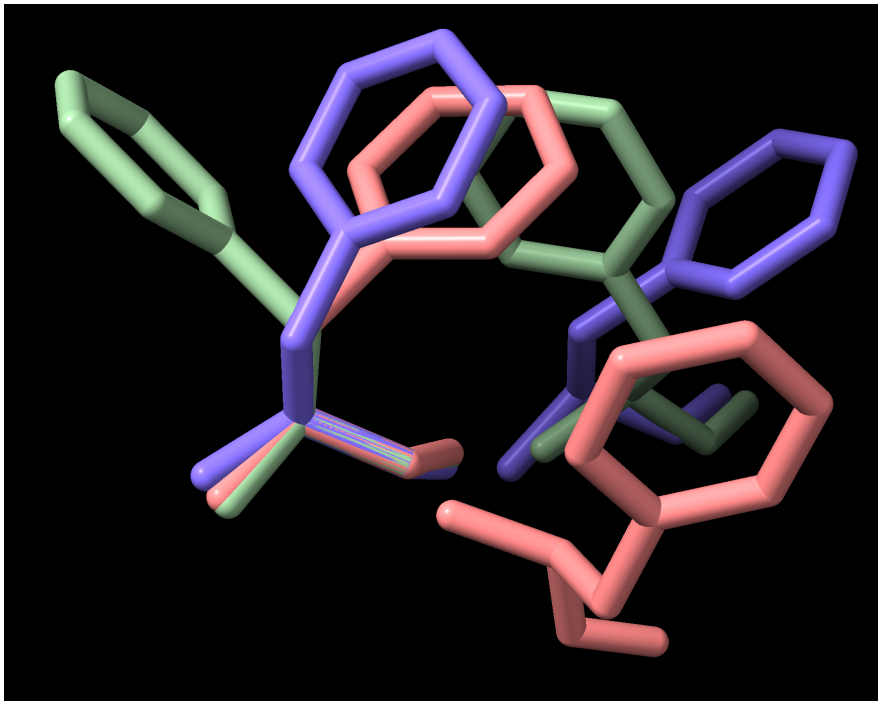


Fig. 3: Neighboring phenylalanine sidechains in PDB file 3PBL (human dopamine D3 receptor)

```
# superimpose all phe-phe pairs - quick hack just to demonstrate concept
# for analyzing pairwise residue interactions.  Generates PDB ATOM records
# placing each PHE at origin and showing all other PHEs in environment

## shorthand for key variables:
cic = myChain.internal_coord
resNameNdx = AtomKey.fields.resname
aaNdx = cic.atomArrayIndex

## select just PHE atoms:
pheAtomSelect = [aaNdx.get(k) for k in aaNdx.keys() if k.akl[resNameNdx] == "F"]
aaF = cic.atomArray[pheAtomSelect]  # numpy fancy indexing makes COPY not view

for ric in cic.ordered_aa_ic_list:  # internal_coords version of get_residues()
    if ric.lc == "F":  # if PHE, get transform matrices for chi1 dihedral
        chi1 = ric.pick_angle("chi1")  # N:CA:CB:CG space has C-alpha at origin
        cst = np.transpose(chi1.cst)  # transform TO chi1 space
        # rcst = np.transpose(chi1.rcst)  # transform FROM chi1 space (not needed here)
        cic.atomArray[pheAtomSelect] = aaF.dot(cst)  # transform just the PHEs
        for res in myChain.get_residues():  # print PHEs in new coordinate space
            if res.resname in ["PHE"]:
                print(res.internal_coord.pdb_residue_string())
        cic.atomArray[pheAtomSelect] = aaF  # restore coordinate space from copy
```

3D printing protein structures

OpenSCAD (https://openscad.org) is a language for creating solid 3D CAD objects. The algorithm to construct a protein structure from internal coordinates is supplied in OpenSCAD with data describing a structure, such that a model can be generated suitable for 3D printing. While other software can generate STL data as a rendering option for 3D printing (e.g. Chimera, https://www.cgl.ucsf.edu/chimera/), this approach generates spheres and cylinders as output and is therefore more amenable to modifications relevant to 3D printing protein structures. Individual residues and bonds can be selected in the OpenSCAD code for special handling, such as highlighting by size or adding rotatable bonds in specific positions (see https://www.thingiverse.com/thing:3957471 for an example).

```
# write OpenSCAD program of spheres and cylinders to 3d print myChain backbone
## set atom load filter to accept backbone only:
IC_Residue.accept_atoms = IC_Residue.accept_backbone
## set chain break cutoff very high to bridge missing residues with long bonds
IC_Chain.MaxPeptideBond = 4.0
## delete existing data to force re-read of all atoms with attributes set above:
myChain.internal_coord = None
write_SCAD(myChain, "myChain.scad", scale=10.0)
```

`internal_coords` control attributes

A few control attributes are available in the `internal_coords` classes to modify or filter data as internal coordinates are calculated. These are listed in Table *Control attributes in Bio.PDB.internal_coords.*:

Table 1: Control attributes in Bio.PDB.internal_coords.

Class	Attribute	Default	Effect
Atom-Key	d2h	False	Convert D atoms to H if True
IC_Chain	MaxPep-tideBond	1.4	Max C-N length w/o chain break; make large to link over missing residues for 3D models
IC_Residu	ac-cept_atoms	mainchain, hydro-gen atoms	override to remove some or all sidechains, H's, D's
IC_Residu	ac-cept_resname:	CYG, YCM, UNK	3-letter names for HETATMs to process, backbone only unless added to ic_data.py
IC_Residu	gly_Cbeta	False	override to generate Gly Cβ atoms based on database averages

15.6.5 Determining atom-atom contacts

Use `NeighborSearch` to perform neighbor lookup. The neighbor lookup is done using a KD tree module written in C (see the `KDTree` class in module `Bio.PDB.kdtrees`), making it very fast. It also includes a fast method to find all point pairs within a certain distance of each other.

15.6.6 Superimposing two structures

Use a `Superimposer` object to superimpose two coordinate sets. This object calculates the rotation and translation matrix that rotates two lists of atoms on top of each other in such a way that their RMSD is minimized. Of course, the two lists need to contain the same number of atoms. The `Superimposer` object can also apply the rotation/translation to a list of atoms. The rotation and translation are stored as a tuple in the `rotran` attribute of the `Superimposer` object (note that the rotation is right multiplying!). The RMSD is stored in the `rmsd` attribute.

The algorithm used by `Superimposer` comes from Golub & Van Loan [Golub1989] and makes use of singular value decomposition (this is implemented in the general `Bio.SVDSuperimposer` module).

Example:

```
>>> sup = Superimposer()
# Specify the atom lists
# 'fixed' and 'moving' are lists of Atom objects
# The moving atoms will be put on the fixed atoms
>>> sup.set_atoms(fixed, moving)
# Print rotation/translation/rmsd
>>> print(sup.rotran)
>>> print(sup.rms)
# Apply rotation/translation to the moving atoms
>>> sup.apply(moving)
```

To superimpose two structures based on their active sites, use the active site atoms to calculate the rotation/translation matrices (as above), and apply these to the whole molecule.

15.6.7 Mapping the residues of two related structures onto each other

First, create an alignment file in FASTA format, then use the `StructureAlignment` class. This class can also be used for alignments with more than two structures.

15.6.8 Calculating the Half Sphere Exposure

Half Sphere Exposure (HSE) is a new, 2D measure of solvent exposure [Hamelryck2005]. Basically, it counts the number of Cα atoms around a residue in the direction of its side chain, and in the opposite direction (within a radius of 13 Å. Despite its simplicity, it outperforms many other measures of solvent exposure.

HSE comes in two flavors: HSEα and HSEβ. The former only uses the Cα atom positions, while the latter uses the Cα and Cβ atom positions. The HSE measure is calculated by the `HSExposure` class, which can also calculate the contact number. The latter class has methods which return dictionaries that map a `Residue` object to its corresponding HSEα, HSEβ and contact number values.

Example:

```
>>> model = structure[0]
>>> hse = HSExposure()
# Calculate HSEalpha
>>> exp_ca = hse.calc_hs_exposure(model, option="CA3")
# Calculate HSEbeta
>>> exp_cb = hse.calc_hs_exposure(model, option="CB")
# Calculate classical coordination number
>>> exp_fs = hse.calc_fs_exposure(model)
# Print HSEalpha for a residue
>>> print(exp_ca[some_residue])
```

15.6.9 Determining the secondary structure

For this functionality, you need to install DSSP (and obtain a license for it — free for academic use, see https://swift.cmbi.umcn.nl/gv/dssp/). Then use the `DSSP` class, which maps `Residue` objects to their secondary structure (and accessible surface area). The DSSP codes are listed in Table *DSSP codes in Bio.PDB.*. Note that DSSP (the program, and thus by consequence the class) cannot handle multiple models!

Table 2: DSSP codes in Bio.PDB.

Code	Secondary structure
H	α-helix
B	Isolated β-bridge residue
E	Strand
G	3-10 helix
I	Π-helix
T	Turn
S	Bend
•	Other

The `DSSP` class can also be used to calculate the accessible surface area of a residue. But see also section *Calculating the residue depth*.

15.6.10 Calculating the residue depth

Residue depth is the average distance of a residue's atoms from the solvent accessible surface. It's a fairly new and very powerful parameterization of solvent accessibility. For this functionality, you need to install Michel Sanner's MSMS program (https://www.scripps.edu/sanner/html/msms_home.html). Then use the `ResidueDepth` class. This class behaves as a dictionary which maps `Residue` objects to corresponding (residue depth, Cα depth) tuples. The Cα depth is the distance of a residue's Cα atom to the solvent accessible surface.

Example:

```
>>> model = structure[0]
>>> rd = ResidueDepth(model, pdb_file)
>>> residue_depth, ca_depth = rd[some_residue]
```

You can also get access to the molecular surface itself (via the `get_surface` function), in the form of a Numeric Python array with the surface points.

15.7 Common problems in PDB files

It is well known that many PDB files contain semantic errors (not the structures themselves, but their representation in PDB files). Bio.PDB tries to handle this in two ways. The PDBParser object can behave in two ways: a restrictive way and a permissive way, which is the default.

Example:

```
# Permissive parser
>>> parser = PDBParser(PERMISSIVE=1)
>>> parser = PDBParser()  # The same (default)
# Strict parser
>>> strict_parser = PDBParser(PERMISSIVE=0)
```

In the permissive state (DEFAULT), PDB files that obviously contain errors are "corrected" (i.e. some residues or atoms are left out). These errors include:

- Multiple residues with the same identifier
- Multiple atoms with the same identifier (taking into account the altloc identifier)

These errors indicate real problems in the PDB file (for details see Hamelryck and Manderick, 2003 [Hamelryck2003A]). In the restrictive state, PDB files with errors cause an exception to occur. This is useful to find errors in PDB files.

Some errors however are automatically corrected. Normally each disordered atom should have a non-blank altloc identifier. However, there are many structures that do not follow this convention, and have a blank and a non-blank identifier for two disordered positions of the same atom. This is automatically interpreted in the right way.

Sometimes a structure contains a list of residues belonging to chain A, followed by residues belonging to chain B, and again followed by residues belonging to chain A, i.e. the chains are 'broken'. This is also correctly interpreted.

15.7.1 Examples

The PDBParser/Structure class was tested on about 800 structures (each belonging to a unique SCOP superfamily). This takes about 20 minutes, or on average 1.5 seconds per structure. Parsing the structure of the large ribosomal subunit (1FKK), which contains about 64000 atoms, takes 10 seconds on a 1000 MHz PC.

Three exceptions were generated in cases where an unambiguous data structure could not be built. In all three cases, the likely cause is an error in the PDB file that should be corrected. Generating an exception in these cases is much better than running the chance of incorrectly describing the structure in a data structure.

Duplicate residues

One structure contains two amino acid residues in one chain with the same sequence identifier (resseq 3) and icode. Upon inspection it was found that this chain contains the residues Thr A3, ..., Gly A202, Leu A3, Glu A204. Clearly, Leu A3 should be Leu A203. A couple of similar situations exist for structure 1FFK (which e.g. contains Gly B64, Met B65, Glu B65, Thr B67, i.e. residue Glu B65 should be Glu B66).

Duplicate atoms

Structure 1EJG contains a Ser/Pro point mutation in chain A at position 22. In turn, Ser 22 contains some disordered atoms. As expected, all atoms belonging to Ser 22 have a non-blank altloc specifier (B or C). All atoms of Pro 22 have altloc A, except the N atom which has a blank altloc. This generates an exception, because all atoms belonging to two residues at a point mutation should have non-blank altloc. It turns out that this atom is probably shared by Ser and Pro 22, as Ser 22 misses the N atom. Again, this points to a problem in the file: the N atom should be present in both the Ser and the Pro residue, in both cases associated with a suitable altloc identifier.

15.7.2 Automatic correction

Some errors are quite common and can be easily corrected without much risk of making a wrong interpretation. These cases are listed below.

A blank altloc for a disordered atom

Normally each disordered atom should have a non-blank altloc identifier. However, there are many structures that do not follow this convention, and have a blank and a non-blank identifier for two disordered positions of the same atom. This is automatically interpreted in the right way.

Broken chains

Sometimes a structure contains a list of residues belonging to chain A, followed by residues belonging to chain B, and again followed by residues belonging to chain A, i.e. the chains are "broken". This is correctly interpreted.

15.7.3 Fatal errors

Sometimes a PDB file cannot be unambiguously interpreted. Rather than guessing and risking a mistake, an exception is generated, and the user is expected to correct the PDB file. These cases are listed below.

Duplicate residues

All residues in a chain should have a unique id. This id is generated based on:

- The sequence identifier (resseq).
- The insertion code (icode).
- The hetfield string ("W" for waters and "H_" followed by the residue name for other hetero residues)
- The residue names of the residues in the case of point mutations (to store the Residue objects in a DisorderedResidue object).

If this does not lead to a unique id something is quite likely wrong, and an exception is generated.

Duplicate atoms

All atoms in a residue should have a unique id. This id is generated based on:

- The atom name (without spaces, or with spaces if a problem arises).
- The altloc specifier.

If this does not lead to a unique id something is quite likely wrong, and an exception is generated.

15.8 Accessing the Protein Data Bank

15.8.1 Downloading structures from the Protein Data Bank

Structures can be downloaded from the PDB (Protein Data Bank) by using the `retrieve_pdb_file` method on a `PDBList` object. The argument for this method is the PDB identifier of the structure.

```
>>> pdbl = PDBList()
>>> pdbl.retrieve_pdb_file("1FAT")
```

The `PDBList` class can also be used as a command-line tool:

```
python PDBList.py 1fat
```

The downloaded file will be called `pdb1fat.ent` and stored in the current working directory. Note that the `retrieve_pdb_file` method also has an optional argument `pdir` that specifies a specific directory in which to store the downloaded PDB files.

The `retrieve_pdb_file` method also has some options to specify the compression format used for the download, and the program used for local decompression (default `.Z` format and `gunzip`). In addition, the PDB ftp site can be specified upon creation of the `PDBList` object. By default, the server of the Worldwide Protein Data Bank (ftp://ftp.wwpdb.org/pub/pdb/data/structures/divided/pdb/) is used. See the API documentation for more details. Thanks again to Kristian Rother for donating this module.

15.8.2 Downloading the entire PDB

The following commands will store all PDB files in the `/data/pdb` directory:

```
python PDBList.py all /data/pdb

python PDBList.py all /data/pdb -d
```

The API method for this is called `download_entire_pdb`. Adding the `-d` option will store all files in the same directory. Otherwise, they are sorted into PDB-style subdirectories according to their PDB ID's. Depending on the traffic, a complete download will take 2-4 days.

15.8.3 Keeping a local copy of the PDB up to date

This can also be done using the `PDBList` object. One simply creates a `PDBList` object (specifying the directory where the local copy of the PDB is present) and calls the `update_pdb` method:

```
>>> pl = PDBList(pdb="/data/pdb")
>>> pl.update_pdb()
```

One can of course make a weekly `cronjob` out of this to keep the local copy automatically up-to-date. The PDB ftp site can also be specified (see API documentation).

`PDBList` has some additional methods that can be of use. The `get_all_obsolete` method can be used to get a list of all obsolete PDB entries. The `changed_this_week` method can be used to obtain the entries that were added, modified or obsoleted during the current week. For more info on the possibilities of `PDBList`, see the API documentation.

15.9 General questions

15.9.1 How well tested is Bio.PDB?

Pretty well, actually. Bio.PDB has been extensively tested on nearly 5500 structures from the PDB - all structures seemed to be parsed correctly. More details can be found in the Bio.PDB Bioinformatics article. Bio.PDB has been used/is being used in many research projects as a reliable tool. In fact, I'm using Bio.PDB almost daily for research purposes and continue working on improving it and adding new features.

15.9.2 How fast is it?

The `PDBParser` performance was tested on about 800 structures (each belonging to a unique SCOP superfamily). This takes about 20 minutes, or on average 1.5 seconds per structure. Parsing the structure of the large ribosomal subunit (1FKK), which contains about 64000 atoms, takes 10 seconds on a 1000 MHz PC. In short: it's more than fast enough for many applications.

15.9.3 Is there support for molecular graphics?

Not directly, mostly since there are quite a few Python based/Python aware solutions already, that can potentially be used with Bio.PDB. My choice is Pymol, BTW (I've used this successfully with Bio.PDB, and there will probably be specific PyMol modules in Bio.PDB soon/some day). Python based/aware molecular graphics solutions include:

- PyMol: https://pymol.org/
- Chimera: https://www.cgl.ucsf.edu/chimera/
- PMV: http://www.scripps.edu/~sanner/python/
- Coot: https://www2.mrc-lmb.cam.ac.uk/personal/pemsley/coot/
- CCP4mg: http://www.ccp4.ac.uk/MG/
- mmLib: http://pymmlib.sourceforge.net/
- VMD: https://www.ks.uiuc.edu/Research/vmd/
- MMTK: http://dirac.cnrs-orleans.fr/MMTK/

15.9.4 Who's using Bio.PDB?

Bio.PDB was used in the construction of DISEMBL, a web server that predicts disordered regions in proteins (http://dis.embl.de/). Bio.PDB has also been used to perform a large scale search for active sites similarities between protein structures in the PDB [Hamelryck2003B], and to develop a new algorithm that identifies linear secondary structure elements [Majumdar2005].

Judging from requests for features and information, Bio.PDB is also used by several LPCs (Large Pharmaceutical Companies :-).

CHAPTER

SIXTEEN

BIO.POPGEN: POPULATION GENETICS

`Bio.PopGen` is a Biopython module supporting population genetics, available in Biopython 1.44 onwards. The objective for the module is to support widely used data formats, applications and databases.

16.1 GenePop

GenePop (http://genepop.curtin.edu.au/) is a popular population genetics software package supporting Hardy-Weinberg tests, linkage disequilibrium, population differentiation, basic statistics, F_{st} and migration estimates, among others. GenePop does not supply sequence based statistics as it doesn't handle sequence data. The GenePop file format is supported by a wide range of other population genetic software applications, thus making it a relevant format in the population genetics field.

`Bio.PopGen` provides a parser and generator of GenePop file format. Utilities to manipulate the content of a record are also provided. Here is an example on how to read a GenePop file (you can find example GenePop data files in the Test/PopGen directory of Biopython):

```
from Bio.PopGen import GenePop

with open("example.gen") as handle:
    rec = GenePop.read(handle)
```

This will read a file called example.gen and parse it. If you do print rec, the record will be output again, in GenePop format.

The most important information in rec will be the loci names and population information (but there is more – use help(GenePop.Record) to check the API documentation). Loci names can be found on rec.loci_list. Population information can be found on rec.populations. Populations is a list with one element per population. Each element is itself a list of individuals, each individual is a pair composed by individual name and a list of alleles (2 per marker), here is an example for rec.populations:

```
[
    [
        ("Ind1", [(1, 2), (3, 3), (200, 201)]),
        ("Ind2", [(2, None), (3, 3), (None, None)]),
    ],
    [
        ("Other1", [(1, 1), (4, 3), (200, 200)]),
    ],
]
```

So we have two populations, the first with two individuals, the second with only one. The first individual of the first population is called Ind1, allelic information for each of the 3 loci follows. Please note that for any locus, information might be missing (see as an example, Ind2 above).

A few utility functions to manipulate GenePop records are made available, here is an example:

```
from Bio.PopGen import GenePop

# Imagine that you have loaded rec, as per the code snippet above...

rec.remove_population(pos)
# Removes a population from a record, pos is the population position in
# rec.populations, remember that it starts on position 0.
# rec is altered.

rec.remove_locus_by_position(pos)
#  Removes a locus by its position, pos is the locus position in
#   rec.loci_list, remember that it starts on position 0.
#   rec is altered.

rec.remove_locus_by_name(name)
# Removes a locus by its name, name is the locus name as in
# rec.loci_list. If the name doesn't exist the function fails
# silently.
# rec is altered.

rec_loci = rec.split_in_loci()
# Splits a record in loci, that is, for each loci, it creates a new
# record, with a single loci and all populations.
# The result is returned in a dictionary, being each key the locus name.
# The value is the GenePop record.
# rec is not altered.

rec_pops = rec.split_in_pops(pop_names)
# Splits a record in populations, that is, for each population, it creates
# a new record, with a single population and all loci.
# The result is returned in a dictionary, being each key
# the population name. As population names are not available in GenePop,
# they are passed in array (pop_names).
# The value of each dictionary entry is the GenePop record.
# rec is not altered.
```

GenePop does not support population names, a limitation which can be cumbersome at times. Functionality to enable population names is currently being planned for Biopython. These extensions won't break compatibility in any way with the standard format. In the medium term, we would also like to support the GenePop web service.

CHAPTER

SEVENTEEN

PHYLOGENETICS WITH BIO.PHYLO

The Bio.Phylo module was introduced in Biopython 1.54. Following the lead of SeqIO and AlignIO, it aims to provide a common way to work with phylogenetic trees independently of the source data format, as well as a consistent API for I/O operations.

Bio.Phylo is described in an open-access journal article Talevich *et al.* 2012 [Talevich2012], which you might also find helpful.

17.1 Demo: What's in a Tree?

To get acquainted with the module, let's start with a tree that we've already constructed, and inspect it a few different ways. Then we'll colorize the branches, to use a special phyloXML feature, and finally save it.

Create a simple Newick file named `simple.dnd` using your favorite text editor, or use simple.dnd provided with the Biopython source code:

```
(((A,B),(C,D)),(E,F,G));
```

This tree has no branch lengths, only a topology and labeled terminals. (If you have a real tree file available, you can follow this demo using that instead.)

Launch the Python interpreter of your choice:

```
$ ipython -pylab
```

For interactive work, launching the IPython interpreter with the `-pylab` flag enables **matplotlib** integration, so graphics will pop up automatically. We'll use that during this demo.

Now, within Python, read the tree file, giving the file name and the name of the format.

```
>>> from Bio import Phylo
>>> tree = Phylo.read("simple.dnd", "newick")
```

Printing the tree object as a string gives us a look at the entire object hierarchy.

```
>>> print(tree)
Tree(rooted=False, weight=1.0)
    Clade()
        Clade()
            Clade()
                Clade(name='A')
                Clade(name='B')
```

(continues on next page)

(continued from previous page)

```
            Clade()
                Clade(name='C')
                Clade(name='D')
        Clade()
            Clade(name='E')
            Clade(name='F')
            Clade(name='G')
```

The `Tree` object contains global information about the tree, such as whether it's rooted or unrooted. It has one root clade, and under that, it's nested lists of clades all the way down to the tips.

The function `draw_ascii` creates a simple ASCII-art (plain text) dendrogram. This is a convenient visualization for interactive exploration, in case better graphical tools aren't available.

```
>>> from Bio import Phylo
>>> tree = Phylo.read("simple.dnd", "newick")
>>> Phylo.draw_ascii(tree)
                                                 ____________________________ A
                             ____________________|
                            |                    |____________________________ B
  __________________________|
 |                          |                     ____________________________ C
 |                          |____________________|
_|                                               |____________________________ D
 |
 |                           ____________________ E
 |                          |
 |__________________________|____________________ F
                            |
                            |____________________ G
```

If you have **matplotlib** or **pylab** installed, you can create a graphical tree using the `draw` function.

```
>>> tree.rooted = True
```

```
>>> Phylo.draw(tree)
```

See Fig. 1.

17.1.1 Coloring branches within a tree

The function `draw` supports the display of different colors and branch widths in a tree. As of Biopython 1.59, the `color` and `width` attributes are available on the basic Clade object and there's nothing extra required to use them. Both attributes refer to the branch leading the given clade, and apply recursively, so all descendent branches will also inherit the assigned width and color values during display.

In earlier versions of Biopython, these were special features of PhyloXML trees, and using the attributes required first converting the tree to a subclass of the basic tree object called Phylogeny, from the Bio.Phylo.PhyloXML module.

In Biopython 1.55 and later, this is a convenient tree method:

```
>>> tree = tree.as_phyloxml()
```

In Biopython 1.54, you can accomplish the same thing with one extra import:

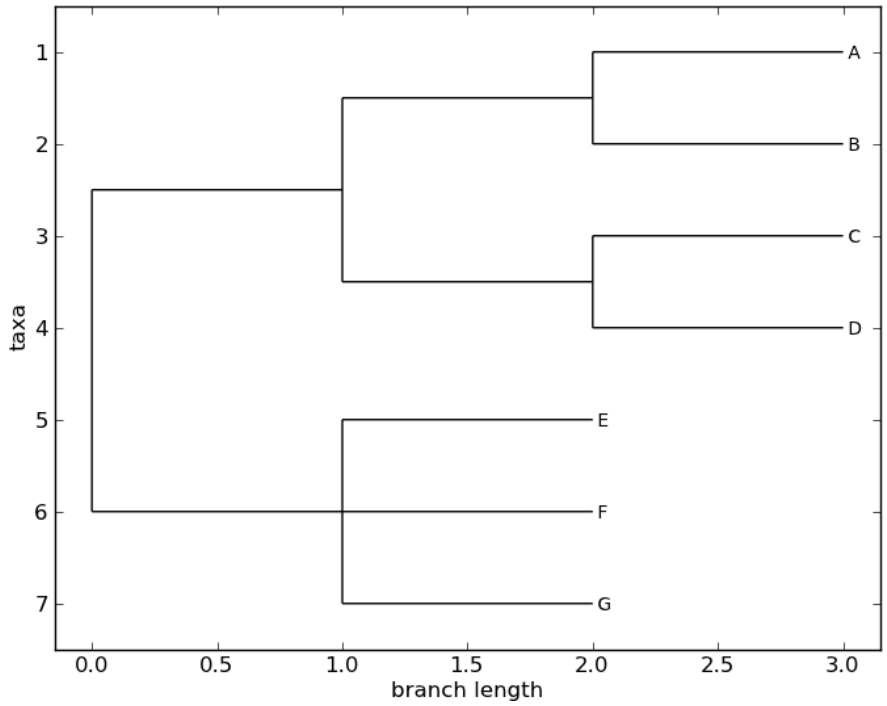


Fig. 1: A rooted tree drawn with `Phylo.draw`.

```
>>> from Bio.Phylo.PhyloXML import Phylogeny
>>> tree = Phylogeny.from_tree(tree)
```

Note that the file formats Newick and Nexus don't support branch colors or widths, so if you use these attributes in Bio.Phylo, you will only be able to save the values in PhyloXML format. (You can still save a tree as Newick or Nexus, but the color and width values will be skipped in the output file.)

Now we can begin assigning colors. First, we'll color the root clade gray. We can do that by assigning the 24-bit color value as an RGB triple, an HTML-style hex string, or the name of one of the predefined colors.

```
>>> tree.root.color = (128, 128, 128)
```

Or:

```
>>> tree.root.color = "#808080"
```

Or:

```
>>> tree.root.color = "gray"
```

Colors for a clade are treated as cascading down through the entire clade, so when we colorize the root here, it turns the whole tree gray. We can override that by assigning a different color lower down on the tree.

Let's target the most recent common ancestor (MRCA) of the nodes named "E" and "F". The `common_ancestor` method returns a reference to that clade in the original tree, so when we color that clade "salmon", the color will show up in the original tree.

```
>>> mrca = tree.common_ancestor({"name": "E"}, {"name": "F"})
>>> mrca.color = "salmon"
```

If we happened to know exactly where a certain clade is in the tree, in terms of nested list entries, we can jump directly to that position in the tree by indexing it. Here, the index `[0,1]` refers to the second child of the first child of the root.

```
>>> tree.clade[0, 1].color = "blue"
```

Finally, show our work:

```
>>> Phylo.draw(tree)
```

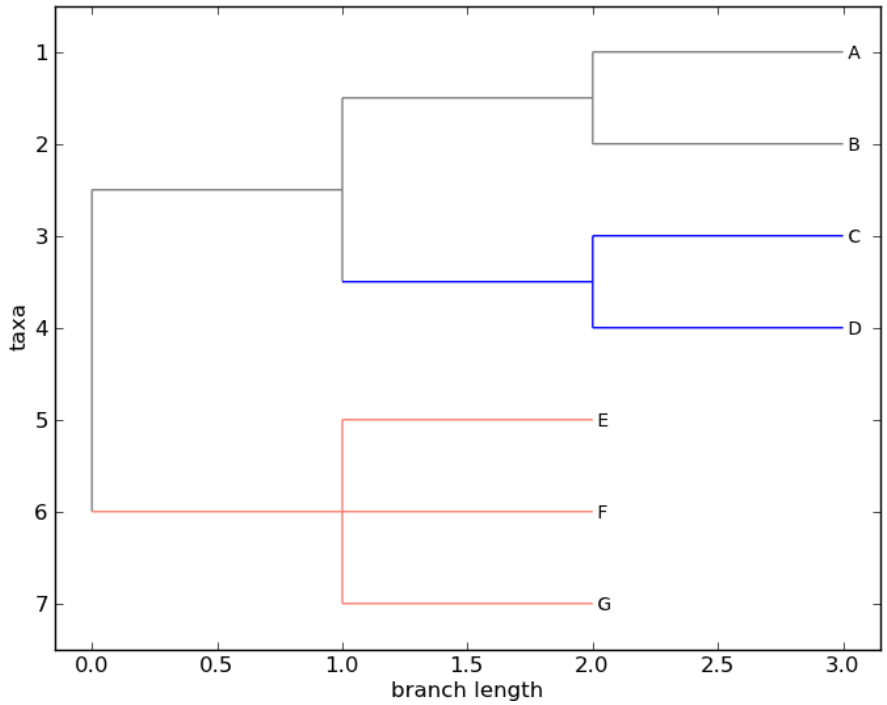


Fig. 2: A colorized tree drawn with `Phylo.draw`.

See Fig. 2.

Note that a clade's color includes the branch leading to that clade, as well as its descendents. The common ancestor of E and F turns out to be just under the root, and with this coloring we can see exactly where the root of the tree is.

My, we've accomplished a lot! Let's take a break here and save our work. Call the `write` function with a file name or handle — here we use standard output, to see what would be written — and the format `phyloxml`. PhyloXML saves the colors we assigned, so you can open this phyloXML file in another tree viewer like Archaeopteryx, and the colors will show up there, too.

```
>>> import sys
>>> n = Phylo.write(tree, sys.stdout, "phyloxml")
<phyloxml ...>
  <phylogeny rooted="true">
    <clade>
      <color>
        <red>128</red>
        <green>128</green>
        <blue>128</blue>
      </color>
      <clade>
```

(continues on next page)

(continued from previous page)

```
        <clade>
          <clade>
            <name>A</name>
          </clade>
          <clade>
            <name>B</name>
          </clade>
        </clade>
        <clade>
          <color>
            <red>0</red>
            <green>0</green>
            <blue>255</blue>
          </color>
          <clade>
            <name>C</name>
          </clade>
          ...
    </clade>
  </phylogeny>
</phyloxml>
>>> n
1
```

The rest of this chapter covers the core functionality of Bio.Phylo in greater detail. For more examples of using Bio.Phylo, see the cookbook page on Biopython.org:

http://biopython.org/wiki/Phylo_cookbook

17.2 I/O functions

Like SeqIO and AlignIO, Phylo handles file input and output through four functions: `parse`, `read`, `write` and `convert`, all of which support the tree file formats Newick, NEXUS, phyloXML and NeXML, as well as the Comparative Data Analysis Ontology (CDAO).

The `read` function parses a single tree in the given file and returns it. Careful; it will raise an error if the file contains more than one tree, or no trees.

```
>>> from Bio import Phylo
>>> tree = Phylo.read("Tests/Nexus/int_node_labels.nwk", "newick")
>>> print(tree)
Tree(rooted=False, weight=1.0)
    Clade(branch_length=75.0, name='gymnosperm')
        Clade(branch_length=25.0, name='Coniferales')
            Clade(branch_length=25.0)
                Clade(branch_length=10.0, name='Tax+nonSci')
                    Clade(branch_length=90.0, name='Taxaceae')
                        Clade(branch_length=125.0, name='Cephalotaxus')
                        ...
```

(Example files are available in the `Tests/Nexus/` and `Tests/PhyloXML/` directories of the Biopython distribution.)

To handle multiple (or an unknown number of) trees, use the `parse` function iterates through each of the trees in the given file:

```
>>> trees = Phylo.parse("Tests/PhyloXML/phyloxml_examples.xml", "phyloxml")
>>> for tree in trees:
...     print(tree)
...
Phylogeny(description='phyloXML allows to use either a "branch_length" attribute...',␣
↪name='example from Prof. Joe Felsenstein's book "Inferring Phyl...', rooted=True)
    Clade()
        Clade(branch_length=0.06)
            Clade(branch_length=0.102, name='A')
            ...
```

Write a tree or iterable of trees back to file with the `write` function:

```
>>> trees = Phylo.parse("Tests/PhyloXML/phyloxml_examples.xml", "phyloxml")
>>> tree1 = next(trees)
>>> Phylo.write(tree1, "tree1.nwk", "newick")
1
>>> Phylo.write(trees, "other_trees.xml", "phyloxml")  # write the remaining trees
12
```

Convert files between any of the supported formats with the `convert` function:

```
>>> Phylo.convert("tree1.nwk", "newick", "tree1.xml", "nexml")
1
>>> Phylo.convert("other_trees.xml", "phyloxml", "other_trees.nex", "nexus")
12
```

To use strings as input or output instead of actual files, use `StringIO` as you would with SeqIO and AlignIO:

```
>>> from Bio import Phylo
>>> from io import StringIO
>>> handle = StringIO("(((A,B),(C,D)),(E,F,G));")
>>> tree = Phylo.read(handle, "newick")
```

17.3 View and export trees

The simplest way to get an overview of a `Tree` object is to `print` it:

```
>>> from Bio import Phylo
>>> tree = Phylo.read("PhyloXML/example.xml", "phyloxml")
>>> print(tree)
Phylogeny(description='phyloXML allows to use either a "branch_length" attribute...',␣
↪name='example from Prof. Joe Felsenstein's book "Inferring Phyl...', rooted=True)
    Clade()
        Clade(branch_length=0.06)
            Clade(branch_length=0.102, name='A')
            Clade(branch_length=0.23, name='B')
        Clade(branch_length=0.4, name='C')
```

This is essentially an outline of the object hierarchy Biopython uses to represent a tree. But more likely, you'd want to see a drawing of the tree. There are three functions to do this.

As we saw in the demo, `draw_ascii` prints an ascii-art drawing of the tree (a rooted phylogram) to standard output, or an open file handle if given. Not all of the available information about the tree is shown, but it provides a way to quickly view the tree without relying on any external dependencies.

```
>>> tree = Phylo.read("PhyloXML/example.xml", "phyloxml")
>>> Phylo.draw_ascii(tree)
              ____________________ A
  ___________|
_|           |_____________________________________________ B
 |
 |_____________________________________________________________________________ C
```

The `draw` function draws a more attractive image using the matplotlib library. See the API documentation for details on the arguments it accepts to customize the output.

```
>>> Phylo.draw(tree, branch_labels=lambda c: c.branch_length)
```

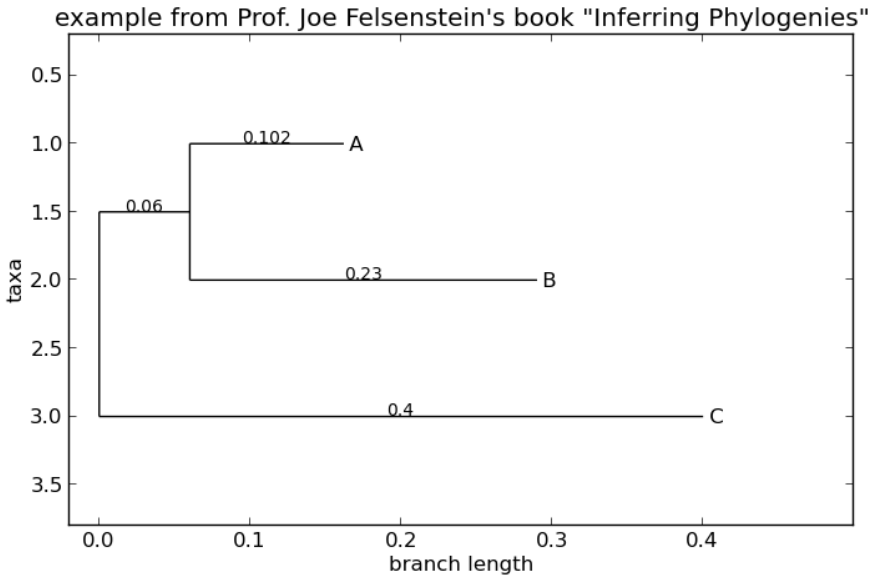


Fig. 3: A simple rooted tree plotted with the `draw` function.

See Fig. 3 for example.

See the Phylo page on the Biopython wiki (http://biopython.org/wiki/Phylo) for descriptions and examples of the more advanced functionality in `draw_ascii`, `draw_graphviz` and `to_networkx`.

17.4 Using Tree and Clade objects

The `Tree` objects produced by `parse` and `read` are containers for recursive sub-trees, attached to the `Tree` object at the `root` attribute (whether or not the phylogenetic tree is actually considered rooted). A `Tree` has globally applied information for the phylogeny, such as rootedness, and a reference to a single `Clade`; a `Clade` has node- and clade-specific information, such as branch length, and a list of its own descendent `Clade` instances, attached at the `clades` attribute.

So there is a distinction between `tree` and `tree.root`. In practice, though, you rarely need to worry about it. To smooth over the difference, both `Tree` and `Clade` inherit from `TreeMixin`, which contains the implementations for methods that would be commonly used to search, inspect or modify a tree or any of its clades. This means that almost all of the methods supported by `tree` are also available on `tree.root` and any clade below it. (`Clade` also has a `root` property, which returns the clade object itself.)

17.4.1 Search and traversal methods

For convenience, we provide a couple of simplified methods that return all external or internal nodes directly as a list:

`get_terminals`
: makes a list of all of this tree's terminal (leaf) nodes.

`get_nonterminals`
: makes a list of all of this tree's nonterminal (internal) nodes.

These both wrap a method with full control over tree traversal, `find_clades`. Two more traversal methods, `find_elements` and `find_any`, rely on the same core functionality and accept the same arguments, which we'll call a "target specification" for lack of a better description. These specify which objects in the tree will be matched and returned during iteration. The first argument can be any of the following types:

- A **TreeElement instance**, which tree elements will match by identity — so searching with a Clade instance as the target will find that clade in the tree;
- A **string**, which matches tree elements' string representation — in particular, a clade's `name` *(added in Biopython 1.56)*;
- A **class** or **type**, where every tree element of the same type (or sub-type) will be matched;
- A **dictionary** where keys are tree element attributes and values are matched to the corresponding attribute of each tree element. This one gets even more elaborate:
  - If an `int` is given, it matches numerically equal attributes, e.g. 1 will match 1 or 1.0
  - If a boolean is given (True or False), the corresponding attribute value is evaluated as a boolean and checked for the same
  - `None` matches `None`
  - If a string is given, the value is treated as a regular expression (which must match the whole string in the corresponding element attribute, not just a prefix). A given string without special regex characters will match string attributes exactly, so if you don't use regexes, don't worry about it. For example, in a tree with clade names Foo1, Foo2 and Foo3, `tree.find_clades({"name": "Foo1"})` matches Foo1, `{"name": "Foo.*"}` matches all three clades, and `{"name": "Foo"}` doesn't match anything.

  Since floating-point arithmetic can produce some strange behavior, we don't support matching `floats` directly. Instead, use the boolean `True` to match every element with a nonzero value in the specified attribute, then filter on that attribute manually with an inequality (or exact number, if you like living dangerously).

  If the dictionary contains multiple entries, a matching element must match each of the given attribute values — think "and", not "or".

- A **function** taking a single argument (it will be applied to each element in the tree), returning True or False. For convenience, LookupError, AttributeError and ValueError are silenced, so this provides another safe way to search for floating-point values in the tree, or some more complex characteristic.

After the target, there are two optional keyword arguments:

terminal
— A boolean value to select for or against terminal clades (a.k.a. leaf nodes): True searches for only terminal clades, False for non-terminal (internal) clades, and the default, None, searches both terminal and non-terminal clades, as well as any tree elements lacking the `is_terminal` method.

order
— Tree traversal order: `"preorder"` (default) is depth-first search, `"postorder"` is DFS with child nodes preceding parents, and `"level"` is breadth-first search.

Finally, the methods accept arbitrary keyword arguments which are treated the same way as a dictionary target specification: keys indicate the name of the element attribute to search for, and the argument value (string, integer, None or boolean) is compared to the value of each attribute found. If no keyword arguments are given, then any TreeElement types are matched. The code for this is generally shorter than passing a dictionary as the target specification: `tree.find_clades({"name": "Foo1"})` can be shortened to `tree.find_clades(name="Foo1")`.

(In Biopython 1.56 or later, this can be even shorter: `tree.find_clades("Foo1")`)

Now that we've mastered target specifications, here are the methods used to traverse a tree:

`find_clades`
Find each clade containing a matching element. That is, find each element as with `find_elements`, but return the corresponding clade object. (This is usually what you want.)

The result is an iterable through all matching objects, searching depth-first by default. This is not necessarily the same order as the elements appear in the Newick, Nexus or XML source file!

`find_elements`
Find all tree elements matching the given attributes, and return the matching elements themselves. Simple Newick trees don't have complex sub-elements, so this behaves the same as `find_clades` on them. PhyloXML trees often do have complex objects attached to clades, so this method is useful for extracting those.

`find_any`
Return the first element found by `find_elements()`, or None. This is also useful for checking whether any matching element exists in the tree, and can be used in a conditional.

Two more methods help navigating between nodes in the tree:

`get_path`
List the clades directly between the tree root (or current clade) and the given target. Returns a list of all clade objects along this path, ending with the given target, but excluding the root clade.

`trace`
List of all clade object between two targets in this tree. Excluding start, including finish.

17.4.2 Information methods

These methods provide information about the whole tree (or any clade).

common_ancestor
: Find the most recent common ancestor of all the given targets. (This will be a Clade object). If no target is given, returns the root of the current clade (the one this method is called from); if 1 target is given, this returns the target itself. However, if any of the specified targets are not found in the current tree (or clade), an exception is raised.

count_terminals
: Counts the number of terminal (leaf) nodes within the tree.

depths
: Create a mapping of tree clades to depths. The result is a dictionary where the keys are all of the Clade instances in the tree, and the values are the distance from the root to each clade (including terminals). By default the distance is the cumulative branch length leading to the clade, but with the `unit_branch_lengths=True` option, only the number of branches (levels in the tree) is counted.

distance
: Calculate the sum of the branch lengths between two targets. If only one target is specified, the other is the root of this tree.

total_branch_length
: Calculate the sum of all the branch lengths in this tree. This is usually just called the "length" of the tree in phylogenetics, but we use a more explicit name to avoid confusion with Python terminology.

The rest of these methods are boolean checks:

is_bifurcating
: True if the tree is strictly bifurcating; i.e. all nodes have either 2 or 0 children (internal or external, respectively). The root may have 3 descendents and still be considered part of a bifurcating tree.

is_monophyletic
: Test if all of the given targets comprise a complete subclade — i.e., there exists a clade such that its terminals are the same set as the given targets. The targets should be terminals of the tree. For convenience, this method returns the common ancestor (MCRA) of the targets if they are monophyletic (instead of the value `True`), and `False` otherwise.

is_parent_of
: True if target is a descendent of this tree — not required to be a direct descendent. To check direct descendents of a clade, simply use list membership testing: `if subclade in clade: ...`

is_preterminal
: True if all direct descendents are terminal; False if any direct descendent is not terminal.

17.4.3 Modification methods

These methods modify the tree in-place. If you want to keep the original tree intact, make a complete copy of the tree first, using Python's `copy` module:

```
tree = Phylo.read("example.xml", "phyloxml")
import copy

newtree = copy.deepcopy(tree)
```

collapse
: Deletes the target from the tree, relinking its children to its parent.

`collapse_all`
: Collapse all the descendents of this tree, leaving only terminals. Branch lengths are preserved, i.e. the distance to each terminal stays the same. With a target specification (see above), collapses only the internal nodes matching the specification.

`ladderize`
: Sort clades in-place according to the number of terminal nodes. Deepest clades are placed last by default. Use `reverse=True` to sort clades deepest-to-shallowest.

`prune`
: Prunes a terminal clade from the tree. If taxon is from a bifurcation, the connecting node will be collapsed and its branch length added to remaining terminal node. This might no longer be a meaningful value.

`root_with_outgroup`
: Reroot this tree with the outgroup clade containing the given targets, i.e. the common ancestor of the outgroup. This method is only available on Tree objects, not Clades.

  If the outgroup is identical to self.root, no change occurs. If the outgroup clade is terminal (e.g. a single terminal node is given as the outgroup), a new bifurcating root clade is created with a 0-length branch to the given outgroup. Otherwise, the internal node at the base of the outgroup becomes a trifurcating root for the whole tree. If the original root was bifurcating, it is dropped from the tree.

  In all cases, the total branch length of the tree stays the same.

`root_at_midpoint`
: Reroot this tree at the calculated midpoint between the two most distant tips of the tree. (This uses `root_with_outgroup` under the hood.)

`split`
: Generate *n* (default 2) new descendants. In a species tree, this is a speciation event. New clades have the given `branch_length` and the same name as this clade's root plus an integer suffix (counting from 0) — for example, splitting a clade named "A" produces the sub-clades "A0" and "A1".

See the Phylo page on the Biopython wiki (http://biopython.org/wiki/Phylo) for more examples of using the available methods.

17.4.4 Features of PhyloXML trees

The phyloXML file format includes fields for annotating trees with additional data types and visual cues.

See the PhyloXML page on the Biopython wiki (http://biopython.org/wiki/PhyloXML) for descriptions and examples of using the additional annotation features provided by PhyloXML.

17.5 Running external applications

While Bio.Phylo doesn't infer trees from alignments itself, there are third-party programs available that do. These can be accessed from within python by using the `subprocess` module.

Below is an example on how to use a python script to interact with PhyML (http://www.atgc-montpellier.fr/phyml/). The program accepts an input alignment in `phylip-relaxed` format (that's Phylip format, but without the 10-character limit on taxon names) and a variety of options.

```
>>> import subprocess
>>> cmd = "phyml -i Tests/Phylip/random.phy"
>>> results = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, text=True)
```

The 'stdout = subprocess.PIPE' argument makes the output of the program accessible through 'results.stdout' for debugging purposes, (the same can be done for 'stderr'), and 'text=True' makes the returned information be a python string, instead of a 'bytes' object.

This generates a tree file and a stats file with the names [*input filename*]_phyml_tree.txt and [*input filename*]_phyml_stats.txt. The tree file is in Newick format:

```
>>> from Bio import Phylo
>>> tree = Phylo.read("Tests/Phylip/random.phy_phyml_tree.txt", "newick")
>>> Phylo.draw_ascii(tree)
  ____________________ F
 |
 | I
 |
_|                      _________ C
 |             _________|
 |            |         |         , J
 |            |         |_________|
 |            |                   |           , H
 |____________|                   |___________|
              |                               |_________________ D
              |
              , G
              |
              |                  , E
              |__________________|
                                 |                                  _____________________________ A
                                 |__________________|
                                                    |__________ B
```

The `subprocess` module can also be used for interacting with any other programs that provide a command line interface such as RAxML (https://sco.h-its.org/exelixis/software.html), FastTree (http://www.microbesonline.org/fasttree/), `dnaml` and `protml`.

17.6 PAML integration

Biopython 1.58 brought support for PAML (http://abacus.gene.ucl.ac.uk/software/paml.html), a suite of programs for phylogenetic analysis by maximum likelihood. Currently the programs codeml, baseml and yn00 are implemented. Due to PAML's usage of control files rather than command line arguments to control runtime options, usage of this wrapper strays from the format of other application wrappers in Biopython.

A typical workflow would be to initialize a PAML object, specifying an alignment file, a tree file, an output file and a working directory. Next, runtime options are set via the `set_options()` method or by reading an existing control file. Finally, the program is run via the `run()` method and the output file is automatically parsed to a results dictionary.

Here is an example of typical usage of codeml:

```
>>> from Bio.Phylo.PAML import codeml
>>> cml = codeml.Codeml()
>>> cml.alignment = "Tests/PAML/Alignments/alignment.phylip"
>>> cml.tree = "Tests/PAML/Trees/species.tree"
>>> cml.out_file = "results.out"
>>> cml.working_dir = "./scratch"
```

(continues on next page)

(continued from previous page)

```
>>> cml.set_options(
...     seqtype=1,
...     verbose=0,
...     noisy=0,
...     RateAncestor=0,
...     model=0,
...     NSsites=[0, 1, 2],
...     CodonFreq=2,
...     cleandata=1,
...     fix_alpha=1,
...     kappa=4.54006,
... )
>>> results = cml.run()
>>> ns_sites = results.get("NSsites")
>>> m0 = ns_sites.get(0)
>>> m0_params = m0.get("parameters")
>>> print(m0_params.get("omega"))
```

Existing output files may be parsed as well using a module's `read()` function:

```
>>> results = codeml.read("Tests/PAML/Results/codeml/codeml_NSsites_all.out")
>>> print(results.get("lnL max"))
```

Detailed documentation for this new module currently lives on the Biopython wiki: http://biopython.org/wiki/PAML

17.7 Future plans

Bio.Phylo is under active development. Here are some features we might add in future releases:

New methods
: Generally useful functions for operating on Tree or Clade objects appear on the Biopython wiki first, so that casual users can test them and decide if they're useful before we add them to Bio.Phylo:

  http://biopython.org/wiki/Phylo_cookbook

Bio.Nexus port
: Much of this module was written during Google Summer of Code 2009, under the auspices of NESCent, as a project to implement Python support for the phyloXML data format (see *Features of PhyloXML trees*). Support for Newick and Nexus formats was added by porting part of the existing Bio.Nexus module to the new classes used by Bio.Phylo.

  Currently, Bio.Nexus contains some useful features that have not yet been ported to Bio.Phylo classes — notably, calculating a consensus tree. If you find some functionality lacking in Bio.Phylo, try poking through Bio.Nexus to see if it's there instead.

We're open to any suggestions for improving the functionality and usability of this module; just let us know on the mailing list or our bug database.

Finally, if you need additional functionality not yet included in the Phylo module, check if it's available in another of the high-quality Python libraries for phylogenetics such as DendroPy (https://dendropy.org/) or PyCogent (http://pycogent.org/). Since these libraries also support standard file formats for phylogenetic trees, you can easily transfer data between libraries by writing to a temporary file or StringIO object.

CHAPTER

EIGHTEEN

SEQUENCE MOTIF ANALYSIS USING BIO.MOTIFS

This chapter gives an overview of the functionality of the `Bio.motifs` package included in Biopython. It is intended for people who are involved in the analysis of sequence motifs, so I'll assume that you are familiar with basic notions of motif analysis. In case something is unclear, please look at Section *Useful links* for some relevant links.

Most of this chapter describes the new `Bio.motifs` package included in Biopython 1.61 onwards, which is replacing the older `Bio.Motif` package introduced with Biopython 1.50, which was in turn based on two older former Biopython modules, `Bio.AlignAce` and `Bio.MEME`. It provides most of their functionality with a unified motif object implementation.

Speaking of other libraries, if you are reading this you might be interested in TAMO, another python library designed to deal with sequence motifs. It supports more *de-novo* motif finders, but it is not a part of Biopython and has some restrictions on commercial use.

18.1 Motif objects

Since we are interested in motif analysis, we need to take a look at `Motif` objects in the first place. For that we need to import the Bio.motifs library:

```
>>> from Bio import motifs
```

and we can start creating our first motif objects. We can either create a `Motif` object from a list of instances of the motif, or we can obtain a `Motif` object by parsing a file from a motif database or motif finding software.

18.1.1 Creating a motif from instances

Suppose we have these instances of a DNA motif:

```
>>> from Bio.Seq import Seq
>>> instances = [
...     Seq("TACAA"),
...     Seq("TACGC"),
...     Seq("TACAC"),
...     Seq("TACCC"),
...     Seq("AACCC"),
...     Seq("AATGC"),
...     Seq("AATGC"),
... ]
```

then we can create a Motif object as follows:

```
>>> m = motifs.create(instances)
```

The instances from which this motif was created is stored in the `.alignment` property:

```
>>> print(m.alignment.sequences)
[Seq('TACAA'), Seq('TACGC'), Seq('TACAC'), Seq('TACCC'), Seq('AACCC'), Seq('AATGC'), Seq(
↪'AATGC')]
```

Printing the Motif object shows the instances from which it was constructed:

```
>>> print(m)
TACAA
TACGC
TACAC
TACCC
AACCC
AATGC
AATGC
```

The length of the motif is defined as the sequence length, which should be the same for all instances:

```
>>> len(m)
5
```

The Motif object has an attribute `.counts` containing the counts of each nucleotide at each position. Printing this counts matrix shows it in an easily readable format:

```
>>> print(m.counts)
        0      1      2      3      4
A:   3.00   7.00   0.00   2.00   1.00
C:   0.00   0.00   5.00   2.00   6.00
G:   0.00   0.00   0.00   3.00   0.00
T:   4.00   0.00   2.00   0.00   0.00
```

You can access these counts as a dictionary:

```
>>> m.counts["A"]
[3.0, 7.0, 0.0, 2.0, 1.0]
```

but you can also think of it as a 2D array with the nucleotide as the first dimension and the position as the second dimension:

```
>>> m.counts["T", 0]
4.0
>>> m.counts["T", 2]
2.0
>>> m.counts["T", 3]
0.0
```

You can also directly access columns of the counts matrix

```
>>> m.counts[:, 3]
{'A': 2.0, 'C': 2.0, 'T': 0.0, 'G': 3.0}
```

Instead of the nucleotide itself, you can also use the index of the nucleotide in the alphabet of the motif:

```
>>> m.alphabet
'ACGT'
>>> m.counts["A", :]
(3.0, 7.0, 0.0, 2.0, 1.0)
>>> m.counts[0, :]
(3.0, 7.0, 0.0, 2.0, 1.0)
```

18.1.2 Obtaining a consensus sequence

The consensus sequence of a motif is defined as the sequence of letters along the positions of the motif for which the largest value in the corresponding columns of the `.counts` matrix is obtained:

```
>>> m.consensus
Seq('TACGC')
```

Conversely, the anticonsensus sequence corresponds to the smallest values in the columns of the `.counts` matrix:

```
>>> m.anticonsensus
Seq('CCATG')
```

Note that there is some ambiguity in the definition of the consensus and anticonsensus sequence if in some columns multiple nucleotides have the maximum or minimum count.

For DNA sequences, you can also ask for a degenerate consensus sequence, in which ambiguous nucleotides are used for positions where there are multiple nucleotides with high counts:

```
>>> m.degenerate_consensus
Seq('WACVC')
```

Here, W and R follow the IUPAC nucleotide ambiguity codes: W is either A or T, and V is A, C, or G [Cornish1985]. The degenerate consensus sequence is constructed following the rules specified by Cavener [Cavener1987].

The `motif.counts.calculate_consensus` method lets you specify in detail how the consensus sequence should be calculated. This method largely follows the conventions of the EMBOSS program `cons`, and takes the following arguments:

substitution_matrix
: The scoring matrix used when comparing sequences. By default, it is `None`, in which case we simply count the frequency of each letter. Instead of the default value, you can use the substitution matrices available in `Bio.Align.substitution\_matrices`. Common choices are BLOSUM62 (also known as EBLOSUM62) for protein, and NUC.4.4 (also known as EDNAFULL) for nucleotides. NOTE: Currently, this method has not yet been implemented for values other than the default value `None`.

plurality
: Threshold value for the number of positive matches, divided by the total count in a column, required to reach consensus. If `substitution_matrix` is `None`, then this argument must also be `None`, and is ignored; a `ValueError` is raised otherwise. If `substitution_matrix` is not `None`, then the default value of the plurality is 0.5.

identity
: Number of identities, divided by the total count in a column, required to define a consensus value. If the number of identities is less than identity multiplied by the total count in a column, then the undefined character (`N` for nucleotides and `X` for amino acid sequences) is used in the consensus sequence. If `identity` is 1.0, then only columns of identical letters contribute to the consensus. Default value is zero.

setcase
: threshold for the positive matches, divided by the total count in a column, above which the consensus is is upper-case and below which the consensus is in lower-case. By default, this is equal to 0.5.

This is an example:

```
>>> m.counts.calculate_consensus(identity=0.5, setcase=0.7)
'tACNC'
```

18.1.3 Reverse-complementing a motif

We can get the reverse complement of a motif by calling the `reverse_complement` method on it:

```
>>> r = m.reverse_complement()
>>> r.consensus
Seq('GCGTA')
>>> r.degenerate_consensus
Seq('GBGTW')
>>> print(r)
TTGTA
GCGTA
GTGTA
GGGTA
GGGTT
GCATT
GCATT
```

The reverse complement is only defined for DNA motifs.

18.1.4 Slicing a motif

You can slice the motif to obtain a new `Motif` object for the selected positions:

```
>>> m_sub = m[2:-1]
>>> print(m_sub)
CA
CG
CA
CC
CC
TG
TG
>>> m_sub.consensus
Seq('CG')
>>> m_sub.degenerate_consensus
Seq('CV')
```

18.1.5 Relative entropy

The relative entropy (or Kullback-Leibler distance) H_j of column j of the motif is defined as in [Schneider1986] [Durbin1998]:

$$H_j = \sum_{i=1}^{M} p_{ij} \log\left(\frac{p_{ij}}{b_i}\right)$$

where:

- M – The number of letters in the alphabet (given by `len(m.alphabet)`);
- p_{ij} – The observed frequency of letter i, normalized, in the j-th column (see below);
- b_i – The background probability of letter i (given by `m.background[i]`).

The observed frequency p_{ij} is computed as follows:

$$p_{ij} = \frac{c_{ij} + k_i}{C_j + k}$$

where:

- c_{ij} – the number of times letter i appears in column j of the alignment (given by `m.counts[i, j]`);
- C_j – The total number of letters in column j: $C_j = \sum_{i=1}^{M} c_{ij}$ (given by `sum(m.counts[:, j])`).
- k_i – the pseudocount of letter i (given by `m.pseudocounts[i]`).
- k – the total pseudocount: $k = \sum_{i=1}^{M} k_i$ (given by `sum(m.pseudocounts.values())`).

With these definitions, both p_{ij} and b_i are normalized to 1:

$$\sum_{i=1}^{M} p_{ij} = 1$$

$$\sum_{i=1}^{M} b_i = 1$$

The relative entropy is the same as the information content if the background distribution is uniform.

The relative entropy for each column of motif `m` can be obtained using the `relative_entropy` property:

```
>>> m.relative_entropy
array([1.01477186, 2.        , 1.13687943, 0.44334329, 1.40832722])
```

These values are calculated using the base-2 logarithm, and are therefore in units of bits. The second column (which consists of `A` nucleotides only) has the highest relative entropy; the fourth column (which consists of `A`, `C`, or `G` nucleotides) has the lowest relative entropy). The relative entropy of the motif can be calculated by summing over the columns:

```
>>> print(f"Relative entropy is {sum(m.relative_entropy):0.5f}")
Relative entropy is 6.00332
```

18.1.6 Creating a sequence logo

If we have internet access, we can create a weblogo:

```
>>> m.weblogo("mymotif.png")
```

We should get our logo saved as a PNG in the specified file.

18.2 Reading motifs

Creating motifs from instances by hand is a bit boring, so it's useful to have some I/O functions for reading and writing motifs. There are not any really well established standards for storing motifs, but there are a couple of formats that are more used than others.

18.2.1 JASPAR

One of the most popular motif databases is JASPAR. In addition to the motif sequence information, the JASPAR database stores a lot of meta-information for each motif. The module `Bio.motifs` contains a specialized class `jaspar.Motif` in which this meta-information is represented as attributes:

- `matrix_id` - the unique JASPAR motif ID, e.g. 'MA0004.1'
- `name` - the name of the TF, e.g. 'Arnt'
- `collection` - the JASPAR collection to which the motif belongs, e.g. 'CORE'
- `tf_class` - the structural class of this TF, e.g. 'Zipper-Type'
- `tf_family` - the family to which this TF belongs, e.g. 'Helix-Loop-Helix'
- `species` - the species to which this TF belongs, may have multiple values, these are specified as taxonomy IDs, e.g. 10090
- `tax_group` - the taxonomic supergroup to which this motif belongs, e.g. 'vertebrates'
- `acc` - the accession number of the TF protein, e.g. 'P53762'
- `data_type` - the type of data used to construct this motif, e.g. 'SELEX'
- `medline` - the Pubmed ID of literature supporting this motif, may be multiple values, e.g. 7592839
- `pazar_id` - external reference to the TF in the PAZAR database, e.g. 'TF0000003'
- `comment` - free form text containing notes about the construction of the motif

The `jaspar.Motif` class inherits from the generic `Motif` class and therefore provides all the facilities of any of the motif formats — reading motifs, writing motifs, scanning sequences for motif instances etc.

JASPAR stores motifs in several different ways including three different flat file formats and as an SQL database. All of these formats facilitate the construction of a counts matrix. However, the amount of meta information described above that is available varies with the format.

The JASPAR `sites` format

The first of the three flat file formats contains a list of instances. As an example, these are the beginning and ending lines of the JASPAR `Arnt.sites` file showing known binding sites of the mouse helix-loop-helix transcription factor Arnt.

```
>MA0004 ARNT 1
CACGTGatgtcctc
>MA0004 ARNT 2
CACGTGggaggtac
>MA0004 ARNT 3
CACGTGccgcgcgc
...
>MA0004 ARNT 18
AACGTGacagccctcc
>MA0004 ARNT 19
AACGTGcacatcgtcc
>MA0004 ARNT 20
aggaatCGCGTGc
```

The parts of the sequence in capital letters are the motif instances that were found to align to each other.

We can create a `Motif` object from these instances as follows:

```
>>> from Bio import motifs
>>> with open("Arnt.sites") as handle:
...     arnt = motifs.read(handle, "sites")
...
```

The instances from which this motif was created is stored in the `.alignment` property:

```
>>> print(arnt.alignment.sequences[:3])
[Seq('CACGTG'), Seq('CACGTG'), Seq('CACGTG')]
>>> for sequence in arnt.alignment.sequences:
...     print(sequence)
...
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
AACGTG
AACGTG
AACGTG
```

(continues on next page)

(continued from previous page)

```
AACGTG
CGCGTG
```

The counts matrix of this motif is automatically calculated from the instances:

```
>>> print(arnt.counts)
       0      1      2      3      4      5
A:   4.00  19.00   0.00   0.00   0.00   0.00
C:  16.00   0.00  20.00   0.00   0.00   0.00
G:   0.00   1.00   0.00  20.00   0.00  20.00
T:   0.00   0.00   0.00   0.00  20.00   0.00
```

This format does not store any meta information.

The JASPAR `pfm` format

JASPAR also makes motifs available directly as a count matrix, without the instances from which it was created. This `pfm` format only stores the counts matrix for a single motif. For example, this is the JASPAR file `SRF.pfm` containing the counts matrix for the human SRF transcription factor:

```
 2 9 0 1 32 3 46 1 43 15 2 2
 1 33 45 45 1 1 0 0 0 1 0 1
39 2 1 0 0 0 0 0 0 0 44 43
 4 2 0 0 13 42 0 45 3 30 0 0
```

We can create a motif for this count matrix as follows:

```
>>> with open("SRF.pfm") as handle:
...     srf = motifs.read(handle, "pfm")
...
>>> print(srf.counts)
       0      1      2      3      4      5      6      7      8      9     10     11
A:   2.00   9.00   0.00   1.00  32.00   3.00  46.00   1.00  43.00  15.00   2.00   2.00
C:   1.00  33.00  45.00  45.00   1.00   1.00   0.00   0.00   0.00   1.00   0.00   1.00
G:  39.00   2.00   1.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00  44.00  43.00
T:   4.00   2.00   0.00   0.00  13.00  42.00   0.00  45.00   3.00  30.00   0.00   0.00
```

As this motif was created from the counts matrix directly, it has no instances associated with it:

```
>>> print(srf.alignment)
None
```

We can now ask for the consensus sequence of these two motifs:

```
>>> print(arnt.counts.consensus)
CACGTG
>>> print(srf.counts.consensus)
GCCCATATATGG
```

As with the instances file, no meta information is stored in this format.

The JASPAR format `jaspar`

The `jaspar` file format allows multiple motifs to be specified in a single file. In this format each of the motif records consist of a header line followed by four lines defining the counts matrix. The header line begins with a > character (similar to the Fasta file format) and is followed by the unique JASPAR matrix ID and the TF name. The following example shows a `jaspar` formatted file containing the three motifs Arnt, RUNX1 and MEF2A:

```
>MA0004.1 Arnt
A  [ 4 19  0  0  0  0 ]
C  [16  0 20  0  0  0 ]
G  [ 0  1  0 20  0 20 ]
T  [ 0  0  0  0 20  0 ]
>MA0002.1 RUNX1
A  [10 12  4  1  2  2  0  0  0  8 13 ]
C  [ 2  2  7  1  0  8  0  0  1  2  2 ]
G  [ 3  1  1  0 23  0 26 26  0  0  4 ]
T  [11 11 14 24  1 16  0  0 25 16  7 ]
>MA0052.1 MEF2A
A  [ 1  0 57  2  9  6 37  2 56  6 ]
C  [50  0  1  1  0  0  0  0  0  0 ]
G  [ 0  0  0  0  0  0  0  0  2 50 ]
T  [ 7 58  0 55 49 52 21 56  0  2 ]
```

The motifs are read as follows:

```
>>> fh = open("jaspar_motifs.txt")
>>> for m in motifs.parse(fh, "jaspar"):
...     print(m)
...
TF name  Arnt
Matrix ID   MA0004.1
Matrix:
        0      1      2      3      4      5
A:   4.00  19.00   0.00   0.00   0.00   0.00
C:  16.00   0.00  20.00   0.00   0.00   0.00
G:   0.00   1.00   0.00  20.00   0.00  20.00
T:   0.00   0.00   0.00   0.00  20.00   0.00



TF name  RUNX1
Matrix ID   MA0002.1
Matrix:
        0      1      2      3      4      5      6      7      8      9     10
A:  10.00  12.00   4.00   1.00   2.00   2.00   0.00   0.00   0.00   8.00  13.00
C:   2.00   2.00   7.00   1.00   0.00   8.00   0.00   0.00   1.00   2.00   2.00
G:   3.00   1.00   1.00   0.00  23.00   0.00  26.00  26.00   0.00   0.00   4.00
T:  11.00  11.00  14.00  24.00   1.00  16.00   0.00   0.00  25.00  16.00   7.00



TF name  MEF2A
Matrix ID   MA0052.1
Matrix:
```

(continues on next page)

(continued from previous page)

```
        0      1      2      3      4      5      6      7      8      9
A:   1.00   0.00  57.00   2.00   9.00   6.00  37.00   2.00  56.00   6.00
C:  50.00   0.00   1.00   1.00   0.00   0.00   0.00   0.00   0.00   0.00
G:   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   2.00  50.00
T:   7.00  58.00   0.00  55.00  49.00  52.00  21.00  56.00   0.00   2.00
```

Note that printing a JASPAR motif yields both the counts data and the available meta-information.

Accessing the JASPAR database

In addition to parsing these flat file formats, we can also retrieve motifs from a JASPAR SQL database. Unlike the flat file formats, a JASPAR database allows storing of all possible meta information defined in the JASPAR `Motif` class. It is beyond the scope of this document to describe how to set up a JASPAR database (please see the main JASPAR website). Motifs are read from a JASPAR database using the `Bio.motifs.jaspar.db` module. First connect to the JASPAR database using the JASPAR5 class which models the the latest JASPAR schema:

```
>>> from Bio.motifs.jaspar.db import JASPAR5
>>>
>>> JASPAR_DB_HOST = "yourhostname"  # fill in these values
>>> JASPAR_DB_NAME = "yourdatabase"
>>> JASPAR_DB_USER = "yourusername"
>>> JASPAR_DB_PASS = "yourpassword"
>>>
>>> jdb = JASPAR5(
...     host=JASPAR_DB_HOST,
...     name=JASPAR_DB_NAME,
...     user=JASPAR_DB_USER,
...     password=JASPAR_DB_PASS,
... )
```

Now we can fetch a single motif by its unique JASPAR ID with the `fetch_motif_by_id` method. Note that a JASPAR ID consists of a base ID and a version number separated by a decimal point, e.g. 'MA0004.1'. The `fetch_motif_by_id` method allows you to use either the fully specified ID or just the base ID. If only the base ID is provided, the latest version of the motif is returned.

```
>>> arnt = jdb.fetch_motif_by_id("MA0004")
```

Printing the motif reveals that the JASPAR SQL database stores much more meta-information than the flat files:

```
>>> print(arnt)
TF name Arnt
Matrix ID   MA0004.1
Collection  CORE
TF class    Zipper-Type
TF family   Helix-Loop-Helix
Species 10090
Taxonomic group vertebrates
Accession   ['P53762']
Data type used  SELEX
Medline 7592839
PAZAR ID    TF0000003
Comments    -
```

(continues on next page)

(continued from previous page)

```
Matrix:
    0      1      2      3      4      5
A:   4.00  19.00   0.00   0.00   0.00   0.00
C:  16.00   0.00  20.00   0.00   0.00   0.00
G:   0.00   1.00   0.00  20.00   0.00  20.00
T:   0.00   0.00   0.00   0.00  20.00   0.00
```

We can also fetch motifs by name. The name must be an exact match (partial matches or database wildcards are not currently supported). Note that as the name is not guaranteed to be unique, the `fetch_motifs_by_name` method actually returns a list.

```
>>> motifs = jdb.fetch_motifs_by_name("Arnt")
>>> print(motifs[0])
TF name Arnt
Matrix ID   MA0004.1
Collection  CORE
TF class    Zipper-Type
TF family   Helix-Loop-Helix
Species 10090
Taxonomic group vertebrates
Accession   ['P53762']
Data type used  SELEX
Medline 7592839
PAZAR ID    TF0000003
Comments    -
Matrix:
    0      1      2      3      4      5
A:   4.00  19.00   0.00   0.00   0.00   0.00
C:  16.00   0.00  20.00   0.00   0.00   0.00
G:   0.00   1.00   0.00  20.00   0.00  20.00
T:   0.00   0.00   0.00   0.00  20.00   0.00
```

The `fetch_motifs` method allows you to fetch motifs which match a specified set of criteria. These criteria include any of the above described meta information as well as certain matrix properties such as the minimum information content (`min_ic` in the example below), the minimum length of the matrix or the minimum number of sites used to construct the matrix. Only motifs which pass ALL the specified criteria are returned. Note that selection criteria which correspond to meta information which allow for multiple values may be specified as either a single value or a list of values, e.g. `tax_group` and `tf_family` in the example below.

```
>>> motifs = jdb.fetch_motifs(
...     collection="CORE",
...     tax_group=["vertebrates", "insects"],
...     tf_class="Winged Helix-Turn-Helix",
...     tf_family=["Forkhead", "Ets"],
...     min_ic=12,
... )
>>> for motif in motifs:
...     pass  # do something with the motif
...
```

Compatibility with Perl TFBS modules

An important thing to note is that the JASPAR `Motif` class was designed to be compatible with the popular Perl TFBS modules. Therefore some specifics about the choice of defaults for background and pseudocounts as well as how information content is computed and sequences searched for instances is based on this compatibility criteria. These choices are noted in the specific subsections below.

- **Choice of background:**
  The Perl `TFBS` modules appear to allow a choice of custom background probabilities (although the documentation states that uniform background is assumed). However the default is to use a uniform background. Therefore it is recommended that you use a uniform background for computing the position-specific scoring matrix (PSSM). This is the default when using the Biopython `motifs` module.
- **Choice of pseudocounts:**
  By default, the Perl `TFBS` modules use a pseudocount equal to $\sqrt{N} * \text{bg}[\text{nucleotide}]$, where N represents the total number of sequences used to construct the matrix. To apply this same pseudocount formula, set the motif `pseudocounts` attribute using the `jaspar.calculate\_pseudcounts()` function:

```
>>> motif.pseudocounts = motifs.jaspar.calculate_pseudocounts(motif)
```

  Note that it is possible for the counts matrix to have an unequal number of sequences making up the columns. The pseudocount computation uses the average number of sequences making up the matrix. However, when `normalize` is called on the counts matrix, each count value in a column is divided by the total number of sequences making up that specific column, not by the average number of sequences. This differs from the Perl `TFBS` modules because the normalization is not done as a separate step and so the average number of sequences is used throughout the computation of the pssm. Therefore, for matrices with unequal column counts, the PSSM computed by the `motifs` module will differ somewhat from the pssm computed by the Perl `TFBS` modules.
- **Computation of matrix information content:**
  The information content (IC) or specificity of a matrix is computed using the `mean` method of the `PositionSpecificScoringMatrix` class. However of note, in the Perl `TFBS` modules the default behavior is to compute the IC without first applying pseudocounts, even though by default the PSSMs are computed using pseudocounts as described above.
- **Searching for instances:**
  Searching for instances with the Perl `TFBS` motifs was usually performed using a relative score threshold, i.e. a score in the range 0 to 1. In order to compute the absolute PSSM score corresponding to a relative score one can use the equation:

```
>>> abs_score = (pssm.max - pssm.min) * rel_score + pssm.min
```

  To convert the absolute score of an instance back to a relative score, one can use the equation:

```
>>> rel_score = (abs_score - pssm.min) / (pssm.max - pssm.min)
```

  For example, using the Arnt motif before, let's search a sequence with a relative score threshold of 0.8.

```
>>> test_seq = Seq("TAAGCGTGCACGCGCAACACGTGCATTA")
>>> arnt.pseudocounts = motifs.jaspar.calculate_pseudocounts(arnt)
>>> pssm = arnt.pssm
>>> max_score = pssm.max
>>> min_score = pssm.min
>>> abs_score_threshold = (max_score - min_score) * 0.8 + min_score
>>> for pos, score in pssm.search(test_seq, threshold=abs_score_threshold):
...     rel_score = (score - min_score) / (max_score - min_score)
```

(continues on next page)

(continued from previous page)

```
...     print(f"Position {pos}: score = {score:5.3f}, rel. score = {rel_score:5.3f}
↪")
...
Position 2: score = 5.362, rel. score = 0.801
Position 8: score = 6.112, rel. score = 0.831
Position -20: score = 7.103, rel. score = 0.870
Position 17: score = 10.351, rel. score = 1.000
Position -11: score = 10.351, rel. score = 1.000
```

18.2.2 MEME

MEME [Bailey1994] is a tool for discovering motifs in a group of related DNA or protein sequences. It takes as input a group of DNA or protein sequences and outputs as many motifs as requested. Therefore, in contrast to JASPAR files, MEME output files typically contain multiple motifs. This is an example.

At the top of an output file generated by MEME shows some background information about the MEME and the version of MEME used:

```
********************************************************************************
MEME - Motif discovery tool
********************************************************************************
MEME version 3.0 (Release date: 2004/08/18 09:07:01)
...
```

Further down, the input set of training sequences is recapitulated:

```
********************************************************************************
TRAINING SET
********************************************************************************
DATAFILE= INO_up800.s
ALPHABET= ACGT
Sequence name            Weight Length  Sequence name            Weight Length
-------------            ------ ------  -------------            ------ ------
CHO1                     1.0000    800  CHO2                     1.0000    800
FAS1                     1.0000    800  FAS2                     1.0000    800
ACC1                     1.0000    800  INO1                     1.0000    800
OPI3                     1.0000    800
********************************************************************************
```

and the exact command line that was used:

```
********************************************************************************
COMMAND LINE SUMMARY
********************************************************************************
This information can also be useful in the event you wish to report a
problem with the MEME software.

command: meme -mod oops -dna -revcomp -nmotifs 2 -bfile yeast.nc.6.freq INO_up800.s
...
```

Next is detailed information on each motif that was found:

```
********************************************************************************
MOTIF  1        width =   12   sites =   7   llr = 95   E-value = 2.0e-001
********************************************************************************
--------------------------------------------------------------------------------
        Motif 1 Description
--------------------------------------------------------------------------------
Simplified        A  :::9:a::::3:
pos.-specific     C  ::a:9:11691a
probability       G  ::::1::94:4:
matrix            T  aa:1::9::11:
```

To parse this file (stored as `meme.dna.oops.txt`), use

```
>>> with open("meme.INO_up800.classic.oops.xml") as handle:
...     record = motifs.parse(handle, "meme")
...
```

The `motifs.parse` command reads the complete file directly, so you can close the file after calling `motifs.parse`. The header information is stored in attributes:

```
>>> record.version
'5.0.1'
>>> record.datafile
'common/INO_up800.s'
>>> record.command
'meme common/INO_up800.s -oc results/meme10 -mod oops -dna -revcomp -bfile common/yeast.
↪nc.6.freq -nmotifs 2 -objfun classic -minw 8 -nostatus '
>>> record.alphabet
'ACGT'
>>> record.sequences
['sequence_0', 'sequence_1', 'sequence_2', 'sequence_3', 'sequence_4', 'sequence_5',
↪'sequence_6']
```

The record is an object of the `Bio.motifs.meme.Record` class. The class inherits from list, and you can think of `record` as a list of Motif objects:

```
>>> len(record)
2
>>> motif = record[0]
>>> print(motif.consensus)
GCGGCATGTGAAA
>>> print(motif.degenerate_consensus)
GSKGCATGTGAAA
```

In addition to these generic motif attributes, each motif also stores its specific information as calculated by MEME. For example,

```
>>> motif.num_occurrences
7
>>> motif.length
13
>>> evalue = motif.evalue
>>> print("%3.1g" % evalue)
```

(continues on next page)

(continued from previous page)

```
0.2
>>> motif.name
'GSKGCATGTGAAA'
>>> motif.id
'motif_1'
```

In addition to using an index into the record, as we did above, you can also find it by its name:

```
>>> motif = record["GSKGCATGTGAAA"]
```

Each motif has an attribute `.alignment` with the sequence alignment in which the motif was found, providing some information on each of the sequences:

```
>>> len(motif.alignment)
7
>>> motif.alignment.sequences[0]
Instance('GCGGCATGTGAAA')
>>> motif.alignment.sequences[0].motif_name
'GSKGCATGTGAAA'
>>> motif.alignment.sequences[0].sequence_name
'INO1'
>>> motif.alignment.sequences[0].sequence_id
'sequence_5'
>>> motif.alignment.sequences[0].start
620
>>> motif.alignment.sequences[0].strand
'+'
>>> motif.alignment.sequences[0].length
13
>>> pvalue = motif.alignment.sequences[0].pvalue
>>> print("%5.3g" % pvalue)
1.21e-08
```

MAST

18.2.3 TRANSFAC

TRANSFAC is a manually curated database of transcription factors, together with their genomic binding sites and DNA binding profiles [Matys2003]. While the file format used in the TRANSFAC database is nowadays also used by others, we will refer to it as the TRANSFAC file format.

A minimal file in the TRANSFAC format looks as follows:

```
ID  motif1
P0      A      C      G      T
01      1      2      2      0      S
02      2      1      2      0      R
03      3      0      1      1      A
04      0      5      0      0      C
05      5      0      0      0      A
06      0      0      4      1      G
07      0      1      4      0      G
```

(continues on next page)

(continued from previous page)

```
08      0       0       0       5       T
09      0       0       5       0       G
10      0       1       2       2       K
11      0       2       0       3       Y
12      1       0       3       1       G
//
```

This file shows the frequency matrix of motif `motif1` of 12 nucleotides. In general, one file in the TRANSFAC format can contain multiple motifs. For example, this is the contents of the example TRANSFAC file `transfac.dat`:

```
VV  EXAMPLE January 15, 2013
XX
//
ID  motif1
P0      A       C       G       T
01      1       2       2       0       S
02      2       1       2       0       R
03      3       0       1       1       A
...
11      0       2       0       3       Y
12      1       0       3       1       G
//
ID  motif2
P0      A       C       G       T
01      2       1       2       0       R
02      1       2       2       0       S
...
09      0       0       0       5       T
10      0       2       0       3       Y
//
```

To parse a TRANSFAC file, use

```
>>> with open("transfac.dat") as handle:
...     record = motifs.parse(handle, "TRANSFAC")
...
```

If any discrepancies between the file contents and the TRANSFAC file format are detected, a `ValueError` is raised. Note that you may encounter files that do not follow the TRANSFAC format strictly. For example, the number of spaces between columns may be different, or a tab may be used instead of spaces. Use `strict=False` to enable parsing such files without raising a `ValueError`:

```
>>> record = motifs.parse(handle, "TRANSFAC", strict=False)
```

When parsing a non-compliant file, we recommend to check the record returned by `motif.parse` to ensure that it is consistent with the file contents.

The overall version number, if available, is stored as `record.version`:

```
>>> record.version
'EXAMPLE January 15, 2013'
```

Each motif in `record` is in instance of the `Bio.motifs.transfac.Motif` class, which inherits both from the `Bio.motifs.Motif` class and from a Python dictionary. The dictionary uses the two-letter keys to store any additional

information about the motif:

```
>>> motif = record[0]
>>> motif.degenerate_consensus  # Using the Bio.motifs.Motif property
Seq('SRACAGGTGKYG')
>>> motif["ID"]  # Using motif as a dictionary
'motif1'
```

TRANSFAC files are typically much more elaborate than this example, containing lots of additional information about the motif. Table *Fields commonly found in TRANSFAC files* lists the two-letter field codes that are commonly found in TRANSFAC files:

Table 1: Fields commonly found in TRANSFAC files

AC	Accession number
AS	Accession numbers, secondary
BA	Statistical basis
BF	Binding factors
BS	Factor binding sites underlying the matrix
CC	Comments
CO	Copyright notice
DE	Short factor description
DR	External databases
DT	Date created/updated
HC	Subfamilies
HP	Superfamilies
ID	Identifier
NA	Name of the binding factor
OC	Taxonomic classification
OS	Species/Taxon
OV	Older version
PV	Preferred version
TY	Type
XX	Empty line; these are not stored in the Record.

Each motif also has an attribute `.references` containing the references associated with the motif, using these two-letter keys:

Table 2: Fields used to store references in TRANSFAC files

RN	Reference number
RA	Reference authors
RL	Reference data
RT	Reference title
RX	PubMed ID

Printing the motifs writes them out in their native TRANSFAC format:

```
>>> print(record)
VV  EXAMPLE January 15, 2013
XX
//
ID  motif1
```

(continues on next page)

(continued from previous page)

```
XX
P0      A      C      G      T
01      1      2      2      0      S
02      2      1      2      0      R
03      3      0      1      1      A
04      0      5      0      0      C
05      5      0      0      0      A
06      0      0      4      1      G
07      0      1      4      0      G
08      0      0      0      5      T
09      0      0      5      0      G
10      0      1      2      2      K
11      0      2      0      3      Y
12      1      0      3      1      G
XX
//
ID  motif2
XX
P0      A      C      G      T
01      2      1      2      0      R
02      1      2      2      0      S
03      0      5      0      0      C
04      3      0      1      1      A
05      0      0      4      1      G
06      5      0      0      0      A
07      0      1      4      0      G
08      0      0      5      0      G
09      0      0      0      5      T
10      0      2      0      3      Y
XX
//
```

You can export the motifs in the TRANSFAC format by capturing this output in a string and saving it in a file:

```
>>> text = str(record)
>>> with open("mytransfacfile.dat", "w") as out_handle:
...     out_handle.write(text)
...
```

18.3 Writing motifs

Speaking of exporting, let's look at export functions in general. We can use the `format` built-in function to write the motif in the simple JASPAR `pfm` format:

```
>>> print(format(arnt, "pfm"))
  4.00  19.00   0.00   0.00   0.00   0.00
 16.00   0.00  20.00   0.00   0.00   0.00
  0.00   1.00   0.00  20.00   0.00  20.00
  0.00   0.00   0.00   0.00  20.00   0.00
```

Similarly, we can use `format` to write the motif in the JASPAR `jaspar` format:

```
>>> print(format(arnt, "jaspar"))
>MA0004.1  Arnt
A [  4.00  19.00   0.00   0.00   0.00   0.00]
C [ 16.00   0.00  20.00   0.00   0.00   0.00]
G [  0.00   1.00   0.00  20.00   0.00  20.00]
T [  0.00   0.00   0.00   0.00  20.00   0.00]
```

To write the motif in a TRANSFAC-like matrix format, use

```
>>> print(format(m, "transfac"))
P0      A      C      G      T
01      3      0      0      4      W
02      7      0      0      0      A
03      0      5      0      2      C
04      2      2      3      0      V
05      1      6      0      0      C
XX
//
```

To write out multiple motifs, you can use `motifs.write`. This function can be used regardless of whether the motifs originated from a TRANSFAC file. For example,

```
>>> two_motifs = [arnt, srf]
>>> print(motifs.write(two_motifs, "transfac"))
P0      A      C      G      T
01      4     16      0      0      C
02     19      0      1      0      A
03      0     20      0      0      C
04      0      0     20      0      G
05      0      0      0     20      T
06      0      0     20      0      G
XX
//
P0      A      C      G      T
01      2      1     39      4      G
02      9     33      2      2      C
03      0     45      1      0      C
04      1     45      0      0      C
05     32      1      0     13      A
06      3      1      0     42      T
07     46      0      0      0      A
08      1      0      0     45      T
09     43      0      0      3      A
10     15      1      0     30      W
11      2      0     44      0      G
12      2      1     43      0      G
XX
//
```

Or, to write multiple motifs in the `jaspar` format:

```
>>> two_motifs = [arnt, mef2a]
>>> print(motifs.write(two_motifs, "jaspar"))
```

(continues on next page)

(continued from previous page)

```
>MA0004.1  Arnt
A [  4.00  19.00   0.00   0.00   0.00   0.00]
C [ 16.00   0.00  20.00   0.00   0.00   0.00]
G [  0.00   1.00   0.00  20.00   0.00  20.00]
T [  0.00   0.00   0.00   0.00  20.00   0.00]
>MA0052.1  MEF2A
A [  1.00   0.00  57.00   2.00   9.00   6.00  37.00   2.00  56.00   6.00]
C [ 50.00   0.00   1.00   1.00   0.00   0.00   0.00   0.00   0.00   0.00]
G [  0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   2.00  50.00]
T [  7.00  58.00   0.00  55.00  49.00  52.00  21.00  56.00   0.00   2.00]
```

18.4 Position-Weight Matrices

The `.counts` attribute of a Motif object shows how often each nucleotide appeared at each position along the alignment. We can normalize this matrix by dividing by the number of instances in the alignment, resulting in the probability of each nucleotide at each position along the alignment. We refer to these probabilities as the position-weight matrix. However, beware that in the literature this term may also be used to refer to the position-specific scoring matrix, which we discuss below.

Usually, pseudocounts are added to each position before normalizing. This avoids overfitting of the position-weight matrix to the limited number of motif instances in the alignment, and can also prevent probabilities from becoming zero. To add a fixed pseudocount to all nucleotides at all positions, specify a number for the `pseudocounts` argument:

```
>>> pwm = m.counts.normalize(pseudocounts=0.5)
>>> print(pwm)
        0      1      2      3      4
A:   0.39   0.83   0.06   0.28   0.17
C:   0.06   0.06   0.61   0.28   0.72
G:   0.06   0.06   0.06   0.39   0.06
T:   0.50   0.06   0.28   0.06   0.06
```

Alternatively, `pseudocounts` can be a dictionary specifying the pseudocounts for each nucleotide. For example, as the GC content of the human genome is about 40%, you may want to choose the pseudocounts accordingly:

```
>>> pwm = m.counts.normalize(pseudocounts={"A": 0.6, "C": 0.4, "G": 0.4, "T": 0.6})
>>> print(pwm)
        0      1      2      3      4
A:   0.40   0.84   0.07   0.29   0.18
C:   0.04   0.04   0.60   0.27   0.71
G:   0.04   0.04   0.04   0.38   0.04
T:   0.51   0.07   0.29   0.07   0.07
```

The position-weight matrix has its own methods to calculate the consensus, anticonsensus, and degenerate consensus sequences:

```
>>> pwm.consensus
Seq('TACGC')
>>> pwm.anticonsensus
Seq('CCGTG')
>>> pwm.degenerate_consensus
Seq('WACNC')
```

Note that due to the pseudocounts, the degenerate consensus sequence calculated from the position-weight matrix is slightly different from the degenerate consensus sequence calculated from the instances in the motif:

```
>>> m.degenerate_consensus
Seq('WACVC')
```

The reverse complement of the position-weight matrix can be calculated directly from the `pwm`:

```
>>> rpwm = pwm.reverse_complement()
>>> print(rpwm)
        0      1      2      3      4
A:   0.07   0.07   0.29   0.07   0.51
C:   0.04   0.38   0.04   0.04   0.04
G:   0.71   0.27   0.60   0.04   0.04
T:   0.18   0.29   0.07   0.84   0.40
```

18.5 Position-Specific Scoring Matrices

Using the background distribution and PWM with pseudo-counts added, it's easy to compute the log-odds ratios, telling us what are the log odds of a particular symbol to be coming from a motif against the background. We can use the `.log_odds()` method on the position-weight matrix:

```
>>> pssm = pwm.log_odds()
>>> print(pssm)
        0      1      2      3      4
A:   0.68   1.76  -1.91   0.21  -0.49
C:  -2.49  -2.49   1.26   0.09   1.51
G:  -2.49  -2.49  -2.49   0.60  -2.49
T:   1.03  -1.91   0.21  -1.91  -1.91
```

Here we can see positive values for symbols more frequent in the motif than in the background and negative for symbols more frequent in the background. 0.0 means that it's equally likely to see a symbol in the background and in the motif.

This assumes that A, C, G, and T are equally likely in the background. To calculate the position-specific scoring matrix against a background with unequal probabilities for A, C, G, T, use the `background` argument. For example, against a background with a 40% GC content, use

```
>>> background = {"A": 0.3, "C": 0.2, "G": 0.2, "T": 0.3}
>>> pssm = pwm.log_odds(background)
>>> print(pssm)
        0      1      2      3      4
A:   0.42   1.49  -2.17  -0.05  -0.75
C:  -2.17  -2.17   1.58   0.42   1.83
G:  -2.17  -2.17  -2.17   0.92  -2.17
T:   0.77  -2.17  -0.05  -2.17  -2.17
```

The maximum and minimum score obtainable from the PSSM are stored in the `.max` and `.min` properties:

```
>>> print("%4.2f" % pssm.max)
6.59
>>> print("%4.2f" % pssm.min)
-10.85
```

The mean and standard deviation of the PSSM scores with respect to a specific background are calculated by the `.mean` and `.std` methods.

```
>>> mean = pssm.mean(background)
>>> std = pssm.std(background)
>>> print("mean = %0.2f, standard deviation = %0.2f" % (mean, std))
mean = 3.21, standard deviation = 2.59
```

A uniform background is used if `background` is not specified. The mean is equal to the Kullback-Leibler divergence or relative entropy described in Section *Relative entropy*.

The `.reverse_complement`, `.consensus`, `.anticonsensus`, and `.degenerate_consensus` methods can be applied directly to PSSM objects.

18.6 Searching for instances

The most frequent use for a motif is to find its instances in some sequence. For the sake of this section, we will use an artificial sequence like this:

```
>>> test_seq = Seq("TACACTGCATTACAACCCAAGCATTA")
>>> len(test_seq)
26
```

18.6.1 Searching for exact matches

The simplest way to find instances, is to look for exact matches of the true instances of the motif:

```
>>> for pos, seq in test_seq.search(m.alignment):
...     print("%i %s" % (pos, seq))
...
0 TACAC
10 TACAA
13 AACCC
```

We can do the same with the reverse complement (to find instances on the complementary strand):

```
>>> for pos, seq in test_seq.search(r.alignment):
...     print("%i %s" % (pos, seq))
...
6 GCATT
20 GCATT
```

18.6.2 Searching for matches using the PSSM score

It's just as easy to look for positions, giving rise to high log-odds scores against our motif:

```
>>> for position, score in pssm.search(test_seq, threshold=3.0):
...     print("Position %d: score = %5.3f" % (position, score))
...
Position 0: score = 5.622
Position -20: score = 4.601
Position 10: score = 3.037
Position 13: score = 5.738
Position -6: score = 4.601
```

The negative positions refer to instances of the motif found on the reverse strand of the test sequence, and follow the Python convention on negative indices. Therefore, the instance of the motif at `pos` is located at `test_seq[pos:pos+len(m)]` both for positive and for negative values of `pos`.

You may notice the threshold parameter, here set arbitrarily to 3.0. This is in log_2, so we are now looking only for words, which are eight times more likely to occur under the motif model than in the background. The default threshold is 0.0, which selects everything that looks more like the motif than the background.

You can also calculate the scores at all positions along the sequence:

```
>>> pssm.calculate(test_seq)
array([  5.62230396,  -5.6796999 ,  -3.43177247,   0.93827754,
        -6.84962511,  -2.04066086, -10.84962463,  -3.65614533,
        -0.03370807,  -3.91102552,   3.03734159,  -2.14918518,
        -0.6016975 ,   5.7381525 ,  -0.50977498,  -3.56422281,
        -8.73414803,  -0.09919716,  -0.6016975 ,  -2.39429784,
       -10.84962463,  -3.65614533], dtype=float32)
```

In general, this is the fastest way to calculate PSSM scores. The scores returned by `pssm.calculate` are for the forward strand only. To obtain the scores on the reverse strand, you can take the reverse complement of the PSSM:

```
>>> rpssm = pssm.reverse_complement()
>>> rpssm.calculate(test_seq)
array([ -9.43458748,  -3.06172252,  -7.18665981,  -7.76216221,
        -2.04066086,  -4.26466274,   4.60124254,  -4.2480607 ,
        -8.73414803,  -2.26503372,  -6.49598789,  -5.64668512,
        -8.73414803, -10.84962463,  -4.82356262,  -4.82356262,
        -5.64668512,  -8.73414803,  -4.15613794,  -5.6796999 ,
         4.60124254,  -4.2480607 ], dtype=float32)
```

18.6.3 Selecting a score threshold

If you want to use a less arbitrary way of selecting thresholds, you can explore the distribution of PSSM scores. Since the space for a score distribution grows exponentially with motif length, we are using an approximation with a given precision to keep computation cost manageable:

```
>>> distribution = pssm.distribution(background=background, precision=10**4)
```

The `distribution` object can be used to determine a number of different thresholds. We can specify the requested false-positive rate (probability of "finding" a motif instance in background generated sequence):

```
>>> threshold = distribution.threshold_fpr(0.01)
>>> print("%5.3f" % threshold)
4.009
```

or the false-negative rate (probability of "not finding" an instance generated from the motif):

```
>>> threshold = distribution.threshold_fnr(0.1)
>>> print("%5.3f" % threshold)
-0.510
```

or a threshold (approximately) satisfying some relation between the false-positive rate and the false-negative rate ($\frac{\text{fnr}}{\text{fpr}} \simeq t$):

```
>>> threshold = distribution.threshold_balanced(1000)
>>> print("%5.3f" % threshold)
6.241
```

or a threshold satisfying (roughly) the equality between the $-log$ of the false-positive rate and the information content (as used in patser software by Hertz and Stormo):

```
>>> threshold = distribution.threshold_patser()
>>> print("%5.3f" % threshold)
0.346
```

For example, in case of our motif, you can get the threshold giving you exactly the same results (for this sequence) as searching for instances with balanced threshold with rate of 1000.

```
>>> threshold = distribution.threshold_fpr(0.01)
>>> print("%5.3f" % threshold)
4.009
>>> for position, score in pssm.search(test_seq, threshold=threshold):
...     print("Position %d: score = %5.3f" % (position, score))
...
Position 0: score = 5.622
Position -20: score = 4.601
Position 13: score = 5.738
Position -6: score = 4.601
```

18.7 Each motif object has an associated Position-Specific Scoring Matrix

To facilitate searching for potential TFBSs using PSSMs, both the position-weight matrix and the position-specific scoring matrix are associated with each motif. Using the Arnt motif as an example:

```
>>> from Bio import motifs
>>> with open("Arnt.sites") as handle:
...     motif = motifs.read(handle, "sites")
...
>>> print(motif.counts)
        0      1      2      3      4      5
A:   4.00  19.00   0.00   0.00   0.00   0.00
```

(continues on next page)

(continued from previous page)

```
C:  16.00   0.00  20.00   0.00   0.00   0.00
G:   0.00   1.00   0.00  20.00   0.00  20.00
T:   0.00   0.00   0.00   0.00  20.00   0.00

>>> print(motif.pwm)
        0      1      2      3      4      5
A:   0.20   0.95   0.00   0.00   0.00   0.00
C:   0.80   0.00   1.00   0.00   0.00   0.00
G:   0.00   0.05   0.00   1.00   0.00   1.00
T:   0.00   0.00   0.00   0.00   1.00   0.00

>>> print(motif.pssm)
        0      1      2      3      4      5
A:  -0.32   1.93   -inf   -inf   -inf   -inf
C:   1.68   -inf   2.00   -inf   -inf   -inf
G:   -inf  -2.32   -inf   2.00   -inf   2.00
T:   -inf   -inf   -inf   -inf   2.00   -inf
```

The negative infinities appear here because the corresponding entry in the frequency matrix is 0, and we are using zero pseudocounts by default:

```
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.pseudocounts[letter]))
...
A: 0.00
C: 0.00
G: 0.00
T: 0.00
```

If you change the `.pseudocounts` attribute, the position-frequency matrix and the position-specific scoring matrix are recalculated automatically:

```
>>> motif.pseudocounts = 3.0
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.pseudocounts[letter]))
...
A: 3.00
C: 3.00
G: 3.00
T: 3.00
>>> print(motif.pwm)
        0      1      2      3      4      5
A:   0.22   0.69   0.09   0.09   0.09   0.09
C:   0.59   0.09   0.72   0.09   0.09   0.09
G:   0.09   0.12   0.09   0.72   0.09   0.72
T:   0.09   0.09   0.09   0.09   0.72   0.09
```

```
>>> print(motif.pssm)
        0      1      2      3      4      5
A:  -0.19   1.46  -1.42  -1.42  -1.42  -1.42
C:   1.25  -1.42   1.52  -1.42  -1.42  -1.42
G:  -1.42  -1.00  -1.42   1.52  -1.42   1.52
```

(continues on next page)

(continued from previous page)

```
T:  -1.42  -1.42  -1.42  -1.42   1.52  -1.42
```

You can also set the `.pseudocounts` to a dictionary over the four nucleotides if you want to use different pseudocounts for them. Setting `motif.pseudocounts` to `None` resets it to its default value of zero.

The position-specific scoring matrix depends on the background distribution, which is uniform by default:

```
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.background[letter]))
...
A: 0.25
C: 0.25
G: 0.25
T: 0.25
```

Again, if you modify the background distribution, the position-specific scoring matrix is recalculated:

```
>>> motif.background = {"A": 0.2, "C": 0.3, "G": 0.3, "T": 0.2}
>>> print(motif.pssm)
        0      1      2      3      4      5
A:   0.13   1.78  -1.09  -1.09  -1.09  -1.09
C:   0.98  -1.68   1.26  -1.68  -1.68  -1.68
G:  -1.68  -1.26  -1.68   1.26  -1.68   1.26
T:  -1.09  -1.09  -1.09  -1.09   1.85  -1.09
```

Setting `motif.background` to `None` resets it to a uniform distribution:

```
>>> motif.background = None
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.background[letter]))
...
A: 0.25
C: 0.25
G: 0.25
T: 0.25
```

If you set `motif.background` equal to a single value, it will be interpreted as the GC content:

```
>>> motif.background = 0.8
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.background[letter]))
...
A: 0.10
C: 0.40
G: 0.40
T: 0.10
```

Note that you can now calculate the mean of the PSSM scores over the background against which it was computed:

```
>>> print("%f" % motif.pssm.mean(motif.background))
4.703928
```

as well as its standard deviation:

```
>>> print("%f" % motif.pssm.std(motif.background))
3.290900
```

and its distribution:

```
>>> distribution = motif.pssm.distribution(background=motif.background)
>>> threshold = distribution.threshold_fpr(0.01)
>>> print("%f" % threshold)
3.854375
```

Note that the position-weight matrix and the position-specific scoring matrix are recalculated each time you call `motif.pwm` or `motif.pssm`, respectively. If speed is an issue and you want to use the PWM or PSSM repeatedly, you can save them as a variable, as in

```
>>> pssm = motif.pssm
```

18.8 Comparing motifs

Once we have more than one motif, we might want to compare them.

Before we start comparing motifs, I should point out that motif boundaries are usually quite arbitrary. This means we often need to compare motifs of different lengths, so comparison needs to involve some kind of alignment. This means we have to take into account two things:

- alignment of motifs
- some function to compare aligned motifs

To align the motifs, we use ungapped alignment of PSSMs and substitute zeros for any missing columns at the beginning and end of the matrices. This means that effectively we are using the background distribution for columns missing from the PSSM. The distance function then returns the minimal distance between motifs, as well as the corresponding offset in their alignment.

To give an example, let us first load another motif, which is similar to our test motif `m`:

```
>>> with open("REB1.pfm") as handle:
...     m_reb1 = motifs.read(handle, "pfm")
...
>>> m_reb1.consensus
Seq('GTTACCCGG')
>>> print(m_reb1.counts)
        0      1      2      3      4      5      6      7      8
A:  30.00   0.00   0.00 100.00   0.00   0.00   0.00   0.00  15.00
C:  10.00   0.00   0.00   0.00 100.00 100.00 100.00   0.00  15.00
G:  50.00   0.00   0.00   0.00   0.00   0.00   0.00  60.00  55.00
T:  10.00 100.00 100.00   0.00   0.00   0.00   0.00  40.00  15.00
```

To make the motifs comparable, we choose the same values for the pseudocounts and the background distribution as our motif `m`:

```
>>> m_reb1.pseudocounts = {"A": 0.6, "C": 0.4, "G": 0.4, "T": 0.6}
>>> m_reb1.background = {"A": 0.3, "C": 0.2, "G": 0.2, "T": 0.3}
>>> pssm_reb1 = m_reb1.pssm
>>> print(pssm_reb1)
```

(continues on next page)

(continued from previous page)

```
        0      1      2      3      4      5      6      7      8
A:   0.00  -5.67  -5.67   1.72  -5.67  -5.67  -5.67  -5.67  -0.97
C:  -0.97  -5.67  -5.67  -5.67   2.30   2.30   2.30  -5.67  -0.41
G:   1.30  -5.67  -5.67  -5.67  -5.67  -5.67  -5.67   1.57   1.44
T:  -1.53   1.72   1.72  -5.67  -5.67  -5.67  -5.67   0.41  -0.97
```

We'll compare these motifs using the Pearson correlation. Since we want it to resemble a distance measure, we actually take $1 - r$, where r is the Pearson correlation coefficient (PCC):

```
>>> distance, offset = pssm.dist_pearson(pssm_reb1)
>>> print("distance = %5.3g" % distance)
distance = 0.239
>>> print(offset)
-2
```

This means that the best PCC between motif `m` and `m_reb1` is obtained with the following alignment:

```
m:      bbTACGCbb
m_reb1: GTTACCCGG
```

where `b` stands for background distribution. The PCC itself is roughly $1 - 0.239 = 0.761$.

18.9 *De novo* motif finding

Currently, Biopython has only limited support for *de novo* motif finding. Namely, we support running `xxmotif` and also parsing of MEME. Since the number of motif finding tools is growing rapidly, contributions of new parsers are welcome.

18.9.1 MEME

Let's assume, you have run MEME on sequences of your choice with your favorite parameters and saved the output in the file `meme.out`. You can retrieve the motifs reported by MEME by running the following piece of code:

```
>>> from Bio import motifs
>>> with open("meme.psp_test.classic.zoops.xml") as handle:
...     motifsM = motifs.parse(handle, "meme")
...
```

```
>>> motifsM
[<Bio.motifs.meme.Motif object at 0xc356b0>]
```

Besides the most wanted list of motifs, the result object contains more useful information, accessible through properties with self-explanatory names:

- `.alphabet`
- `.datafile`
- `.sequences`
- `.version`
- `.command`

The motifs returned by the MEME Parser can be treated exactly like regular Motif objects (with instances), they also provide some extra functionality, by adding additional information about the instances.

```
>>> motifsM[0].consensus
Seq('GCTTATGTAA')
>>> motifsM[0].alignment.sequences[0].sequence_name
'iYFL005W'
>>> motifsM[0].alignment.sequences[0].sequence_id
'sequence_15'
>>> motifsM[0].alignment.sequences[0].start
480
>>> motifsM[0].alignment.sequences[0].strand
'+'
```

```
>>> motifsM[0].alignment.sequences[0].pvalue
1.97e-06
```

18.10 Useful links

- Sequence motif in wikipedia
- PWM in wikipedia
- Consensus sequence in wikipedia
- Comparison of different motif finding programs

CHAPTER

NINETEEN

CLUSTER ANALYSIS

Cluster analysis is the grouping of items into clusters based on the similarity of the items to each other. In bioinformatics, clustering is widely used in gene expression data analysis to find groups of genes with similar gene expression profiles. This may identify functionally related genes, as well as suggest the function of presently unknown genes.

The Biopython module `Bio.Cluster` provides commonly used clustering algorithms and was designed with the application to gene expression data in mind. However, this module can also be used for cluster analysis of other types of data. `Bio.Cluster` and the underlying C Clustering Library is described by De Hoon *et al.* [DeHoon2004].

The following four clustering approaches are implemented in `Bio.Cluster`:

- Hierarchical clustering (pairwise centroid-, single-, complete-, and average-linkage);
- k-means, k-medians, and k-medoids clustering;
- Self-Organizing Maps;
- Principal Component Analysis.

19.1 Data representation

The data to be clustered are represented by a $n \times m$ Numerical Python array `data`. Within the context of gene expression data clustering, typically the rows correspond to different genes whereas the columns correspond to different experimental conditions. The clustering algorithms in `Bio.Cluster` can be applied both to rows (genes) and to columns (experiments).

19.2 Missing values

The $n \times m$ Numerical Python integer array `mask` indicates if any of the values in `data` are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing and is ignored in the analysis.

19.3 Random number generator

The k-means/medians/medoids clustering algorithms and Self-Organizing Maps (SOMs) include the use of a random number generator. The uniform random number generator in `Bio.Cluster` is based on the algorithm by L'Ecuyer [Lecuyer1988], while random numbers following the binomial distribution are generated using the BTPE algorithm by Kachitvichyanukul and Schmeiser [Kachitvichyanukul1988]. The random number generator is initialized automatically during its first call. As this random number generator uses a combination of two multiplicative linear congruential generators, two (integer) seeds are needed for initialization, for which we use the system-supplied random number generator `rand` (in the C standard library). We initialize this generator by calling `srand` with the epoch time in seconds,

and use the first two random numbers generated by `rand` as seeds for the uniform random number generator in `Bio.Cluster`.

19.3.1 Distance functions

In order to cluster items into groups based on their similarity, we should first define what exactly we mean by *similar*. `Bio.Cluster` provides eight distance functions, indicated by a single character, to measure similarity, or conversely, distance:

- `'e'`: Euclidean distance;
- `'b'`: City-block distance.
- `'c'`: Pearson correlation coefficient;
- `'a'`: Absolute value of the Pearson correlation coefficient;
- `'u'`: Uncentered Pearson correlation (equivalent to the cosine of the angle between two data vectors);
- `'x'`: Absolute uncentered Pearson correlation;
- `'s'`: Spearman's rank correlation;
- `'k'`: Kendall's τ.

The first two are true distance functions that satisfy the triangle inequality:

$$d(\underline{u}, \underline{v}) \leq d(\underline{u}, \underline{w}) + d(\underline{w}, \underline{v}) \text{ for all } \underline{u}, \underline{v}, \underline{w},$$

and are therefore referred to as *metrics*. In everyday language, this means that the shortest distance between two points is a straight line.

The remaining six distance measures are related to the correlation coefficient, where the distance d is defined in terms of the correlation r by $d = 1 - r$. Note that these distance functions are *semi-metrics* that do not satisfy the triangle inequality. For example, for

$$\underline{u} = (1, 0, -1);$$

$$\underline{v} = (1, 1, 0);$$

$$\underline{w} = (0, 1, 1);$$

we find a Pearson distance $d(\underline{u}, \underline{w}) = 1.8660$, while $d(\underline{u}, \underline{v}) + d(\underline{v}, \underline{w}) = 1.6340$.

19.4 Euclidean distance

In `Bio.Cluster`, we define the Euclidean distance as

$$d = \frac{1}{n} \sum_{i=1}^{n} (x_i - y_i)^2.$$

Only those terms are included in the summation for which both x_i and y_i are present, and the denominator n is chosen accordingly. As the expression data x_i and y_i are subtracted directly from each other, we should make sure that the expression data are properly normalized when using the Euclidean distance.

19.5 City-block distance

The city-block distance, alternatively known as the Manhattan distance, is related to the Euclidean distance. Whereas the Euclidean distance corresponds to the length of the shortest path between two points, the city-block distance is the sum of distances along each dimension. As gene expression data tend to have missing values, in `Bio.Cluster` we define the city-block distance as the sum of distances divided by the number of dimensions:

$$d = \frac{1}{n}\sum_{i=1}^{n}|x_i - y_i|\,.$$

This is equal to the distance you would have to walk between two points in a city, where you have to walk along city blocks. As for the Euclidean distance, the expression data are subtracted directly from each other, and we should therefore make sure that they are properly normalized.

19.6 The Pearson correlation coefficient

The Pearson correlation coefficient is defined as

$$r = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{\sigma_x}\right)\left(\frac{y_i - \bar{y}}{\sigma_y}\right),$$

in which $\bar{x}, \bar{y}$ are the sample mean of x and y respectively, and σ_x, σ_y are the sample standard deviation of x and y. The Pearson correlation coefficient is a measure for how well a straight line can be fitted to a scatterplot of x and y. If all the points in the scatterplot lie on a straight line, the Pearson correlation coefficient is either +1 or -1, depending on whether the slope of line is positive or negative. If the Pearson correlation coefficient is equal to zero, there is no correlation between x and y.

The *Pearson distance* is then defined as

$$d_{\mathrm{P}} \equiv 1 - r.$$

As the Pearson correlation coefficient lies between -1 and 1, the Pearson distance lies between 0 and 2.

19.7 Absolute Pearson correlation

By taking the absolute value of the Pearson correlation, we find a number between 0 and 1. If the absolute value is 1, all the points in the scatter plot lie on a straight line with either a positive or a negative slope. If the absolute value is equal to zero, there is no correlation between x and y.

The corresponding distance is defined as

$$d_{\mathrm{A}} \equiv 1 - |r|\,,$$

where r is the Pearson correlation coefficient. As the absolute value of the Pearson correlation coefficient lies between 0 and 1, the corresponding distance lies between 0 and 1 as well.

In the context of gene expression experiments, the absolute correlation is equal to 1 if the gene expression profiles of two genes are either exactly the same or exactly opposite. The absolute correlation coefficient should therefore be used with care.

19.8 Uncentered correlation (cosine of the angle)

In some cases, it may be preferable to use the *uncentered correlation* instead of the regular Pearson correlation coefficient. The uncentered correlation is defined as

$$r_{\mathrm{U}} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{x_i}{\sigma_x^{(0)}}\right)\left(\frac{y_i}{\sigma_y^{(0)}}\right),$$

where

$$\sigma_x^{(0)} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2};$$

$$\sigma_y^{(0)} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} y_i^2}.$$

This is the same expression as for the regular Pearson correlation coefficient, except that the sample means $\bar{x}, \bar{y}$ are set equal to zero. The uncentered correlation may be appropriate if there is a zero reference state. For instance, in the case of gene expression data given in terms of log-ratios, a log-ratio equal to zero corresponds to the green and red signal being equal, which means that the experimental manipulation did not affect the gene expression.

The distance corresponding to the uncentered correlation coefficient is defined as

$$d_{\mathrm{U}} \equiv 1 - r_{\mathrm{U}},$$

where r_{U} is the uncentered correlation. As the uncentered correlation coefficient lies between -1 and 1, the corresponding distance lies between 0 and 2.

The uncentered correlation is equal to the cosine of the angle of the two data vectors in n-dimensional space, and is often referred to as such.

19.9 Absolute uncentered correlation

As for the regular Pearson correlation, we can define a distance measure using the absolute value of the uncentered correlation:

$$d_{\mathrm{AU}} \equiv 1 - \left|r_{\mathrm{U}}\right|,$$

where r_{U} is the uncentered correlation coefficient. As the absolute value of the uncentered correlation coefficient lies between 0 and 1, the corresponding distance lies between 0 and 1 as well.

Geometrically, the absolute value of the uncentered correlation is equal to the cosine between the supporting lines of the two data vectors (i.e., the angle without taking the direction of the vectors into consideration).

19.10 Spearman rank correlation

The Spearman rank correlation is an example of a non-parametric similarity measure, and tends to be more robust against outliers than the Pearson correlation.

To calculate the Spearman rank correlation, we replace each data value by their rank if we would order the data in each vector by their value. We then calculate the Pearson correlation between the two rank vectors instead of the data vectors.

As in the case of the Pearson correlation, we can define a distance measure corresponding to the Spearman rank correlation as

$$d_{\mathrm{S}} \equiv 1 - r_{\mathrm{S}},$$

where r_{S} is the Spearman rank correlation.

19.11 Kendall's τ

Kendall's τ is another example of a non-parametric similarity measure. It is similar to the Spearman rank correlation, but instead of the ranks themselves only the relative ranks are used to calculate τ (see Snedecor & Cochran [Snedecor1989]).

We can define a distance measure corresponding to Kendall's τ as

$$d_{\mathrm{K}} \equiv 1 - \tau.$$

As Kendall's τ is always between -1 and 1, the corresponding distance will be between 0 and 2.

19.12 Weighting

For most of the distance functions available in `Bio.Cluster`, a weight vector can be applied. The weight vector contains weights for the items in the data vector. If the weight for item i is w_i, then that item is treated as if it occurred w_i times in the data. The weight do not have to be integers.

19.13 Calculating the distance matrix

The distance matrix is a square matrix with all pairwise distances between the items in `data`, and can be calculated by the function `distancematrix` in the `Bio.Cluster` module:

```
>>> from Bio.Cluster import distancematrix
>>> matrix = distancematrix(data)
```

where the following arguments are defined:

- `data` (required)
  Array containing the data for the items.
- `mask` (default: `None`)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.
- `weight` (default: `None`)
  The weights to be used when calculating distances. If `weight` is `None`, then equal weights are assumed.
- `transpose` (default: `0`)
  Determines if the distances between the rows of `data` are to be calculated (`transpose` is `False`), or between the columns of `data` (`transpose` is `True`).
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see *Distance functions*).

To save memory, the distance matrix is returned as a list of 1D arrays. The number of columns in each row is equal to the row number. Hence, the first row has zero elements. For example,

```
>>> from numpy import array
>>> from Bio.Cluster import distancematrix
>>> data = array([[0, 1,  2,  3],
...               [4, 5,  6,  7],
...               [8, 9, 10, 11],
...               [1, 2,  3,  4]])  # fmt: skip
...
>>> distances = distancematrix(data, dist="e")
```

yields a distance matrix

```
>>> distances
[array([], dtype=float64), array([ 16.]), array([ 64.,  16.]), array([  1.,   9.,  49.])]
```

which can be rewritten as

```
[array([], dtype=float64), array([16.0]), array([64.0, 16.0]), array([1.0, 9.0, 49.0])]
```

This corresponds to the distance matrix:

$$\begin{pmatrix} 0 & 16 & 64 & 1 \\ 16 & 0 & 16 & 9 \\ 64 & 16 & 0 & 49 \\ 1 & 9 & 49 & 0 \end{pmatrix}.$$

19.13.1 Calculating cluster properties

19.14 Calculating the cluster centroids

The centroid of a cluster can be defined either as the mean or as the median of each dimension over all cluster items. The function `clustercentroids` in `Bio.Cluster` can be used to calculate either:

```
>>> from Bio.Cluster import clustercentroids
>>> cdata, cmask = clustercentroids(data)
```

where the following arguments are defined:

- `data` (required)
  Array containing the data for the items.
- `mask` (default: `None`)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.
- `clusterid` (default: `None`)
  Vector of integers showing to which cluster each item belongs. If `clusterid` is `None`, then all items are assumed to belong to the same cluster.
- `method` (default: `'a'`)
  Specifies whether the arithmetic mean (`method=='a'`) or the median (`method=='m'`) is used to calculate the cluster center.

- `transpose` (default: `0`)
  Determines if the centroids of the rows of `data` are to be calculated (`transpose` is `False`), or the centroids of the columns of `data` (`transpose` is `True`).

This function returns the tuple `(cdata, cmask)`. The centroid data are stored in the 2D Numerical Python array `cdata`, with missing data indicated by the 2D Numerical Python integer array `cmask`. The dimensions of these arrays are (number of clusters, number of columns) if `transpose` is `0`, or (number of rows, number of clusters) if `transpose` is `1`. Each row (if `transpose` is `0`) or column (if `transpose` is `1`) contains the averaged data corresponding to the centroid of each cluster.

19.15 Calculating the distance between clusters

Given a distance function between *items*, we can define the distance between two *clusters* in several ways. The distance between the arithmetic means of the two clusters is used in pairwise centroid-linkage clustering and in k-means clustering. In k-medoids clustering, the distance between the medians of the two clusters is used instead. The shortest pairwise distance between items of the two clusters is used in pairwise single-linkage clustering, while the longest pairwise distance is used in pairwise maximum-linkage clustering. In pairwise average-linkage clustering, the distance between two clusters is defined as the average over the pairwise distances.

To calculate the distance between two clusters, use

```
>>> from Bio.Cluster import clusterdistance
>>> distance = clusterdistance(data)
```

where the following arguments are defined:

- `data` (required)
  Array containing the data for the items.
- `mask` (default: `None`)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.
- `weight` (default: `None`)
  The weights to be used when calculating distances. If `weight` is `None`, then equal weights are assumed.
- `index1` (default: `0`)
  A list containing the indices of the items belonging to the first cluster. A cluster containing only one item i can be represented either as a list `[i]`, or as an integer `i`.
- `index2` (default: `0`)
  A list containing the indices of the items belonging to the second cluster. A cluster containing only one items i can be represented either as a list `[i]`, or as an integer `i`.
- `method` (default: `'a'`)
  Specifies how the distance between clusters is defined:
  - `'a'`: Distance between the two cluster centroids (arithmetic mean);
  - `'m'`: Distance between the two cluster centroids (median);
  - `'s'`: Shortest pairwise distance between items in the two clusters;
  - `'x'`: Longest pairwise distance between items in the two clusters;
  - `'v'`: Average over the pairwise distances between items in the two clusters.
- `dist` (default: `'e'`, Euclidean distance)

Defines the distance function to be used (see *Distance functions*).

- `transpose` (default: `0`)
  If `transpose` is `False`, calculate the distance between the rows of `data`. If `transpose` is `True`, calculate the distance between the columns of `data`.

19.15.1 Partitioning algorithms

Partitioning algorithms divide items into k clusters such that the sum of distances over the items to their cluster centers is minimal. The number of clusters k is specified by the user. Three partitioning algorithms are available in `Bio.Cluster`:

- k-means clustering
- k-medians clustering
- k-medoids clustering

These algorithms differ in how the cluster center is defined. In k-means clustering, the cluster center is defined as the mean data vector averaged over all items in the cluster. Instead of the mean, in k-medians clustering the median is calculated for each dimension in the data vector. Finally, in k-medoids clustering the cluster center is defined as the item which has the smallest sum of distances to the other items in the cluster. This clustering algorithm is suitable for cases in which the distance matrix is known but the original data matrix is not available, for example when clustering proteins based on their structural similarity.

The expectation-maximization (EM) algorithm is used to find this partitioning into k groups. In the initialization of the EM algorithm, we randomly assign items to clusters. To ensure that no empty clusters are produced, we use the binomial distribution to randomly choose the number of items in each cluster to be one or more. We then randomly permute the cluster assignments to items such that each item has an equal probability to be in any cluster. Each cluster is thus guaranteed to contain at least one item.

We then iterate:

- Calculate the centroid of each cluster, defined as either the mean, the median, or the medoid of the cluster;
- Calculate the distances of each item to the cluster centers;
- For each item, determine which cluster centroid is closest;
- Reassign each item to its closest cluster, or stop the iteration if no further item reassignments take place.

To avoid clusters becoming empty during the iteration, in k-means and k-medians clustering the algorithm keeps track of the number of items in each cluster, and prohibits the last remaining item in a cluster from being reassigned to a different cluster. For k-medoids clustering, such a check is not needed, as the item that functions as the cluster centroid has a zero distance to itself, and will therefore never be closer to a different cluster.

As the initial assignment of items to clusters is done randomly, usually a different clustering solution is found each time the EM algorithm is executed. To find the optimal clustering solution, the k-means algorithm is repeated many times, each time starting from a different initial random clustering. The sum of distances of the items to their cluster center is saved for each run, and the solution with the smallest value of this sum will be returned as the overall clustering solution.

How often the EM algorithm should be run depends on the number of items being clustered. As a rule of thumb, we can consider how often the optimal solution was found; this number is returned by the partitioning algorithms as implemented in this library. If the optimal solution was found many times, it is unlikely that better solutions exist than the one that was found. However, if the optimal solution was found only once, there may well be other solutions with a smaller within-cluster sum of distances. If the number of items is large (more than several hundreds), it may be difficult to find the globally optimal solution.

The EM algorithm terminates when no further reassignments take place. We noticed that for some sets of initial cluster assignments, the EM algorithm fails to converge due to the same clustering solution reappearing periodically after a small number of iteration steps. We therefore check for the occurrence of such periodic solutions during the iteration. After a given number of iteration steps, the current clustering result is saved as a reference. By comparing the clustering result after each subsequent iteration step to the reference state, we can determine if a previously encountered clustering result is found. In such a case, the iteration is halted. If after a given number of iterations the reference state has not yet been encountered, the current clustering solution is saved to be used as the new reference state. Initially, ten iteration steps are executed before resaving the reference state. This number of iteration steps is doubled each time, to ensure that periodic behavior with longer periods can also be detected.

19.16 k-means and k-medians

The k-means and k-medians algorithms are implemented as the function `kcluster` in `Bio.Cluster`:

```
>>> from Bio.Cluster import kcluster
>>> clusterid, error, nfound = kcluster(data)
```

where the following arguments are defined:

- `data` (required)
  Array containing the data for the items.
- `nclusters` (default: `2`)
  The number of clusters k.
- `mask` (default: `None`)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.
- `weight` (default: `None`)
  The weights to be used when calculating distances. If `weight` is `None`, then equal weights are assumed.
- `transpose` (default: `0`)
  Determines if rows (`transpose` is `0`) or columns (`transpose` is `1`) are to be clustered.
- `npass` (default: `1`)
  The number of times the k-means/-medians clustering algorithm is performed, each time with a different (random) initial condition. If `initialid` is given, the value of `npass` is ignored and the clustering algorithm is run only once, as it behaves deterministically in that case.
- `method` (default: `a`)
  describes how the center of a cluster is found:
  - `method=='a'`: arithmetic mean (k-means clustering);
  - `method=='m'`: median (k-medians clustering).

  For other values of `method`, the arithmetic mean is used.
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see *Distance functions*). Whereas all eight distance measures are accepted by `kcluster`, from a theoretical viewpoint it is best to use the Euclidean distance for the k-means algorithm, and the city-block distance for k-medians.
- `initialid` (default: `None`)
  Specifies the initial clustering to be used for the EM algorithm. If `initialid` is `None`, then a different random initial clustering is used for each of the `npass` runs of the EM algorithm. If `initialid` is not `None`, then it

should be equal to a 1D array containing the cluster number (between `0` and `nclusters-1`) for each item. Each cluster should contain at least one item. With the initial clustering specified, the EM algorithm is deterministic.

This function returns a tuple `(clusterid, error, nfound)`, where `clusterid` is an integer array containing the number of the cluster to which each row or cluster was assigned, `error` is the within-cluster sum of distances for the optimal clustering solution, and `nfound` is the number of times this optimal solution was found.

19.17 k-medoids clustering

The `kmedoids` routine performs k-medoids clustering on a given set of items, using the distance matrix and the number of clusters passed by the user:

```
>>> from Bio.Cluster import kmedoids
>>> clusterid, error, nfound = kmedoids(distance)
```

where the following arguments are defined: , nclusters=2, npass=1, initialid=None)|

- `distance` (required)
  The matrix containing the distances between the items; this matrix can be specified in three ways:
  - as a 2D Numerical Python array (in which only the left-lower part of the array will be accessed):

    ```
    distance = array([[0.0, 1.1, 2.3], [1.1, 0.0, 4.5], [2.3, 4.5, 0.0]])
    ```

  - as a 1D Numerical Python array containing consecutively the distances in the left-lower part of the distance matrix:

    ```
    distance = array([1.1, 2.3, 4.5])
    ```

  - as a list containing the rows of the left-lower part of the distance matrix:

    ```
    distance = [array([]), array([1.1]), array([2.3, 4.5])]
    ```

  These three expressions correspond to the same distance matrix.
- `nclusters` (default: `2`)
  The number of clusters k.
- `npass` (default: `1`)
  The number of times the k-medoids clustering algorithm is performed, each time with a different (random) initial condition. If `initialid` is given, the value of `npass` is ignored, as the clustering algorithm behaves deterministically in that case.
- `initialid` (default: `None`)
  Specifies the initial clustering to be used for the EM algorithm. If `initialid` is `None`, then a different random initial clustering is used for each of the `npass` runs of the EM algorithm. If `initialid` is not `None`, then it should be equal to a 1D array containing the cluster number (between `0` and `nclusters-1`) for each item. Each cluster should contain at least one item. With the initial clustering specified, the EM algorithm is deterministic.

This function returns a tuple `(clusterid, error, nfound)`, where `clusterid` is an array containing the number of the cluster to which each item was assigned, `error` is the within-cluster sum of distances for the optimal k-medoids clustering solution, and `nfound` is the number of times the optimal solution was found. Note that the cluster number in `clusterid` is defined as the item number of the item representing the cluster centroid.

19.17.1 Hierarchical clustering

Hierarchical clustering methods are inherently different from the k-means clustering method. In hierarchical clustering, the similarity in the expression profile between genes or experimental conditions are represented in the form of a tree structure. This tree structure can be shown graphically by programs such as Treeview and Java Treeview, which has contributed to the popularity of hierarchical clustering in the analysis of gene expression data.

The first step in hierarchical clustering is to calculate the distance matrix, specifying all the distances between the items to be clustered. Next, we create a node by joining the two closest items. Subsequent nodes are created by pairwise joining of items or nodes based on the distance between them, until all items belong to the same node. A tree structure can then be created by retracing which items and nodes were merged. Unlike the EM algorithm, which is used in k-means clustering, the complete process of hierarchical clustering is deterministic.

Several flavors of hierarchical clustering exist, which differ in how the distance between subnodes is defined in terms of their members. In `Bio.Cluster`, pairwise single, maximum, average, and centroid linkage are available.

- In pairwise single-linkage clustering, the distance between two nodes is defined as the shortest distance among the pairwise distances between the members of the two nodes.
- In pairwise maximum-linkage clustering, alternatively known as pairwise complete-linkage clustering, the distance between two nodes is defined as the longest distance among the pairwise distances between the members of the two nodes.
- In pairwise average-linkage clustering, the distance between two nodes is defined as the average over all pairwise distances between the items of the two nodes.
- In pairwise centroid-linkage clustering, the distance between two nodes is defined as the distance between their centroids. The centroids are calculated by taking the mean over all the items in a cluster. As the distance from each newly formed node to existing nodes and items need to be calculated at each step, the computing time of pairwise centroid-linkage clustering may be significantly longer than for the other hierarchical clustering methods. Another peculiarity is that (for a distance measure based on the Pearson correlation), the distances do not necessarily increase when going up in the clustering tree, and may even decrease. This is caused by an inconsistency between the centroid calculation and the distance calculation when using the Pearson correlation: Whereas the Pearson correlation effectively normalizes the data for the distance calculation, no such normalization occurs for the centroid calculation.

For pairwise single-, complete-, and average-linkage clustering, the distance between two nodes can be found directly from the distances between the individual items. Therefore, the clustering algorithm does not need access to the original gene expression data, once the distance matrix is known. For pairwise centroid-linkage clustering, however, the centroids of newly formed subnodes can only be calculated from the original data and not from the distance matrix.

The implementation of pairwise single-linkage hierarchical clustering is based on the SLINK algorithm [Sibson1973], which is much faster and more memory-efficient than a straightforward implementation of pairwise single-linkage clustering. The clustering result produced by this algorithm is identical to the clustering solution found by the conventional single-linkage algorithm. The single-linkage hierarchical clustering algorithm implemented in this library can be used to cluster large gene expression data sets, for which conventional hierarchical clustering algorithms fail due to excessive memory requirements and running time.

19.18 Representing a hierarchical clustering solution

The result of hierarchical clustering consists of a tree of nodes, in which each node joins two items or subnodes. Usually, we are not only interested in which items or subnodes are joined at each node, but also in their similarity (or distance) as they are joined. To store one node in the hierarchical clustering tree, we make use of the class `Node`, which defined in `Bio.Cluster`. An instance of `Node` has three attributes:

- `left`
- `right`
- `distance`

Here, `left` and `right` are integers referring to the two items or subnodes that are joined at this node, and `distance` is the distance between them. The items being clustered are numbered from 0 to (number of items $-$ 1), while clusters are numbered from -1 to $-$ (number of items $-$ 1). Note that the number of nodes is one less than the number of items.

To create a new `Node` object, we need to specify `left` and `right`; `distance` is optional.

```
>>> from Bio.Cluster import Node
>>> Node(2, 3)
(2, 3): 0
>>> Node(2, 3, 0.91)
(2, 3): 0.91
```

The attributes `left`, `right`, and `distance` of an existing `Node` object can be modified directly:

```
>>> node = Node(4, 5)
>>> node.left = 6
>>> node.right = 2
>>> node.distance = 0.73
>>> node
(6, 2): 0.73
```

An error is raised if `left` and `right` are not integers, or if `distance` cannot be converted to a floating-point value.

The Python class `Tree` represents a full hierarchical clustering solution. A `Tree` object can be created from a list of `Node` objects:

```
>>> from Bio.Cluster import Node, Tree
>>> nodes = [Node(1, 2, 0.2), Node(0, 3, 0.5), Node(-2, 4, 0.6), Node(-1, -3, 0.9)]
>>> tree = Tree(nodes)
>>> print(tree)
(1, 2): 0.2
(0, 3): 0.5
(-2, 4): 0.6
(-1, -3): 0.9
```

The `Tree` initializer checks if the list of nodes is a valid hierarchical clustering result:

```
>>> nodes = [Node(1, 2, 0.2), Node(0, 2, 0.5)]
>>> Tree(nodes)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Inconsistent tree
```

Individual nodes in a `Tree` object can be accessed using square brackets:

```
>>> nodes = [Node(1, 2, 0.2), Node(0, -1, 0.5)]
>>> tree = Tree(nodes)
>>> tree[0]
(1, 2): 0.2
>>> tree[1]
(0, -1): 0.5
>>> tree[-1]
(0, -1): 0.5
```

As a `Tree` object is immutable, we cannot change individual nodes in a `Tree` object. However, we can convert the tree to a list of nodes, modify this list, and create a new tree from this list:

```
>>> tree = Tree([Node(1, 2, 0.1), Node(0, -1, 0.5), Node(-2, 3, 0.9)])
>>> print(tree)
(1, 2): 0.1
(0, -1): 0.5
(-2, 3): 0.9
>>> nodes = tree[:]
>>> nodes[0] = Node(0, 1, 0.2)
>>> nodes[1].left = 2
>>> tree = Tree(nodes)
>>> print(tree)
(0, 1): 0.2
(2, -1): 0.5
(-2, 3): 0.9
```

This guarantees that any `Tree` object is always well-formed.

To display a hierarchical clustering solution with visualization programs such as Java Treeview, it is better to scale all node distances such that they are between zero and one. This can be accomplished by calling the `scale` method on an existing `Tree` object:

```
>>> tree.scale()
```

This method takes no arguments, and returns `None`.

Before drawing the tree, you may also want to reorder the tree nodes. A hierarchical clustering solution of n items can be drawn as 2^{n-1} different but equivalent dendrograms by switching the left and right subnode at each node. The `tree.sort(order)` method visits each node in the hierarchical clustering tree and verifies if the average order value of the left subnode is less than or equal to the average order value of the right subnode. If not, the left and right subnodes are exchanged. Here, the order values of the items are given by the user. In the resulting dendrogram, items in the left-to-right order will tend to have increasing order values. The method will return the indices of the elements in the left-to-right order after sorting:

```
>>> indices = tree.sort(order)
```

such that item `indices[i]` will occur at position i in the dendrogram.

After hierarchical clustering, the items can be grouped into k clusters based on the tree structure stored in the `Tree` object by cutting the tree:

```
>>> clusterid = tree.cut(nclusters=1)
```

where `nclusters` (defaulting to 1) is the desired number of clusters k. This method ignores the top $k-1$ linking events in the tree structure, resulting in k separated clusters of items. The number of clusters k should be positive,

and less than or equal to the number of items. This method returns an array `clusterid` containing the number of the cluster to which each item is assigned. Clusters are numbered 0 to $k-1$ in their left-to-right order in the dendrogram.

19.19 Performing hierarchical clustering

To perform hierarchical clustering, use the `treecluster` function in `Bio.Cluster`.

```
>>> from Bio.Cluster import treecluster
>>> tree = treecluster(data)
```

where the following arguments are defined:

- `data`
  Array containing the data for the items.
- `mask` (default: `None`)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.
- `weight` (default: `None`)
  The weights to be used when calculating distances. If `weight` is `None`, then equal weights are assumed.
- `transpose` (default: `0`)
  Determines if rows (`transpose` is `False`) or columns (`transpose` is `True`) are to be clustered.
- `method` (default: `'m'`)
  defines the linkage method to be used:
  - `method=='s'`: pairwise single-linkage clustering
  - `method=='m'`: pairwise maximum- (or complete-) linkage clustering
  - `method=='c'`: pairwise centroid-linkage clustering
  - `method=='a'`: pairwise average-linkage clustering
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see *Distance functions*).

To apply hierarchical clustering on a precalculated distance matrix, specify the `distancematrix` argument when calling `treecluster` function instead of the `data` argument:

```
>>> from Bio.Cluster import treecluster
>>> tree = treecluster(distancematrix=distance)
```

In this case, the following arguments are defined:

- `distancematrix`
  The distance matrix, which can be specified in three ways:
  - as a 2D Numerical Python array (in which only the left-lower part of the array will be accessed):

    ```
    distance = array([[0.0, 1.1, 2.3], [1.1, 0.0, 4.5], [2.3, 4.5, 0.0]])
    ```

  - as a 1D Numerical Python array containing consecutively the distances in the left-lower part of the distance matrix:

```
distance = array([1.1, 2.3, 4.5])
```

  - as a list containing the rows of the left-lower part of the distance matrix:

```
distance = [array([]), array([1.1]), array([2.3, 4.5])]
```

  These three expressions correspond to the same distance matrix. As `treecluster` may shuffle the values in the distance matrix as part of the clustering algorithm, be sure to save this array in a different variable before calling `treecluster` if you need it later.

- `method`
  The linkage method to be used:
  - `method=='s'`: pairwise single-linkage clustering
  - `method=='m'`: pairwise maximum- (or complete-) linkage clustering
  - `method=='a'`: pairwise average-linkage clustering

  While pairwise single-, maximum-, and average-linkage clustering can be calculated from the distance matrix alone, pairwise centroid-linkage cannot.

When calling `treecluster`, either `data` or `distancematrix` should be `None`.

This function returns a `Tree` object. This object contains $(\text{number of items} - 1)$ nodes, where the number of items is the number of rows if rows were clustered, or the number of columns if columns were clustered. Each node describes a pairwise linking event, where the node attributes `left` and `right` each contain the number of one item or subnode, and `distance` the distance between them. Items are numbered from 0 to $(\text{number of items} - 1)$, while clusters are numbered -1 to $-(\text{number of items} - 1)$.

19.19.1 Self-Organizing Maps

Self-Organizing Maps (SOMs) were invented by Kohonen to describe neural networks (see for instance Kohonen, 1997 [Kohonen1997]). Tamayo (1999) first applied Self-Organizing Maps to gene expression data [Tamayo1999].

SOMs organize items into clusters that are situated in some topology. Usually a rectangular topology is chosen. The clusters generated by SOMs are such that neighboring clusters in the topology are more similar to each other than clusters far from each other in the topology.

The first step to calculate a SOM is to randomly assign a data vector to each cluster in the topology. If rows are being clustered, then the number of elements in each data vector is equal to the number of columns.

An SOM is then generated by taking rows one at a time, and finding which cluster in the topology has the closest data vector. The data vector of that cluster, as well as those of the neighboring clusters, are adjusted using the data vector of the row under consideration. The adjustment is given by

$$\Delta \underline{x}_{\text{cell}} = \tau \cdot (\underline{x}_{\text{row}} - \underline{x}_{\text{cell}}) \,.$$

The parameter τ is a parameter that decreases at each iteration step. We have used a simple linear function of the iteration step:

$$\tau = \tau_{\text{init}} \cdot \left(1 - \frac{i}{n}\right),$$

τ_{init} is the initial value of τ as specified by the user, i is the number of the current iteration step, and n is the total number of iteration steps to be performed. While changes are made rapidly in the beginning of the iteration, at the end of iteration only small changes are made.

All clusters within a radius R are adjusted to the gene under consideration. This radius decreases as the calculation progresses as

$$R = R_{\max} \cdot \left(1 - \frac{i}{n}\right),$$

in which the maximum radius is defined as

$$R_{\max} = \sqrt{N_x^2 + N_y^2},$$

where (N_x, N_y) are the dimensions of the rectangle defining the topology.

The function `somcluster` implements the complete algorithm to calculate a Self-Organizing Map on a rectangular grid. First it initializes the random number generator. The node data are then initialized using the random number generator. The order in which genes or samples are used to modify the SOM is also randomized. The total number of iterations in the SOM algorithm is specified by the user.

To run `somcluster`, use

```
>>> from Bio.Cluster import somcluster
>>> clusterid, celldata = somcluster(data)
```

where the following arguments are defined:

- `data` (required)
  Array containing the data for the items.
- `mask` (default: `None`)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.
- `weight` (default: `None`)
  contains the weights to be used when calculating distances. If `weight` is `None`, then equal weights are assumed.
- `transpose` (default: `0`)
  Determines if rows (`transpose` is `0`) or columns (`transpose` is 1) are to be clustered.
- `nxgrid, nygrid` (default: `2, 1`)
  The number of cells horizontally and vertically in the rectangular grid on which the Self-Organizing Map is calculated.
- `inittau` (default: `0.02`)
  The initial value for the parameter τ that is used in the SOM algorithm. The default value for `inittau` is 0.02, which was used in Michael Eisen's Cluster/TreeView program.
- `niter` (default: `1`)
  The number of iterations to be performed.
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see *Distance functions*).

This function returns the tuple `(clusterid, celldata)`:

- `clusterid`:
  An array with two columns, where the number of rows is equal to the number of items that were clustered. Each row contains the x and y coordinates of the cell in the rectangular SOM grid to which the item was assigned.
- `celldata`:

An array with dimensions (`nxgrid`, `nygrid`, number of columns) if rows are being clustered, or (`nxgrid`, `nygrid`, number of rows) if columns are being clustered. Each element `[ix][iy]` of this array is a 1D vector containing the gene expression data for the centroid of the cluster in the grid cell with coordinates `[ix][iy]`.

19.19.2 Principal Component Analysis

Principal Component Analysis (PCA) is a widely used technique for analyzing multivariate data. A practical example of applying Principal Component Analysis to gene expression data is presented by Yeung and Ruzzo (2001) [Yeung2001].

In essence, PCA is a coordinate transformation in which each row in the data matrix is written as a linear sum over basis vectors called principal components, which are ordered and chosen such that each maximally explains the remaining variance in the data vectors. For example, an $n \times 3$ data matrix can be represented as an ellipsoidal cloud of n points in three dimensional space. The first principal component is the longest axis of the ellipsoid, the second principal component the second longest axis of the ellipsoid, and the third principal component is the shortest axis. Each row in the data matrix can be reconstructed as a suitable linear combination of the principal components. However, in order to reduce the dimensionality of the data, usually only the most important principal components are retained. The remaining variance present in the data is then regarded as unexplained variance.

The principal components can be found by calculating the eigenvectors of the covariance matrix of the data. The corresponding eigenvalues determine how much of the variance present in the data is explained by each principal component.

Before applying principal component analysis, typically the mean is subtracted from each column in the data matrix. In the example above, this effectively centers the ellipsoidal cloud around its centroid in 3D space, with the principal components describing the variation of points in the ellipsoidal cloud with respect to their centroid.

The function `pca` below first uses the singular value decomposition to calculate the eigenvalues and eigenvectors of the data matrix. The singular value decomposition is implemented as a translation in C of the Algol procedure `svd` [Golub1971], which uses Householder bidiagonalization and a variant of the QR algorithm. The principal components, the coordinates of each data vector along the principal components, and the eigenvalues corresponding to the principal components are then evaluated and returned in decreasing order of the magnitude of the eigenvalue. If data centering is desired, the mean should be subtracted from each column in the data matrix before calling the `pca` routine.

To apply Principal Component Analysis to a rectangular matrix `data`, use

```
>>> from Bio.Cluster import pca
>>> columnmean, coordinates, components, eigenvalues = pca(data)
```

This function returns a tuple `columnmean, coordinates, components, eigenvalues`:

- `columnmean`
  Array containing the mean over each column in `data`.
- `coordinates`
  The coordinates of each row in `data` with respect to the principal components.
- `components`
  The principal components.
- `eigenvalues`
  The eigenvalues corresponding to each of the principal components.

The original matrix `data` can be recreated by calculating `columnmean + dot(coordinates, components)`.

19.19.3 Handling Cluster/TreeView-type files

Cluster/TreeView are GUI-based codes for clustering gene expression data. They were originally written by Michael Eisen while at Stanford University [Eisen1998]. `Bio.Cluster` contains functions for reading and writing data files that correspond to the format specified for Cluster/TreeView. In particular, by saving a clustering result in that format, TreeView can be used to visualize the clustering results. We recommend using Alok Saldanha's http://jtreeview.sourceforge.net/Java TreeView program [Saldanha2004], which can display hierarchical as well as k-means clustering results.

An object of the class `Record` contains all information stored in a Cluster/TreeView-type data file. To store the information contained in the data file in a `Record` object, we first open the file and then read it:

```
>>> from Bio import Cluster
>>> with open("mydatafile.txt") as handle:
...     record = Cluster.read(handle)
...
```

This two-step process gives you some flexibility in the source of the data. For example, you can use

```
>>> import gzip  # Python standard library
>>> handle = gzip.open("mydatafile.txt.gz", "rt")
```

to open a gzipped file, or

```
>>> from urllib.request import urlopen
>>> from io import TextIOWrapper
>>> url = "https://raw.githubusercontent.com/biopython/biopython/master/Tests/Cluster/
↪cyano.txt"
>>> handle = TextIOWrapper(urlopen(url))
```

to open a file stored on the Internet before calling `read`.

The `read` command reads the tab-delimited text file `mydatafile.txt` containing gene expression data in the format specified for Michael Eisen's Cluster/TreeView program. In this file format, rows represent genes and columns represent samples or observations. For a simple time course, a minimal input file would look like this:

YORF	0 minutes	30 minutes	1 hour	2 hours	4 hours
YAL001C	1	1.3	2.4	5.8	2.4
YAL002W	0.9	0.8	0.7	0.5	0.2
YAL003W	0.8	2.1	4.2	10.1	10.1
YAL005C	1.1	1.3	0.8		0.4
YAL010C	1.2	1	1.1	4.5	8.3

Each row (gene) has an identifier that always goes in the first column. In this example, we are using yeast open reading frame codes. Each column (sample) has a label in the first row. In this example, the labels describe the time at which a sample was taken. The first column of the first row contains a special field that tells the program what kind of objects are in each row. In this case, YORF stands for yeast open reading frame. This field can be any alphanumeric value. The remaining cells in the table contain data for the appropriate gene and sample. The 5.8 in row 2 column 4 means that the observed value for gene YAL001C at 2 hours was 5.8. Missing values are acceptable and are designated by empty cells (e.g. YAL004C at 2 hours).

The input file may contain additional information. A maximal input file would look like this:

YORF	NAME	GWEIGHT	GORDER	0	30	1	2	4
EWEIGHT				1	1	1	1	0
EORDER				5	3	2	1	1
YAL001C	TFIIIC 138 KD SUBUNIT	1	1	1	1.3	2.4	5.8	2.4
YAL002W	UNKNOWN	0.4	3	0.9	0.8	0.7	0.5	0.2
YAL003W	ELONGATION FACTOR EF1-BETA	0.4	2	0.8	2.1	4.2	10.1	10.1
YAL005C	CYTOSOLIC HSP70	0.4	5	1.1	1.3	0.8		0.4

The added columns NAME, GWEIGHT, and GORDER and rows EWEIGHT and EORDER are optional. The NAME column allows you to specify a label for each gene that is distinct from the ID in column 1.

A `Record` object has the following attributes:

- `data`
  The data array containing the gene expression data. Genes are stored row-wise, while samples are stored column-wise.
- `mask`
  This array shows which elements in the `data` array, if any, are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If no data were found to be missing, `mask` is set to `None`.
- `geneid`
  This is a list containing a unique description for each gene (i.e., ORF numbers).
- `genename`
  This is a list containing a description for each gene (i.e., gene name). If not present in the data file, `genename` is set to `None`.
- `gweight`
  The weights that are to be used to calculate the distance in expression profile between genes. If not present in the data file, `gweight` is set to `None`.
- `gorder`
  The preferred order in which genes should be stored in an output file. If not present in the data file, `gorder` is set to `None`.
- `expid`
  This is a list containing a description of each sample, e.g. experimental condition.
- `eweight`
  The weights that are to be used to calculate the distance in expression profile between samples. If not present in the data file, `eweight` is set to `None`.
- `eorder`
  The preferred order in which samples should be stored in an output file. If not present in the data file, `eorder` is set to `None`.
- `uniqid`
  The string that was used instead of UNIQID in the data file.

After loading a `Record` object, each of these attributes can be accessed and modified directly. For example, the data can be log-transformed by taking the logarithm of `record.data`.

19.20 Calculating the distance matrix

To calculate the distance matrix between the items stored in the record, use

```
>>> matrix = record.distancematrix()
```

where the following arguments are defined:

- `transpose` (default: `0`)
  Determines if the distances between the rows of `data` are to be calculated (`transpose` is `False`), or between the columns of `data` (`transpose` is `True`).
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see *Distance functions*).

This function returns the distance matrix as a list of rows, where the number of columns of each row is equal to the row number (see section *Calculating the distance matrix*).

19.21 Calculating the cluster centroids

To calculate the centroids of clusters of items stored in the record, use

```
>>> cdata, cmask = record.clustercentroids()
```

- `clusterid` (default: `None`)
  Vector of integers showing to which cluster each item belongs. If `clusterid` is not given, then all items are assumed to belong to the same cluster.
- `method` (default: `'a'`)
  Specifies whether the arithmetic mean (`method=='a'`) or the median (`method=='m'`) is used to calculate the cluster center.
- `transpose` (default: `0`)
  Determines if the centroids of the rows of `data` are to be calculated (`transpose` is `False`), or the centroids of the columns of `data` (`transpose` is `True`).

This function returns the tuple `cdata, cmask`; see section *Calculating the cluster centroids* for a description.

19.22 Calculating the distance between clusters

To calculate the distance between clusters of items stored in the record, use

```
>>> distance = record.clusterdistance()
```

where the following arguments are defined:

- `index1` (default: `0`)
  A list containing the indices of the items belonging to the first cluster. A cluster containing only one item i can be represented either as a list `[i]`, or as an integer `i`.
- `index2` (default: `0`)
  A list containing the indices of the items belonging to the second cluster. A cluster containing only one item i can be represented either as a list `[i]`, or as an integer `i`.

- `method` (default: `'a'`)
  Specifies how the distance between clusters is defined:
  - `'a'`: Distance between the two cluster centroids (arithmetic mean);
  - `'m'`: Distance between the two cluster centroids (median);
  - `'s'`: Shortest pairwise distance between items in the two clusters;
  - `'x'`: Longest pairwise distance between items in the two clusters;
  - `'v'`: Average over the pairwise distances between items in the two clusters.
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see *Distance functions*).
- `transpose` (default: `0`)
  If `transpose` is `False`, calculate the distance between the rows of `data`. If `transpose` is `True`, calculate the distance between the columns of `data`.

19.23 Performing hierarchical clustering

To perform hierarchical clustering on the items stored in the record, use

```
>>> tree = record.treecluster()
```

where the following arguments are defined:

- `transpose` (default: `0`)
  Determines if rows (`transpose` is `False`) or columns (`transpose` is `True`) are to be clustered.
- `method` (default: `'m'`)
  defines the linkage method to be used:
  - `method=='s'`: pairwise single-linkage clustering
  - `method=='m'`: pairwise maximum- (or complete-) linkage clustering
  - `method=='c'`: pairwise centroid-linkage clustering
  - `method=='a'`: pairwise average-linkage clustering
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see *Distance functions*).
- `transpose`
  Determines if genes or samples are being clustered. If `transpose` is `False`, genes (rows) are being clustered. If `transpose` is `True`, samples (columns) are clustered.

This function returns a `Tree` object. This object contains (number of items $-$ 1) nodes, where the number of items is the number of rows if rows were clustered, or the number of columns if columns were clustered. Each node describes a pairwise linking event, where the node attributes `left` and `right` each contain the number of one item or subnode, and `distance` the distance between them. Items are numbered from 0 to (number of items $-$ 1), while clusters are numbered -1 to $-$ (number of items $-$ 1).

19.24 Performing k-means or k-medians clustering

To perform k-means or k-medians clustering on the items stored in the record, use

```
>>> clusterid, error, nfound = record.kcluster()
```

where the following arguments are defined:

- `nclusters` (default: `2`)
  The number of clusters k.
- `transpose` (default: `0`)
  Determines if rows (`transpose` is `0`) or columns (`transpose` is `1`) are to be clustered.
- `npass` (default: `1`)
  The number of times the k-means/-medians clustering algorithm is performed, each time with a different (random) initial condition. If `initialid` is given, the value of `npass` is ignored and the clustering algorithm is run only once, as it behaves deterministically in that case.
- `method` (default: `a`)
  describes how the center of a cluster is found:
  - `method=='a'`: arithmetic mean (k-means clustering);
  - `method=='m'`: median (k-medians clustering).

  For other values of `method`, the arithmetic mean is used.
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see *Distance functions*).

This function returns a tuple `(clusterid, error, nfound)`, where `clusterid` is an integer array containing the number of the cluster to which each row or cluster was assigned, `error` is the within-cluster sum of distances for the optimal clustering solution, and `nfound` is the number of times this optimal solution was found.

19.25 Calculating a Self-Organizing Map

To calculate a Self-Organizing Map of the items stored in the record, use

```
>>> clusterid, celldata = record.somcluster()
```

where the following arguments are defined:

- `transpose` (default: `0`)
  Determines if rows (`transpose` is `0`) or columns (`transpose` is `1`) are to be clustered.
- `nxgrid, nygrid` (default: `2, 1`)
  The number of cells horizontally and vertically in the rectangular grid on which the Self-Organizing Map is calculated.
- `inittau` (default: `0.02`)
  The initial value for the parameter τ that is used in the SOM algorithm. The default value for `inittau` is 0.02, which was used in Michael Eisen's Cluster/TreeView program.
- `niter` (default: `1`)
  The number of iterations to be performed.
- `dist` (default: `'e'`, Euclidean distance)

Defines the distance function to be used (see *Distance functions*).

This function returns the tuple `(clusterid, celldata)`:

- `clusterid`:
  An array with two columns, where the number of rows is equal to the number of items that were clustered. Each row contains the x and y coordinates of the cell in the rectangular SOM grid to which the item was assigned.
- `celldata`:
  An array with dimensions (`nxgrid`, `nygrid`, number of columns) if rows are being clustered, or (`nxgrid`, `nygrid`, number of rows) if columns are being clustered. Each element `[ix][iy]` of this array is a 1D vector containing the gene expression data for the centroid of the cluster in the grid cell with coordinates `[ix][iy]`.

19.26 Saving the clustering result

To save the clustering result, use

```
>>> record.save(jobname, geneclusters, expclusters)
```

where the following arguments are defined:

- `jobname`
  The string `jobname` is used as the base name for names of the files that are to be saved.
- `geneclusters`
  This argument describes the gene (row-wise) clustering result. In case of k-means clustering, this is a 1D array containing the number of the cluster each gene belongs to. It can be calculated using `kcluster`. In case of hierarchical clustering, `geneclusters` is a `Tree` object.
- `expclusters`
  This argument describes the (column-wise) clustering result for the experimental conditions. In case of k-means clustering, this is a 1D array containing the number of the cluster each experimental condition belongs to. It can be calculated using `kcluster`. In case of hierarchical clustering, `expclusters` is a `Tree` object.

This method writes the text file `jobname.cdt`, `jobname.gtr`, `jobname.atr`, `jobname*.kgg`, and/or `jobname*.kag` for subsequent reading by the Java TreeView program. If `geneclusters` and `expclusters` are both `None`, this method only writes the text file `jobname.cdt`; this file can subsequently be read into a new `Record` object.

19.26.1 Example calculation

This is an example of a hierarchical clustering calculation, using single linkage clustering for genes and maximum linkage clustering for experimental conditions. As the Euclidean distance is being used for gene clustering, it is necessary to scale the node distances `genetree` such that they are all between zero and one. This is needed for the Java TreeView code to display the tree diagram correctly. To cluster the experimental conditions, the uncentered correlation is being used. No scaling is needed in this case, as the distances in `exptree` are already between zero and two.

The example data `cyano.txt` can be found in Biopython's `Tests/Cluster` subdirectory and is from the paper Hihara *et al.* 2001 [Hihara2001].

```
>>> from Bio import Cluster
>>> with open("cyano.txt") as handle:
...     record = Cluster.read(handle)
...
```

(continues on next page)

(continued from previous page)

```
>>> genetree = record.treecluster(method="s")
>>> genetree.scale()
>>> exptree = record.treecluster(dist="u", transpose=1)
>>> record.save("cyano_result", genetree, exptree)
```

This will create the files `cyano_result.cdt`, `cyano_result.gtr`, and `cyano_result.atr`.

Similarly, we can save a k-means clustering solution:

```
>>> from Bio import Cluster
>>> with open("cyano.txt") as handle:
...     record = Cluster.read(handle)
...
>>> (geneclusters, error, ifound) = record.kcluster(nclusters=5, npass=1000)
>>> (expclusters, error, ifound) = record.kcluster(nclusters=2, npass=100, transpose=1)
>>> record.save("cyano_result", geneclusters, expclusters)
```

This will create the files `cyano_result_K_G2_A2.cdt`, `cyano_result_K_G2.kgg`, and `cyano_result_K_A2.kag`.

CHAPTER

TWENTY

GRAPHICS INCLUDING GENOMEDIAGRAM

The `Bio.Graphics` module depends on the third party Python library ReportLab. Although focused on producing PDF files, ReportLab can also create encapsulated postscript (EPS) and (SVG) files. In addition to these vector based images, provided certain further dependencies such as the Python Imaging Library (PIL) are installed, ReportLab can also output bitmap images (including JPEG, PNG, GIF, BMP and PICT formats).

20.1 GenomeDiagram

20.1.1 Introduction

The `Bio.Graphics.GenomeDiagram` module was added to Biopython 1.50, having previously been available as a separate Python module dependent on Biopython. GenomeDiagram is described in the Bioinformatics journal publication by Pritchard et al. (2006) [Pritchard2006], which includes some examples images. There is a PDF copy of the old manual here, http://biopython.org/DIST/docs/GenomeDiagram/userguide.pdf which has some more examples.

As the name might suggest, GenomeDiagram was designed for drawing whole genomes, in particular prokaryotic genomes, either as linear diagrams (optionally broken up into fragments to fit better) or as circular wheel diagrams. Have a look at Figure 2 in Toth *et al.* (2006) [Toth2006] for a good example. It proved also well suited to drawing quite detailed figures for smaller genomes such as phage, plasmids or mitochondria, for example see Figures 1 and 2 in Van der Auwera *et al.* (2009) [Vanderauwera2009] (shown with additional manual editing).

This module is easiest to use if you have your genome loaded as a `SeqRecord` object containing lots of `SeqFeature` objects - for example as loaded from a GenBank file (see Chapters *Sequence annotation objects* and *Sequence Input/Output*).

20.1.2 Diagrams, tracks, feature-sets and features

GenomeDiagram uses a nested set of objects. At the top level, you have a diagram object representing a sequence (or sequence region) along the horizontal axis (or circle). A diagram can contain one or more tracks, shown stacked vertically (or radially on circular diagrams). These will typically all have the same length and represent the same sequence region. You might use one track to show the gene locations, another to show regulatory regions, and a third track to show the GC percentage.

The most commonly used type of track will contain features, bundled together in feature-sets. You might choose to use one feature-set for all your CDS features, and another for tRNA features. This isn't required - they can all go in the same feature-set, but it makes it easier to update the properties of just selected features (e.g. make all the tRNA features red).

There are two main ways to build up a complete diagram. Firstly, the top down approach where you create a diagram object, and then using its methods add track(s), and use the track methods to add feature-set(s), and use their methods

to add the features. Secondly, you can create the individual objects separately (in whatever order suits your code), and then combine them.

20.1.3 A top down example

We're going to draw a whole genome from a `SeqRecord` object read in from a GenBank file (see Chapter *Sequence Input/Output*). This example uses the pPCP1 plasmid from *Yersinia pestis biovar Microtus*, the file is included with the Biopython unit tests under the GenBank folder, or online NC_005816.gb from our website.

```
from reportlab.lib import colors
from reportlab.lib.units import cm
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO

record = SeqIO.read("NC_005816.gb", "genbank")
```

We're using a top down approach, so after loading in our sequence we next create an empty diagram, then add an (empty) track, and to that add an (empty) feature set:

```
gd_diagram = GenomeDiagram.Diagram("Yersinia pestis biovar Microtus plasmid pPCP1")
gd_track_for_features = gd_diagram.new_track(1, name="Annotated Features")
gd_feature_set = gd_track_for_features.new_set()
```

Now the fun part - we take each gene `SeqFeature` object in our `SeqRecord`, and use it to generate a feature on the diagram. We're going to color them blue, alternating between a dark blue and a light blue.

```
for feature in record.features:
    if feature.type != "gene":
        # Exclude this feature
        continue
    if len(gd_feature_set) % 2 == 0:
        color = colors.blue
    else:
        color = colors.lightblue
    gd_feature_set.add_feature(feature, color=color, label=True)
```

Now we come to actually making the output file. This happens in two steps, first we call the `draw` method, which creates all the shapes using ReportLab objects. Then we call the `write` method which renders these to the requested file format. Note you can output in multiple file formats:

```
gd_diagram.draw(
    format="linear",
    orientation="landscape",
    pagesize="A4",
    fragments=4,
    start=0,
    end=len(record),
)
gd_diagram.write("plasmid_linear.pdf", "PDF")
gd_diagram.write("plasmid_linear.eps", "EPS")
gd_diagram.write("plasmid_linear.svg", "SVG")
```

Also, provided you have the dependencies installed, you can also do bitmaps, for example:

```
gd_diagram.write("plasmid_linear.png", "PNG")
```

The expected output is shown in Fig. 1.

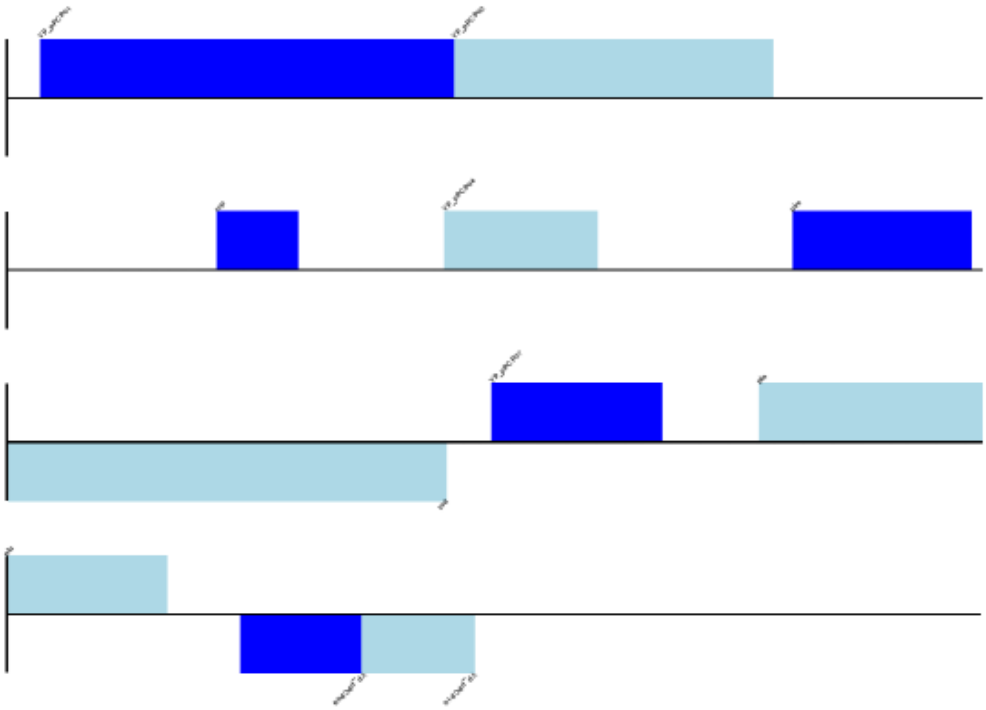


Fig. 1: Simple linear diagram for *Y. pestis biovar Microtus* plasmid pPCP1.

Notice that the `fragments` argument which we set to four controls how many pieces the genome gets broken up into.

If you want to do a circular figure, then try this:

```
gd_diagram.draw(
    format="circular",
    circular=True,
    pagesize=(20 * cm, 20 * cm),
    start=0,
    end=len(record),
    circle_core=0.7,
)
gd_diagram.write("plasmid_circular.pdf", "PDF")
```

The expected output is shown in Fig. 2.

These figures are not very exciting, but we've only just got started.

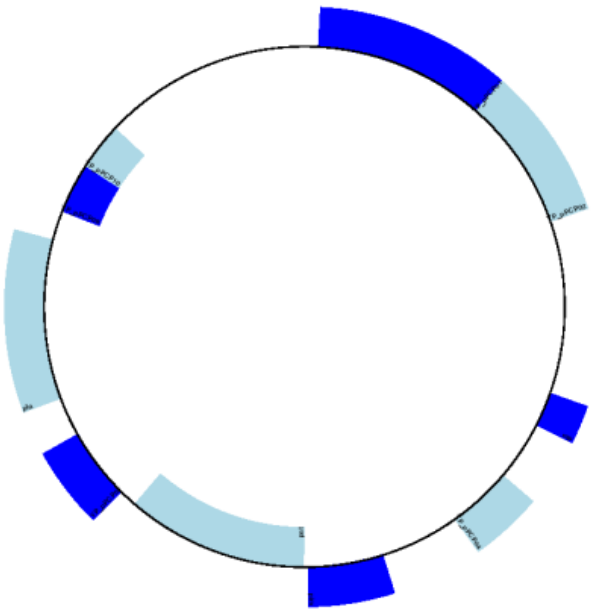


Fig. 2: Simple circular diagram for *Y. pestis biovar Microtus* plasmid pPCP1.

20.1.4 A bottom up example

Now let's produce exactly the same figures, but using the bottom up approach. This means we create the different objects directly (and this can be done in almost any order) and then combine them.

```
from reportlab.lib import colors
from reportlab.lib.units import cm
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO

record = SeqIO.read("NC_005816.gb", "genbank")

# Create the feature set and its feature objects,
gd_feature_set = GenomeDiagram.FeatureSet()
for feature in record.features:
    if feature.type != "gene":
        # Exclude this feature
        continue
    if len(gd_feature_set) % 2 == 0:
        color = colors.blue
    else:
        color = colors.lightblue
    gd_feature_set.add_feature(feature, color=color, label=True)
# (this for loop is the same as in the previous example)

# Create a track, and a diagram
gd_track_for_features = GenomeDiagram.Track(name="Annotated Features")
gd_diagram = GenomeDiagram.Diagram("Yersinia pestis biovar Microtus plasmid pPCP1")

# Now have to glue the bits together...
gd_track_for_features.add_set(gd_feature_set)
```

(continues on next page)

(continued from previous page)

```
gd_diagram.add_track(gd_track_for_features, 1)
```

You can now call the `draw` and `write` methods as before to produce a linear or circular diagram, using the code at the end of the top-down example above. The figures should be identical.

20.1.5 Features without a SeqFeature

In the above example we used a `SeqRecord`'s `SeqFeature` objects to build our diagram (see also Section *Feature, location and position objects*). Sometimes you won't have `SeqFeature` objects, but just the coordinates for a feature you want to draw. You have to create minimal `SeqFeature` object, but this is easy:

```
from Bio.SeqFeature import SeqFeature, SimpleLocation

my_seq_feature = SeqFeature(SimpleLocation(50, 100, strand=+1))
```

For strand, use +1 for the forward strand, -1 for the reverse strand, and `None` for both. Here is a short self contained example:

```
from Bio.SeqFeature import SeqFeature, SimpleLocation
from Bio.Graphics import GenomeDiagram
from reportlab.lib.units import cm

gdd = GenomeDiagram.Diagram("Test Diagram")
gdt_features = gdd.new_track(1, greytrack=False)
gds_features = gdt_features.new_set()

# Add three features to show the strand options,
feature = SeqFeature(SimpleLocation(25, 125, strand=+1))
gds_features.add_feature(feature, name="Forward", label=True)
feature = SeqFeature(SimpleLocation(150, 250, strand=None))
gds_features.add_feature(feature, name="Strandless", label=True)
feature = SeqFeature(SimpleLocation(275, 375, strand=-1))
gds_features.add_feature(feature, name="Reverse", label=True)

gdd.draw(format="linear", pagesize=(15 * cm, 4 * cm), fragments=1, start=0, end=400)
gdd.write("GD_labels_default.pdf", "pdf")
```

The output is shown at the top of Fig. 3 (in the default feature color, pale green).

Notice that we have used the `name` argument here to specify the caption text for these features. This is discussed in more detail next.

20.1.6 Feature captions

Recall we used the following (where `feature` was a `SeqFeature` object) to add a feature to the diagram:

```
gd_feature_set.add_feature(feature, color=color, label=True)
```

In the example above the `SeqFeature` annotation was used to pick a sensible caption for the features. By default the following possible entries under the `SeqFeature` object's qualifiers dictionary are used: `gene`, `label`, `name`, `locus_tag`, and `product`. More simply, you can specify a name directly:

```
gd_feature_set.add_feature(feature, color=color, label=True, name="My Gene")
```

In addition to the caption text for each feature's label, you can also choose the font, position (this defaults to the start of the sigil, you can also choose the middle or at the end) and orientation (for linear diagrams only, where this defaults to rotated by 45 degrees):

```
# Large font, parallel with the track
gd_feature_set.add_feature(
    feature, label=True, color="green", label_size=25, label_angle=0
)

# Very small font, perpendicular to the track (towards it)
gd_feature_set.add_feature(
    feature,
    label=True,
    color="purple",
    label_position="end",
    label_size=4,
    label_angle=90,
)

# Small font, perpendicular to the track (away from it)
gd_feature_set.add_feature(
    feature,
    label=True,
    color="blue",
    label_position="middle",
    label_size=6,
    label_angle=-90,
)
```

Combining each of these three fragments with the complete example in the previous section should give something like the tracks in Fig. 3.

We've not shown it here, but you can also set `label_color` to control the label's color (used in Section *A nice example*).

You'll notice the default font is quite small - this makes sense because you will usually be drawing many (small) features on a page, not just a few large ones as shown here.

20.1.7 Feature sigils

The examples above have all just used the default sigil for the feature, a plain box, which was all that was available in the last publicly released standalone version of GenomeDiagram. Arrow sigils were included when GenomeDiagram was added to Biopython 1.50:

```
# Default uses a BOX sigil
gd_feature_set.add_feature(feature)

# You can make this explicit:
gd_feature_set.add_feature(feature, sigil="BOX")

# Or opt for an arrow:
gd_feature_set.add_feature(feature, sigil="ARROW")
```

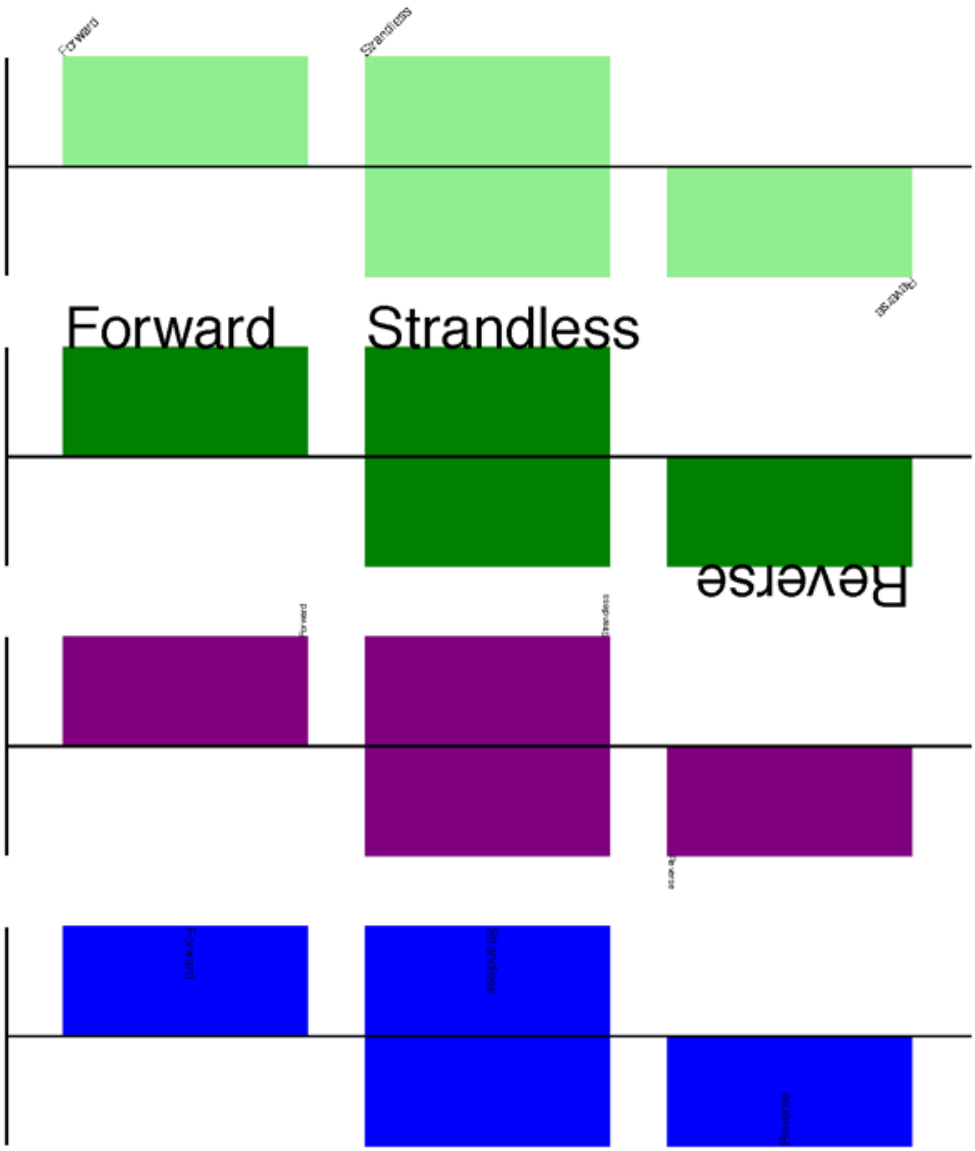


Fig. 3: Simple GenomeDiagram showing label options.
The top plot in pale green shows the default label settings (see Section *Features without a SeqFeature*) while the rest show variations in the label size, position and orientation (see Section *Feature captions*).

Biopython 1.61 added three more sigils,

```
# Box with corners cut off (making it an octagon)
gd_feature_set.add_feature(feature, sigil="OCTO")

# Box with jagged edges (useful for showing breaks in contains)
gd_feature_set.add_feature(feature, sigil="JAGGY")

# Arrow which spans the axis with strand used only for direction
gd_feature_set.add_feature(feature, sigil="BIGARROW")
```

These are shown in Fig. 4. Most sigils fit into a bounding box (as given by the default BOX sigil), either above or below the axis for the forward or reverse strand, or straddling it (double the height) for strand-less features. The BIGARROW sigil is different, always straddling the axis with the direction taken from the feature's stand.

20.1.8 Arrow sigils

We introduced the arrow sigils in the previous section. There are two additional options to adjust the shapes of the arrows, firstly the thickness of the arrow shaft, given as a proportion of the height of the bounding box:

```
# Full height shafts, giving pointed boxes:
gd_feature_set.add_feature(feature, sigil="ARROW", color="brown", arrowshaft_height=1.0)
# Or, thin shafts:
gd_feature_set.add_feature(feature, sigil="ARROW", color="teal", arrowshaft_height=0.2)
# Or, very thin shafts:
gd_feature_set.add_feature(
    feature, sigil="ARROW", color="darkgreen", arrowshaft_height=0.1
)
```

The results are shown in Fig. 5.

Secondly, the length of the arrow head - given as a proportion of the height of the bounding box (defaulting to 0.5, or 50%):

```
# Short arrow heads:
gd_feature_set.add_feature(feature, sigil="ARROW", color="blue", arrowhead_length=0.25)
# Or, longer arrow heads:
gd_feature_set.add_feature(feature, sigil="ARROW", color="orange", arrowhead_length=1)
# Or, very very long arrow heads (i.e. all head, no shaft, so triangles):
gd_feature_set.add_feature(feature, sigil="ARROW", color="red", arrowhead_length=10000)
```

The results are shown in Fig. 6.

Biopython 1.61 adds a new `BIGARROW` sigil which always straddles the axis, pointing left for the reverse strand or right otherwise:

```
# A large arrow straddling the axis:
gd_feature_set.add_feature(feature, sigil="BIGARROW")
```

All the shaft and arrow head options shown above for the `ARROW` sigil can be used for the `BIGARROW` sigil too.

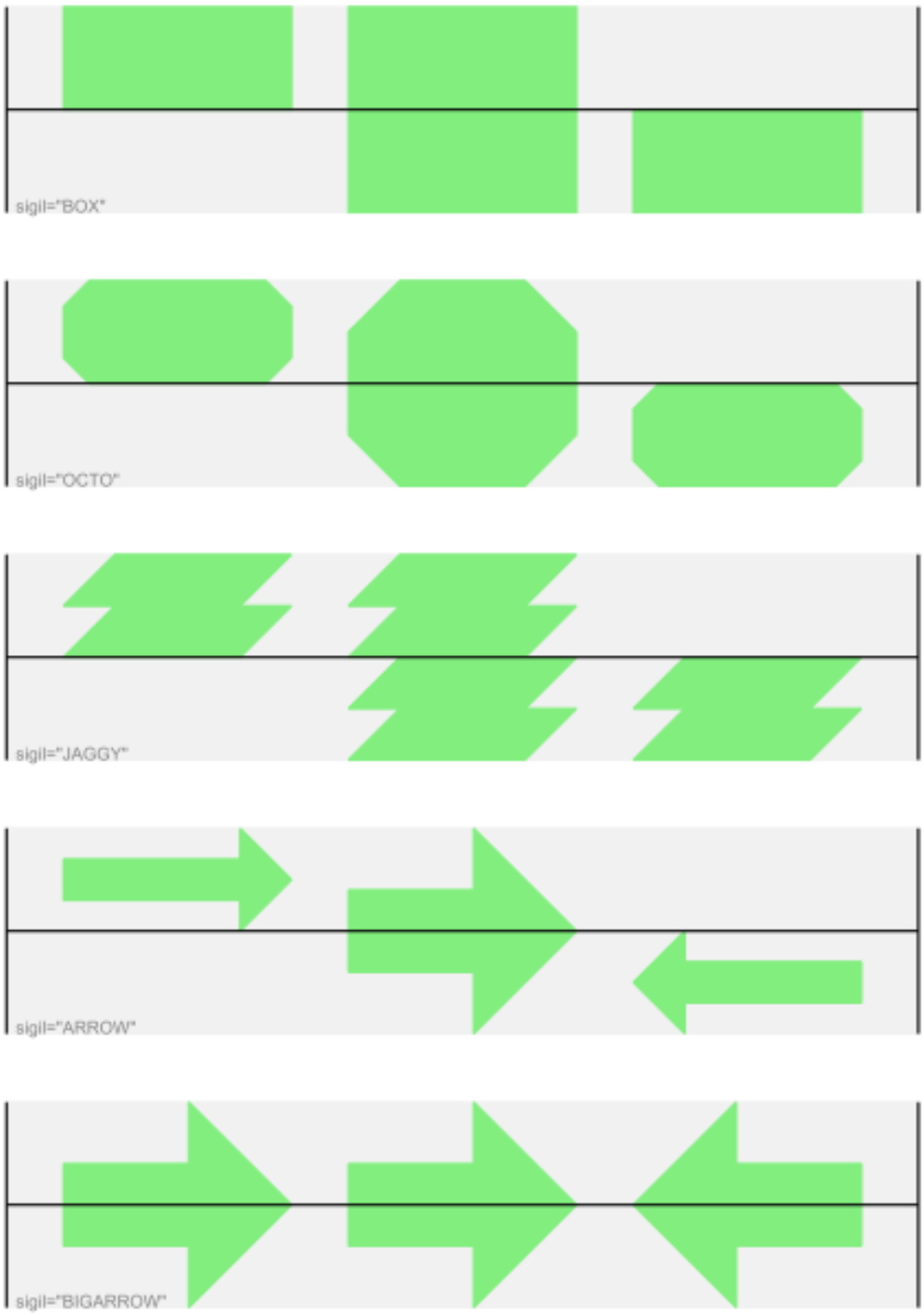


Fig. 4: Simple GenomeDiagram showing different sigils.

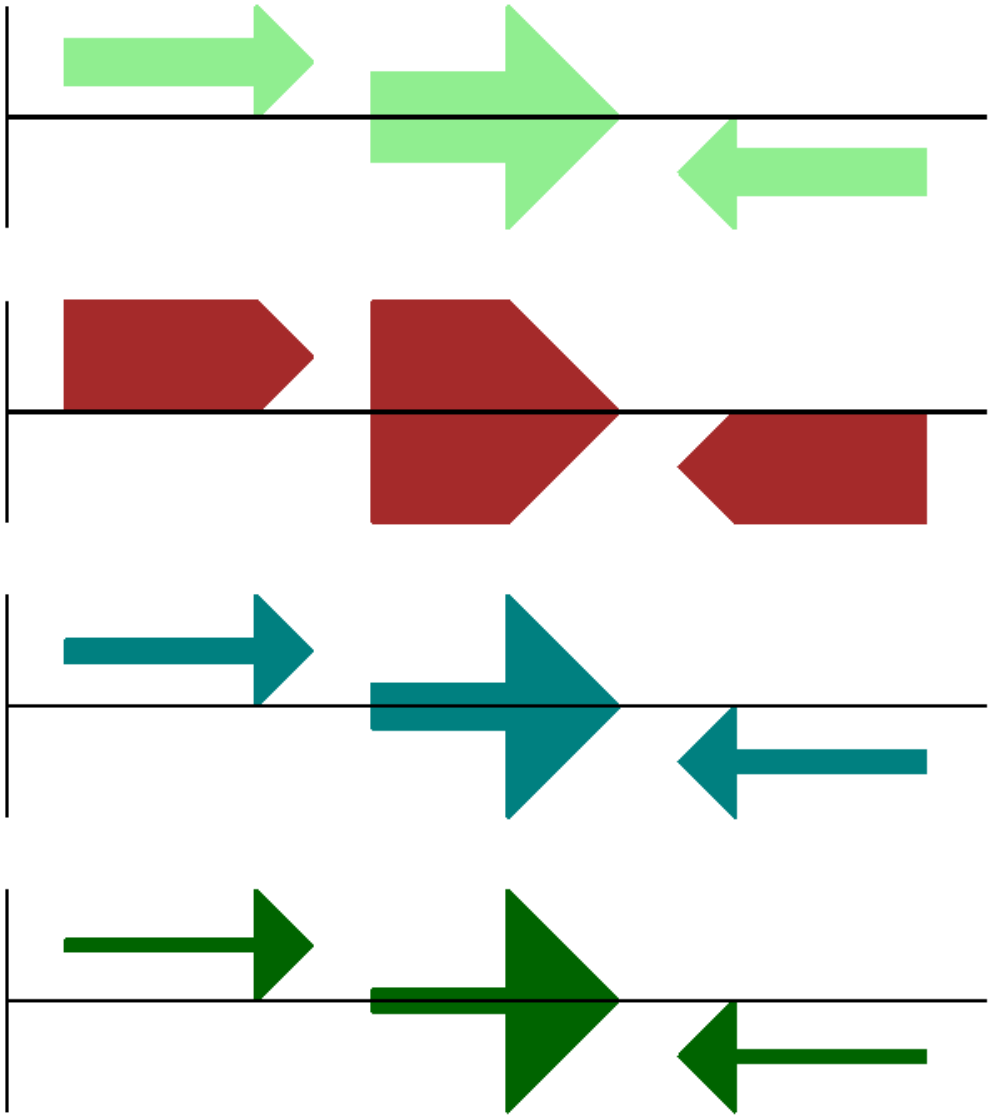


Fig. 5: Simple GenomeDiagram showing arrow shaft options.

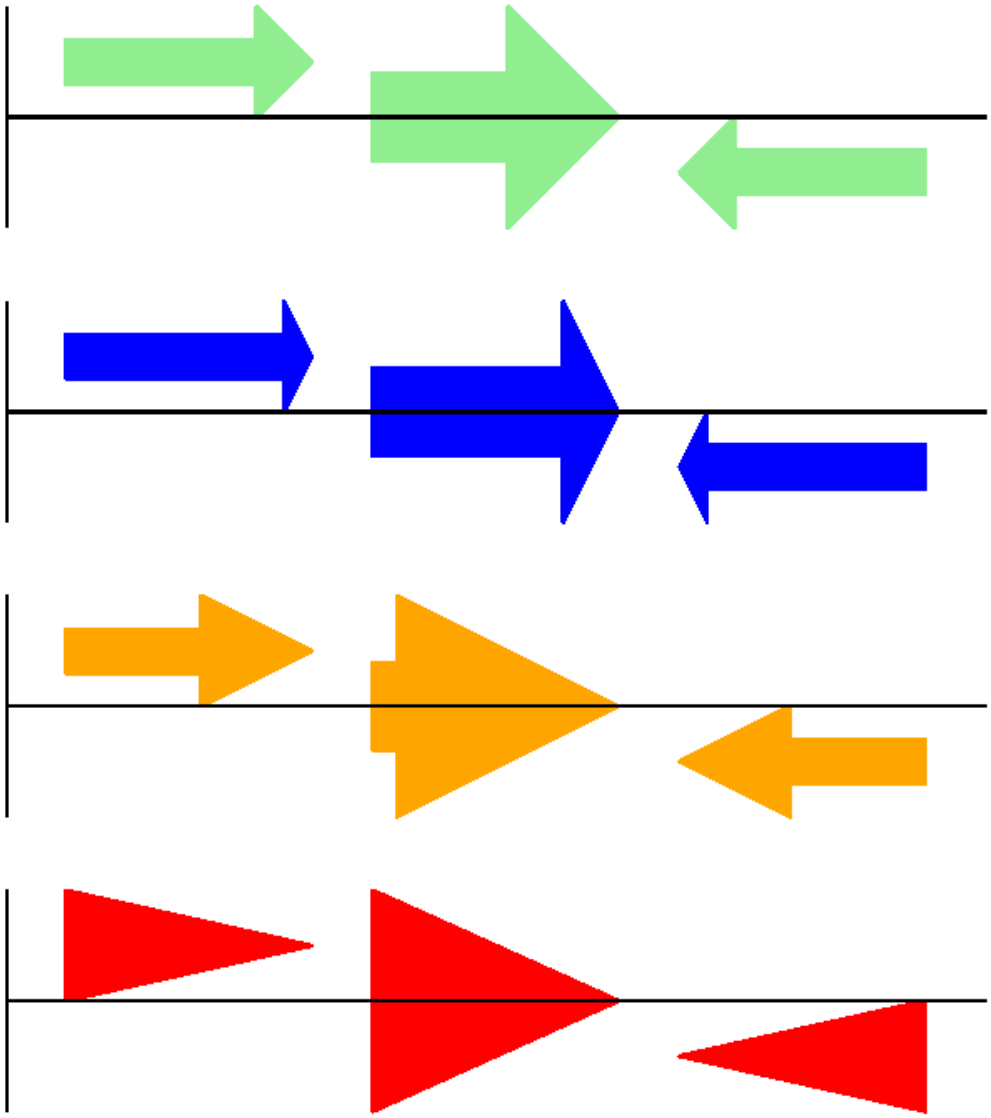


Fig. 6: Simple GenomeDiagram showing arrow head options.

20.1.9 A nice example

Now let's return to the pPCP1 plasmid from *Yersinia pestis biovar Microtus*, and the top down approach used in Section *A top down example*, but take advantage of the sigil options we've now discussed. This time we'll use arrows for the genes, and overlay them with strand-less features (as plain boxes) showing the position of some restriction digest sites.

```
from reportlab.lib import colors
from reportlab.lib.units import cm
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO
from Bio.SeqFeature import SeqFeature, SimpleLocation

record = SeqIO.read("NC_005816.gb", "genbank")

gd_diagram = GenomeDiagram.Diagram(record.id)
gd_track_for_features = gd_diagram.new_track(1, name="Annotated Features")
gd_feature_set = gd_track_for_features.new_set()

for feature in record.features:
    if feature.type != "gene":
        # Exclude this feature
        continue
    if len(gd_feature_set) % 2 == 0:
        color = colors.blue
    else:
        color = colors.lightblue
    gd_feature_set.add_feature(
        feature, sigil="ARROW", color=color, label=True, label_size=14, label_angle=0
    )

# I want to include some strandless features, so for an example
# will use EcoRI recognition sites etc.
for site, name, color in [
    ("GAATTC", "EcoRI", colors.green),
    ("CCCGGG", "SmaI", colors.orange),
    ("AAGCTT", "HindIII", colors.red),
    ("GGATCC", "BamHI", colors.purple),
]:
    index = 0
    while True:
        index = record.seq.find(site, start=index)
        if index == -1:
            break
        feature = SeqFeature(SimpleLocation(index, index + len(site)))
        gd_feature_set.add_feature(
            feature,
            color=color,
            name=name,
            label=True,
            label_size=10,
            label_color=color,
        )
```

(continues on next page)

(continued from previous page)

```
        index += len(site)

gd_diagram.draw(format="linear", pagesize="A4", fragments=4, start=0, end=len(record))
gd_diagram.write("plasmid_linear_nice.pdf", "PDF")
gd_diagram.write("plasmid_linear_nice.eps", "EPS")
gd_diagram.write("plasmid_linear_nice.svg", "SVG")

gd_diagram.draw(
    format="circular",
    circular=True,
    pagesize=(20 * cm, 20 * cm),
    start=0,
    end=len(record),
    circle_core=0.5,
)
gd_diagram.write("plasmid_circular_nice.pdf", "PDF")
gd_diagram.write("plasmid_circular_nice.eps", "EPS")
gd_diagram.write("plasmid_circular_nice.svg", "SVG")
```

The expected output is shown in Figures *Linear diagram for plasmid pPCP1 showing selected restriction digest sites.* and *Circular diagram for plasmid pPCP1 showing selected restriction digest sites* ..

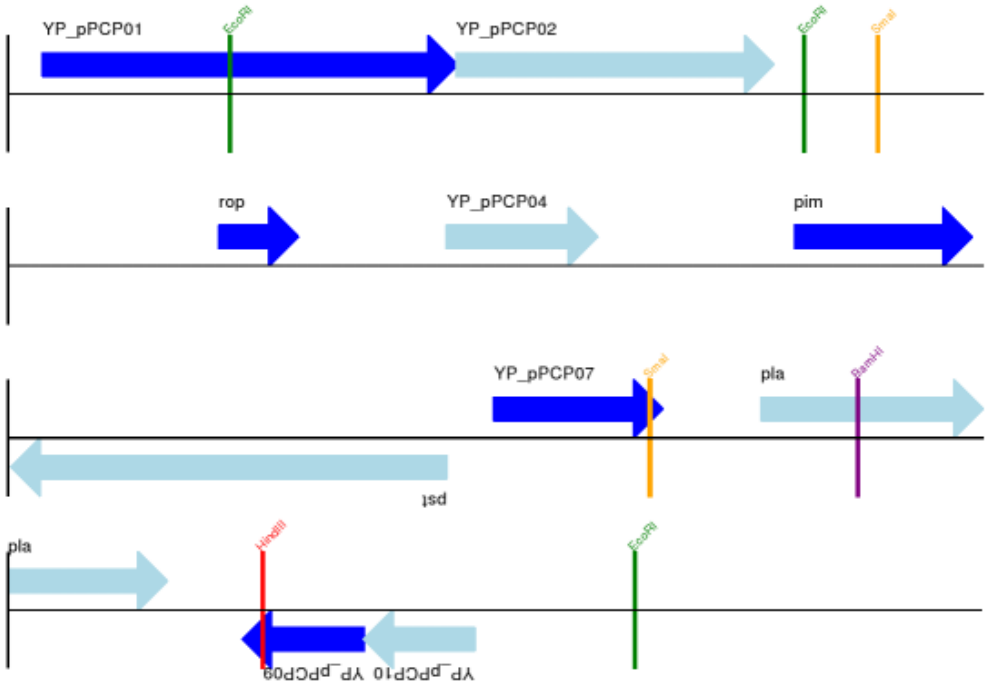


Fig. 7: Linear diagram for plasmid pPCP1 showing selected restriction digest sites.

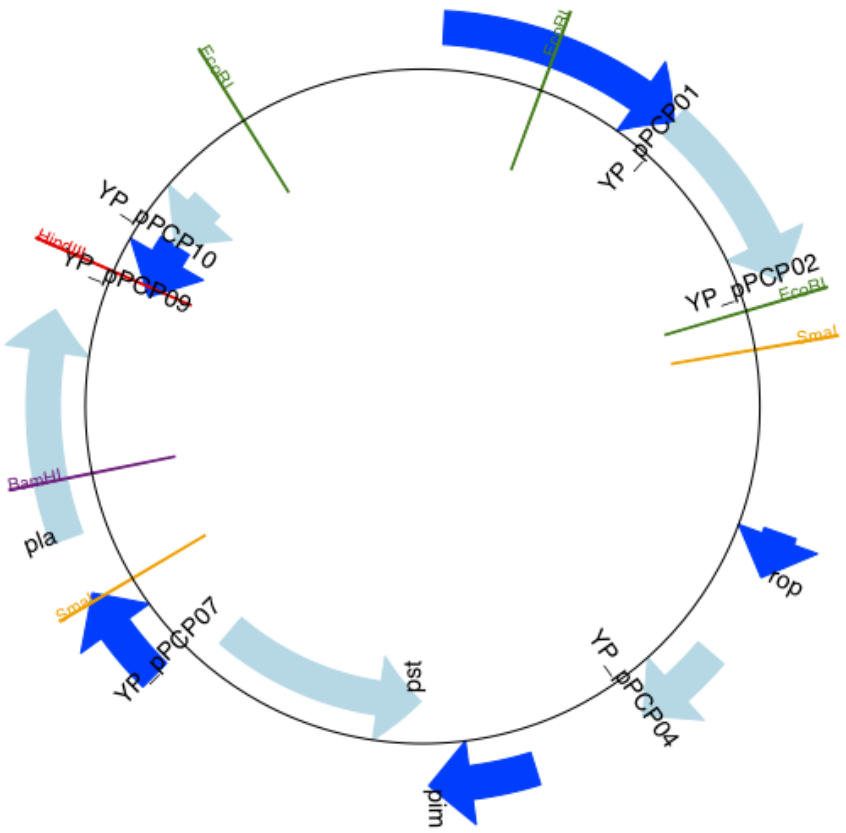


Fig. 8: Circular diagram for plasmid pPCP1 showing selected restriction digest sites .

20.1.10 Multiple tracks

All the examples so far have used a single track, but you can have more than one track – for example show the genes on one, and repeat regions on another. In this example we're going to show three phage genomes side by side to scale, inspired by Figure 6 in Proux *et al.* (2002) [Proux2002]. We'll need the GenBank files for the following three phage:

- `NC_002703` – Lactococcus phage Tuc2009, complete genome (38347 bp)
- `AF323668` – Bacteriophage bIL285, complete genome (35538 bp)
- `NC_003212` – *Listeria innocua* Clip11262, complete genome, of which we are focussing only on integrated prophage 5 (similar length).

You can download these using Entrez if you like, see Section *EFetch: Downloading full records from Entrez* for more details. For the third record we've worked out where the phage is integrated into the genome, and slice the record to extract it (with the features preserved, see Section *Slicing a SeqRecord*), and must also reverse complement to match the orientation of the first two phage (again preserving the features, see Section *Reverse-complementing SeqRecord objects*):

```
from Bio import SeqIO

A_rec = SeqIO.read("NC_002703.gbk", "gb")
B_rec = SeqIO.read("AF323668.gbk", "gb")
C_rec = SeqIO.read("NC_003212.gbk", "gb")[2587879:2625807].reverse_complement(name=True)
```

The figure we are imitating used different colors for different gene functions. One way to do this is to edit the GenBank file to record color preferences for each feature - something Sanger's Artemis editor does, and which GenomeDiagram should understand. Here however, we'll just hard code three lists of colors.

Note that the annotation in the GenBank files doesn't exactly match that shown in Proux *et al.*, they have drawn some unannotated genes.

```
from reportlab.lib.colors import (
    red,
    grey,
    orange,
    green,
    brown,
    blue,
    lightblue,
    purple,
)

A_colors = (
    [red] * 5
    + [grey] * 7
    + [orange] * 2
    + [grey] * 2
    + [orange]
    + [grey] * 11
    + [green] * 4
    + [grey]
    + [green] * 2
    + [grey, green]
    + [brown] * 5
    + [blue] * 4
```

(continues on next page)

(continued from previous page)

```
    + [lightblue] * 5
    + [grey, lightblue]
    + [purple] * 2
    + [grey]
)
B_colors = (
    [red] * 6
    + [grey] * 8
    + [orange] * 2
    + [grey]
    + [orange]
    + [grey] * 21
    + [green] * 5
    + [grey]
    + [brown] * 4
    + [blue] * 3
    + [lightblue] * 3
    + [grey] * 5
    + [purple] * 2
)
C_colors = (
    [grey] * 30
    + [green] * 5
    + [brown] * 4
    + [blue] * 2
    + [grey, blue]
    + [lightblue] * 2
    + [grey] * 5
)
```

Now to draw them – this time we add three tracks to the diagram, and also notice they are given different start/end values to reflect their different lengths (this requires Biopython 1.59 or later).

```
from Bio.Graphics import GenomeDiagram

name = "Proux Fig 6"
gd_diagram = GenomeDiagram.Diagram(name)
max_len = 0
for record, gene_colors in zip([A_rec, B_rec, C_rec], [A_colors, B_colors, C_colors]):
    max_len = max(max_len, len(record))
    gd_track_for_features = gd_diagram.new_track(
        1, name=record.name, greytrack=True, start=0, end=len(record)
    )
    gd_feature_set = gd_track_for_features.new_set()

    i = 0
    for feature in record.features:
        if feature.type != "gene":
            # Exclude this feature
            continue
        gd_feature_set.add_feature(
            feature,
```

(continues on next page)

(continued from previous page)

```
            sigil="ARROW",
            color=gene_colors[i],
            label=True,
            name=str(i + 1),
            label_position="start",
            label_size=6,
            label_angle=0,
        )
        i += 1

gd_diagram.draw(format="linear", pagesize="A4", fragments=1, start=0, end=max_len)
gd_diagram.write(name + ".pdf", "PDF")
gd_diagram.write(name + ".eps", "EPS")
gd_diagram.write(name + ".svg", "SVG")
```

The expected output is shown in Fig. 9.

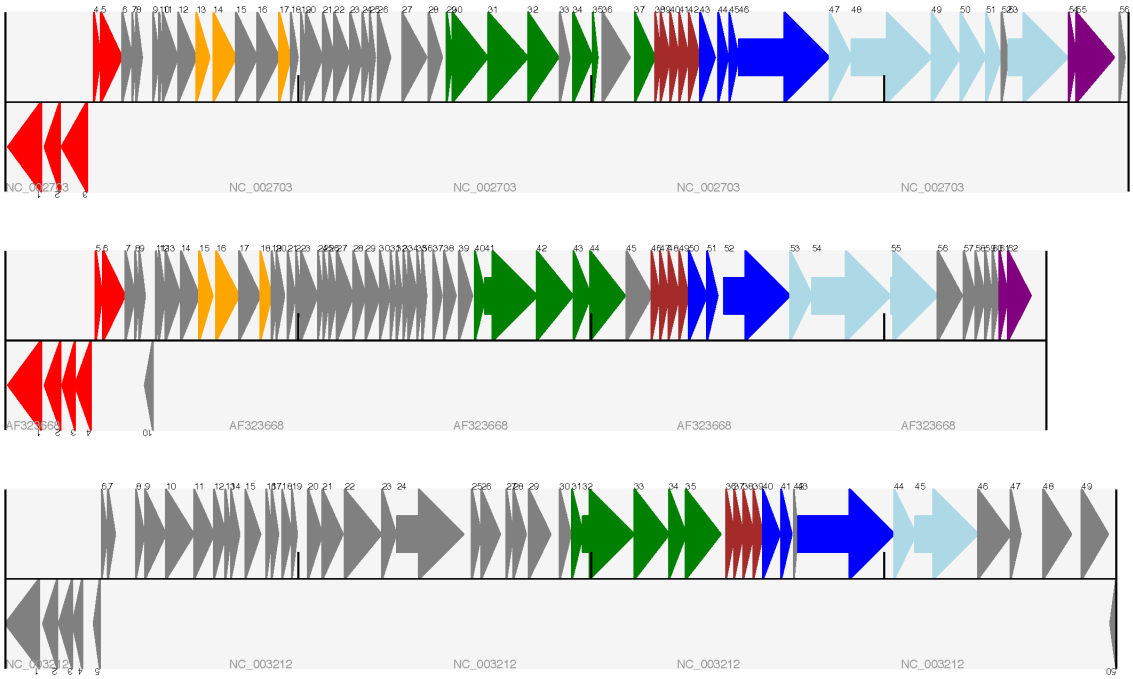


Fig. 9: Linear diagram with three tracks for three phages.
This shows Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from *Listeria innocua* Clip11262 (NC_003212) (see Section *Multiple tracks*).

I did wonder why in the original manuscript there were no red or orange genes marked in the bottom phage. Another important point is here the phage are shown with different lengths - this is because they are all drawn to the same scale (they *are* different lengths).

The key difference from the published figure is they have color-coded links between similar proteins – which is what we will do in the next section.

20.1.11 Cross-Links between tracks

Biopython 1.59 added the ability to draw cross links between tracks - both simple linear diagrams as we will show here, but also linear diagrams split into fragments and circular diagrams.

Continuing the example from the previous section inspired by Figure 6 from Proux *et al.* 2002 [Proux2002], we would need a list of cross links between pairs of genes, along with a score or color to use. Realistically you might extract this from a BLAST file computationally, but here I have manually typed them in.

My naming convention continues to refer to the three phage as A, B and C. Here are the links we want to show between A and B, given as a list of tuples (percentage similarity score, gene in A, gene in B).

```
# Tuc2009 (NC_002703) vs bIL285 (AF323668)
A_vs_B = [
    (99, "Tuc2009_01", "int"),
    (33, "Tuc2009_03", "orf4"),
    (94, "Tuc2009_05", "orf6"),
    (100, "Tuc2009_06", "orf7"),
    (97, "Tuc2009_07", "orf8"),
    (98, "Tuc2009_08", "orf9"),
    (98, "Tuc2009_09", "orf10"),
    (100, "Tuc2009_10", "orf12"),
    (100, "Tuc2009_11", "orf13"),
    (94, "Tuc2009_12", "orf14"),
    (87, "Tuc2009_13", "orf15"),
    (94, "Tuc2009_14", "orf16"),
    (94, "Tuc2009_15", "orf17"),
    (88, "Tuc2009_17", "rusA"),
    (91, "Tuc2009_18", "orf20"),
    (93, "Tuc2009_19", "orf22"),
    (71, "Tuc2009_20", "orf23"),
    (51, "Tuc2009_22", "orf27"),
    (97, "Tuc2009_23", "orf28"),
    (88, "Tuc2009_24", "orf29"),
    (26, "Tuc2009_26", "orf38"),
    (19, "Tuc2009_46", "orf52"),
    (77, "Tuc2009_48", "orf54"),
    (91, "Tuc2009_49", "orf55"),
    (95, "Tuc2009_52", "orf60"),
]
```

Likewise for B and C:

```
# bIL285 (AF323668) vs Listeria innocua prophage 5 (in NC_003212)
B_vs_C = [
    (42, "orf39", "lin2581"),
    (31, "orf40", "lin2580"),
    (49, "orf41", "lin2579"),  # terL
    (54, "orf42", "lin2578"),  # portal
    (55, "orf43", "lin2577"),  # protease
    (33, "orf44", "lin2576"),  # mhp
```

(continues on next page)

(continued from previous page)

```
    (51, "orf46", "lin2575"),
    (33, "orf47", "lin2574"),
    (40, "orf48", "lin2573"),
    (25, "orf49", "lin2572"),
    (50, "orf50", "lin2571"),
    (48, "orf51", "lin2570"),
    (24, "orf52", "lin2568"),
    (30, "orf53", "lin2567"),
    (28, "orf54", "lin2566"),
]
```

For the first and last phage these identifiers are locus tags, for the middle phage there are no locus tags so I've used gene names instead. The following little helper function lets us lookup a feature using either a locus tag or gene name:

```
def get_feature(features, id, tags=["locus_tag", "gene"]):
    """Search list of SeqFeature objects for an identifier under the given tags."""
    for f in features:
        for key in tags:
            # tag may not be present in this feature
            for x in f.qualifiers.get(key, []):
                if x == id:
                    return f
    raise KeyError(id)
```

We can now turn those list of identifier pairs into SeqFeature pairs, and thus find their location coordinates. We can now add all that code and the following snippet to the previous example (just before the `gd_diagram.draw(...)` line – see the finished example script Proux_et_al_2002_Figure_6.py included in the `Doc/examples` folder of the Biopython source code) to add cross links to the figure:

```
from Bio.Graphics.GenomeDiagram import CrossLink
from reportlab.lib import colors

# Note it might have been clearer to assign the track numbers explicitly...
for rec_X, tn_X, rec_Y, tn_Y, X_vs_Y in [
    (A_rec, 3, B_rec, 2, A_vs_B),
    (B_rec, 2, C_rec, 1, B_vs_C),
]:
    track_X = gd_diagram.tracks[tn_X]
    track_Y = gd_diagram.tracks[tn_Y]
    for score, id_X, id_Y in X_vs_Y:
        feature_X = get_feature(rec_X.features, id_X)
        feature_Y = get_feature(rec_Y.features, id_Y)
        color = colors.linearlyInterpolatedColor(
            colors.white, colors.firebrick, 0, 100, score
        )
        link_xy = CrossLink(
            (track_X, feature_X.location.start, feature_X.location.end),
            (track_Y, feature_Y.location.start, feature_Y.location.end),
            color,
            colors.lightgrey,
        )
        gd_diagram.cross_track_links.append(link_xy)
```

There are several important pieces to this code. First the `GenomeDiagram` object has a `cross_track_links` attribute which is just a list of `CrossLink` objects. Each `CrossLink` object takes two sets of track-specific coordinates (here given as tuples, you can alternatively use a `GenomeDiagram.Feature` object instead). You can optionally supply a color, border color, and say if this link should be drawn flipped (useful for showing inversions).

You can also see how we turn the BLAST percentage identity score into a color, interpolating between white (0%) and a dark red (100%). In this example we don't have any problems with overlapping cross-links. One way to tackle that is to use transparency in ReportLab, by using colors with their alpha channel set. However, this kind of shaded color scheme combined with overlap transparency would be difficult to interpret. The expected output is shown in Fig. 10.

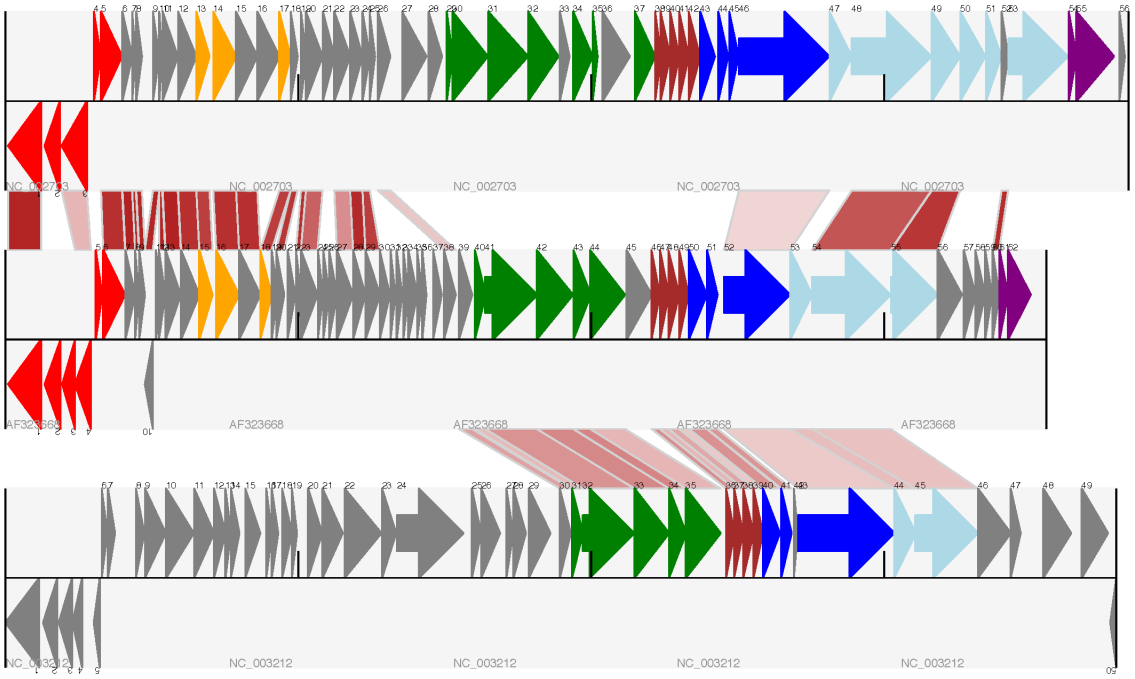


Fig. 10: Linear diagram with three tracks plus basic cross-links.
The three tracks show Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from *Listeria innocua* Clip11262 (NC_003212) plus basic cross-links shaded by percentage identity (see Section *Cross-Links between tracks*).

There is still a lot more that can be done within Biopython to help improve this figure. First of all, the cross links in this case are between proteins which are drawn in a strand specific manor. It can help to add a background region (a feature using the 'BOX' sigil) on the feature track to extend the cross link. Also, we could reduce the vertical height of the feature tracks to allocate more to the links instead – one way to do that is to allocate space for empty tracks. Furthermore, in cases like this where there are no large gene overlaps, we can use the axis-straddling `BIGARROW` sigil, which allows us to further reduce the vertical space needed for the track. These improvements are demonstrated in the example script Proux_et_al_2002_Figure_6.py included in the `Doc/examples` folder of the Biopython source code. The expected output is shown in Fig. 11.

Beyond that, finishing touches you might want to do manually in a vector image editor include fine tuning the placement of gene labels, and adding other custom annotation such as highlighting particular regions.

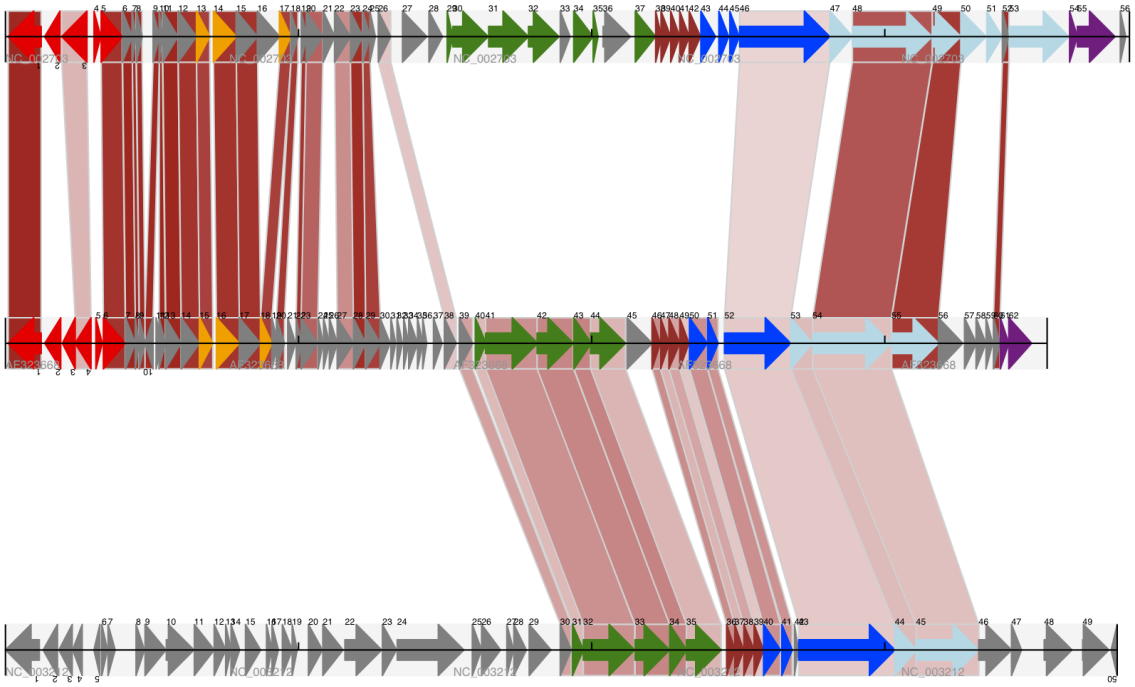


Fig. 11: Linear diagram with three tracks plus shaded cross-links.
The three tracks show Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from *Listeria innocua* Clip11262 (NC_003212) plus cross-links shaded by percentage identity (see Section *Cross-Links between tracks*).

Although not really necessary in this example since none of the cross-links overlap, using a transparent color in ReportLab is a very useful technique for superimposing multiple links. However, in this case a shaded color scheme should be avoided.

20.1.12 Further options

You can control the tick marks to show the scale – after all every graph should show its units, and the number of the grey-track labels.

Also, we have only used the `FeatureSet` so far. GenomeDiagram also has a `GraphSet` which can be used for show line graphs, bar charts and heat plots (e.g. to show plots of GC% on a track parallel to the features).

These options are not covered here yet, so for now we refer you to the User Guide (PDF) included with the standalone version of GenomeDiagram (but please read the next section first), and the docstrings.

20.1.13 Converting old code

If you have old code written using the standalone version of GenomeDiagram, and you want to switch it over to using the new version included with Biopython then you will have to make a few changes - most importantly to your import statements.

Also, the older version of GenomeDiagram used only the UK spellings of color and center (colour and centre). You will need to change to the American spellings, although for several years the Biopython version of GenomeDiagram supported both.

For example, if you used to have:

```
from GenomeDiagram import GDFeatureSet, GDDiagram

gdd = GDDiagram("An example")
...
```

you could just switch the import statements like this:

```
from Bio.Graphics.GenomeDiagram import FeatureSet as GDFeatureSet, Diagram as GDDiagram

gdd = GDDiagram("An example")
...
```

and hopefully that should be enough. In the long term you might want to switch to the new names, but you would have to change more of your code:

```
from Bio.Graphics.GenomeDiagram import FeatureSet, Diagram

gdd = Diagram("An example")
...
```

or:

```
from Bio.Graphics import GenomeDiagram

gdd = GenomeDiagram.Diagram("An example")
...
```

If you run into difficulties, please ask on the Biopython mailing list for advice. One catch is that we have not included the old module `GenomeDiagram.GDUtilities` yet. This included a number of GC% related functions, which will probably be merged under `Bio.SeqUtils` later on.

20.2 Chromosomes

The `Bio.Graphics.BasicChromosome` module allows drawing of chromosomes. There is an example in Jupe *et al.* (2012) [Jupe2012] (open access) using colors to highlight different gene families.

20.2.1 Simple Chromosomes

Here is a very simple example - for which we'll use *Arabidopsis thaliana*.

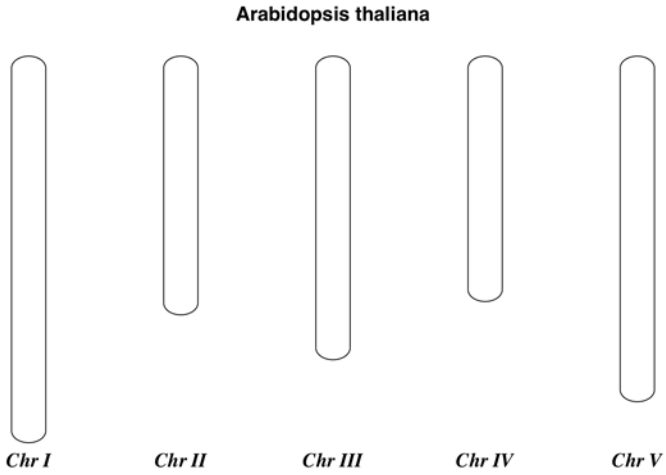


Fig. 12: Simple chromosome diagram for *Arabidopsis thaliana*.

You can skip this bit, but first I downloaded the five sequenced chromosomes as five individual FASTA files from the NCBI's FTP site ftp://ftp.ncbi.nlm.nih.gov/genomes/archive/old_refseq/Arabidopsis_thaliana/ and then parsed them with `Bio.SeqIO` to find out their lengths. You could use the GenBank files for this (and the next example uses those for plotting features), but if all you want is the length it is faster to use the FASTA files for the whole chromosomes:

```
from Bio import SeqIO

entries = [
    ("Chr I", "CHR_I/NC_003070.fna"),
    ("Chr II", "CHR_II/NC_003071.fna"),
    ("Chr III", "CHR_III/NC_003074.fna"),
    ("Chr IV", "CHR_IV/NC_003075.fna"),
    ("Chr V", "CHR_V/NC_003076.fna"),
]
for name, filename in entries:
    record = SeqIO.read(filename, "fasta")
    print(name, len(record))
```

This gave the lengths of the five chromosomes, which we'll now use in the following short demonstration of the BasicChromosome module:

```
from reportlab.lib.units import cm
from Bio.Graphics import BasicChromosome

entries = [
    ("Chr I", 30432563),
    ("Chr II", 19705359),
    ("Chr III", 23470805),
    ("Chr IV", 18585042),
    ("Chr V", 26992728),
]

max_len = 30432563  # Could compute this from the entries dict
telomere_length = 1000000  # For illustration

chr_diagram = BasicChromosome.Organism()
chr_diagram.page_size = (29.7 * cm, 21 * cm)  # A4 landscape

for name, length in entries:
    cur_chromosome = BasicChromosome.Chromosome(name)
    # Set the scale to the MAXIMUM length plus the two telomeres in bp,
    # want the same scale used on all five chromosomes so they can be
    # compared to each other
    cur_chromosome.scale_num = max_len + 2 * telomere_length

    # Add an opening telomere
    start = BasicChromosome.TelomereSegment()
    start.scale = telomere_length
    cur_chromosome.add(start)

    # Add a body - using bp as the scale length here.
    body = BasicChromosome.ChromosomeSegment()
    body.scale = length
    cur_chromosome.add(body)

    # Add a closing telomere
    end = BasicChromosome.TelomereSegment(inverted=True)
    end.scale = telomere_length
    cur_chromosome.add(end)

    # This chromosome is done
    chr_diagram.add(cur_chromosome)

chr_diagram.draw("simple_chrom.pdf", "Arabidopsis thaliana")
```

This should create a very simple PDF file, shown in Fig. 12. This example is deliberately short and sweet. The next example shows the location of features of interest.

20.2.2 Annotated Chromosomes

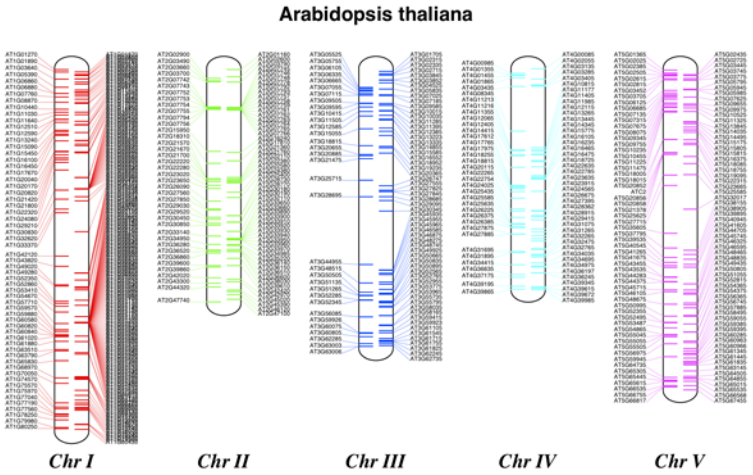


Fig. 13: Chromosome diagram for *Arabidopsis thaliana* showing tRNA genes.

Continuing from the previous example, let's also show the tRNA genes. We'll get their locations by parsing the GenBank files for the five *Arabidopsis thaliana* chromosomes. You'll need to download these files from the NCBI FTP site ftp://ftp.ncbi.nlm.nih.gov/genomes/archive/old_refseq/Arabidopsis_thaliana/, and preserve the subdirectory names or edit the paths below:

```
from reportlab.lib.units import cm
from Bio import SeqIO
from Bio.Graphics import BasicChromosome

entries = [
    ("Chr I", "CHR_I/NC_003070.gbk"),
    ("Chr II", "CHR_II/NC_003071.gbk"),
    ("Chr III", "CHR_III/NC_003074.gbk"),
    ("Chr IV", "CHR_IV/NC_003075.gbk"),
    ("Chr V", "CHR_V/NC_003076.gbk"),
]

max_len = 30432563  # Could compute this from the entries dict
telomere_length = 1000000  # For illustration

chr_diagram = BasicChromosome.Organism()
chr_diagram.page_size = (29.7 * cm, 21 * cm)  # A4 landscape

for index, (name, filename) in enumerate(entries):
    record = SeqIO.read(filename, "genbank")
    length = len(record)
    features = [f for f in record.features if f.type == "tRNA"]
    # Record an Artemis style integer color in the feature's qualifiers,
    # 1 = Black, 2 = Red, 3 = Green, 4 = blue, 5 =cyan, 6 = purple
    for f in features:
```

(continues on next page)

(continued from previous page)

```
        f.qualifiers["color"] = [index + 2]

    cur_chromosome = BasicChromosome.Chromosome(name)
    # Set the scale to the MAXIMUM length plus the two telomeres in bp,
    # want the same scale used on all five chromosomes so they can be
    # compared to each other
    cur_chromosome.scale_num = max_len + 2 * telomere_length

    # Add an opening telomere
    start = BasicChromosome.TelomereSegment()
    start.scale = telomere_length
    cur_chromosome.add(start)

    # Add a body - again using bp as the scale length here.
    body = BasicChromosome.AnnotatedChromosomeSegment(length, features)
    body.scale = length
    cur_chromosome.add(body)

    # Add a closing telomere
    end = BasicChromosome.TelomereSegment(inverted=True)
    end.scale = telomere_length
    cur_chromosome.add(end)

    # This chromosome is done
    chr_diagram.add(cur_chromosome)

chr_diagram.draw("tRNA_chrom.pdf", "Arabidopsis thaliana")
```

It might warn you about the labels being too close together - have a look at the forward strand (right hand side) of Chr I, but it should create a colorful PDF file, shown in Fig. 13.

CHAPTER

TWENTYONE

KEGG

KEGG (https://www.kegg.jp/) is a database resource for understanding high-level functions and utilities of the biological system, such as the cell, the organism and the ecosystem, from molecular-level information, especially large-scale molecular datasets generated by genome sequencing and other high-throughput experimental technologies.

Please note that the KEGG parser implementation in Biopython is incomplete. While the KEGG website indicates many flat file formats, only parsers and writers for compound, enzyme, and map are currently implemented. However, a generic parser is implemented to handle the other formats.

21.1 Parsing KEGG records

Parsing a KEGG record is as simple as using any other file format parser in Biopython. (Before running the following codes, please open http://rest.kegg.jp/get/ec:5.4.2.2 with your web browser and save it as `ec_5.4.2.2.txt`.)

```
>>> from Bio.KEGG import Enzyme
>>> records = Enzyme.parse(open("ec_5.4.2.2.txt"))
>>> record = list(records)[0]
>>> record.classname
['Isomerases;', 'Intramolecular transferases;', 'Phosphotransferases (phosphomutases)']
>>> record.entry
'5.4.2.2'
```

Alternatively, if the input KEGG file has exactly one entry, you can use `read`:

```
>>> from Bio.KEGG import Enzyme
>>> record = Enzyme.read(open("ec_5.4.2.2.txt"))
>>> record.classname
['Isomerases;', 'Intramolecular transferases;', 'Phosphotransferases (phosphomutases)']
>>> record.entry
'5.4.2.2'
```

The following section will shows how to download the above enzyme using the KEGG api as well as how to use the generic parser with data that does not have a custom parser implemented.

21.2 Querying the KEGG API

Biopython has full support for the querying of the KEGG api. Querying all KEGG endpoints are supported; all methods documented by KEGG (https://www.kegg.jp/kegg/rest/keggapi.html) are supported. The interface has some validation of queries which follow rules defined on the KEGG site. However, invalid queries which return a 400 or 404 must be handled by the user.

First, here is how to extend the above example by downloading the relevant enzyme and passing it through the Enzyme parser.

```
>>> from Bio.KEGG import REST
>>> from Bio.KEGG import Enzyme
>>> request = REST.kegg_get("ec:5.4.2.2")
>>> open("ec_5.4.2.2.txt", "w").write(request.read())
>>> records = Enzyme.parse(open("ec_5.4.2.2.txt"))
>>> record = list(records)[0]
>>> record.classname
['Isomerases;', 'Intramolecular transferases;', 'Phosphotransferases (phosphomutases)']
>>> record.entry
'5.4.2.2'
```

Now, here's a more realistic example which shows a combination of querying the KEGG API. This will demonstrate how to extract a unique set of all human pathway gene symbols which relate to DNA repair. The steps that need to be taken to do so are as follows. First, we need to get a list of all human pathways. Secondly, we need to filter those for ones which relate to "repair". Lastly, we need to get a list of all the gene symbols in all repair pathways.

```
from Bio.KEGG import REST

human_pathways = REST.kegg_list("pathway", "hsa").read()

# Filter all human pathways for repair pathways
repair_pathways = []
for line in human_pathways.rstrip().split("\n"):
    entry, description = line.split("\t")
    if "repair" in description:
        repair_pathways.append(entry)

# Get the genes for pathways and add them to a list
repair_genes = []
for pathway in repair_pathways:
    pathway_file = REST.kegg_get(pathway).read()  # query and read each pathway

    # iterate through each KEGG pathway file, keeping track of which section
    # of the file we're in, only read the gene in each pathway
    current_section = None
    for line in pathway_file.rstrip().split("\n"):
        section = line[:12].strip()  # section names are within 12 columns
        if not section == "":
            current_section = section

        if current_section == "GENE":
            gene_identifiers, gene_description = line[12:].split("; ")
            gene_id, gene_symbol = gene_identifiers.split()
```

(continues on next page)

(continued from previous page)

```
            if not gene_symbol in repair_genes:
                repair_genes.append(gene_symbol)

print(
    "There are %d repair pathways and %d repair genes. The genes are:"
    % (len(repair_pathways), len(repair_genes))
)
print(", ".join(repair_genes))
```

The KEGG API wrapper is compatible with all endpoints. Usage is essentially replacing all slashes in the url with commas and using that list as arguments to the corresponding method in the KEGG module. Here are a few examples from the api documentation (https://www.kegg.jp/kegg/docs/keggapi.html).

```
/list/hsa:10458+ece:Z5100            -> REST.kegg_list(["hsa:10458", "ece:Z5100"])
/find/compound/300-310/mol_weight    -> REST.kegg_find("compound", "300-310", "mol_weight
↪")
/get/hsa:10458+ece:Z5100/aaseq      -> REST.kegg_get(["hsa:10458", "ece:Z5100"], "aaseq")
```

CHAPTER

TWENTYTWO

BIO.PHENOTYPE: ANALYZE PHENOTYPIC DATA

This chapter gives an overview of the functionalities of the `Bio.phenotype` package included in Biopython. The scope of this package is the analysis of phenotypic data, which means parsing and analyzing growth measurements of cell cultures. In its current state the package is focused on the analysis of high-throughput phenotypic experiments produced by the Phenotype Microarray technology, but future developments may include other platforms and formats.

22.1 Phenotype Microarrays

The Phenotype Microarray is a technology that measures the metabolism of bacterial and eukaryotic cells on roughly 2000 chemicals, divided in twenty 96-well plates. The technology measures the reduction of a tetrazolium dye by NADH, whose production by the cell is used as a proxy for cell metabolism; color development due to the reduction of this dye is typically measured once every 15 minutes. When cells are grown in a media that sustains cell metabolism, the recorded phenotypic data resembles a sigmoid growth curve, from which a series of growth parameters can be retrieved.

22.1.1 Parsing Phenotype Microarray data

The `Bio.phenotype` package can parse two different formats of Phenotype Microarray data: the CSV (comma separated values) files produced by the machine's proprietary software and JSON files produced by analysis software, like opm or DuctApe. The parser will return one or a generator of PlateRecord objects, depending on whether the read or parse method is being used. You can test the parse function by using the Plates.csv file provided with the Biopython source code.

```
>>> from Bio import phenotype
>>> for record in phenotype.parse("Plates.csv", "pm-csv"):
...     print("%s %i" % (record.id, len(record)))
...
PM01 96
PM01 96
PM09 96
PM09 96
```

The parser returns a series of PlateRecord objects, each one containing a series of WellRecord objects (holding each well's experimental data) arranged in 8 rows and 12 columns; each row is indicated by a uppercase character from A to H, while columns are indicated by a two digit number, from 01 to 12. There are several ways to access WellRecord objects from a PlateRecord objects:

Well identifier
: If you know the well identifier (row + column identifiers) you can access the desired well directly.

```
>>> record["A02"]
WellRecord('(0.0, 12.0), (0.25, 18.0), (0.5, 27.0), (0.75, 35.0), (1.0, 37.0), ...,␣
↪(71.75, 143.0)')
```

Well plate coordinates

The same well can be retrieved by using the row and columns numbers (0-based index).

```
>>> from Bio import phenotype
>>> record = list(phenotype.parse("Plates.csv", "pm-csv"))[-1]
>>> print(record[0, 1].id)
A02
```

Row or column coordinates

A series of WellRecord objects contiguous to each other in the plate can be retrieved in bulk by using the python list slicing syntax on PlateRecord objects; rows and columns are numbered with a 0-based index.

```
>>> print(record[0])
Plate ID: PM09
Well: 12
Rows: 1
Columns: 12
PlateRecord('WellRecord['A01'], WellRecord['A02'], WellRecord['A03'], ...,␣
↪WellRecord['A12']')
>>> print(record[:, 0])
Plate ID: PM09
Well: 8
Rows: 8
Columns: 1
PlateRecord('WellRecord['A01'], WellRecord['B01'], WellRecord['C01'], ...,␣
↪WellRecord['H01']')
>>> print(record[:3, :3])
Plate ID: PM09
Well: 9
Rows: 3
Columns: 3
PlateRecord('WellRecord['A01'], WellRecord['A02'], WellRecord['A03'], ...,␣
↪WellRecord['C03']')
```

22.1.2 Manipulating Phenotype Microarray data

Accessing raw data

The raw data extracted from the PM files is comprised of a series of tuples for each well, containing the time (in hours) and the colorimetric measure (in arbitrary units). Usually the instrument collects data every fifteen minutes, but that can vary between experiments. The raw data can be accessed by iterating on a WellRecord object; in the example below only the first ten time points are shown.

```
>>> from Bio import phenotype
>>> record = list(phenotype.parse("Plates.csv", "pm-csv"))[-1]
>>> well = record["A02"]
```

```
>>> for time, signal in well:
...     print(time, signal)
...
(0.0, 12.0)
(0.25, 18.0)
(0.5, 27.0)
(0.75, 35.0)
(1.0, 37.0)
(1.25, 41.0)
(1.5, 44.0)
(1.75, 44.0)
(2.0, 44.0)
(2.25, 44.0)
[...]
```

This method, while providing a way to access the raw data, doesn't allow a direct comparison between different WellRecord objects, which may have measurements at different time points.

Accessing interpolated data

To make it easier to compare different experiments and in general to allow a more intuitive handling of the phenotypic data, the module allows to define a custom slicing of the time points that are present in the WellRecord object. Colorimetric data for time points that have not been directly measured are derived through a linear interpolation of the available data, otherwise a NaN is returned. This method only works in the time interval where actual data is available. Time intervals can be defined with the same syntax as list indexing; the default time interval is therefore one hour.

```
>>> well[:10]
[12.0, 37.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0]
>>> well[9.55]
44.0
```

Different time intervals can be used, for instance five minutes:

```
>>> for value in well[63 : 64 : 5 / 60]:
...     print(f"{value:0.2f}")
...
110.00
111.00
112.00
113.00
113.33
113.67
114.00
114.33
114.67
115.00
115.00
115.00
>>> for value in well[63.33:73.33]:
...     print(f"{value:0.2f}")
...
113.32
```

(continues on next page)

(continued from previous page)

```
117.00
120.32
128.00
129.64
132.96
136.96
140.00
142.00
nan
```

Control well subtraction

Many Phenotype Microarray plates contain a control well (usually A01), that is a well where the media shouldn't support any growth; the low signal produced by this well can be subtracted from the other wells. The PlateRecord objects have a dedicated function for that, which returns another PlateRecord object with the corrected data.

```
>>> corrected = record.subtract_control(control="A01")
>>> record["A01"][63]
336.0
>>> corrected["A01"][63]
0.0
```

Parameters extraction

Those wells where metabolic activity is observed show a sigmoid behavior for the colorimetric data. To allow an easier way to compare different experiments a sigmoid curve can be fitted onto the data, so that a series of summary parameters can be extracted and used for comparisons. The parameters that can be extracted from the curve are:

- Minimum (**min**) and maximum (**max**) signal;
- Average height (**average_height**);
- Area under the curve (**area**);
- Curve plateau point (**plateau**);
- Curve slope during exponential metabolic activity (**slope**);
- Curve lag time (**lag**).

All the parameters (except **min**, **max** and **average_height**) require the scipy library to be installed.

The fit function uses three sigmoid functions:

Gompertz

$$Ae^{-e^{(\frac{\mu_m e}{A}(\lambda - t)+1)}} + y0$$

Logistic

$$\frac{A}{1+e^{(\frac{4\mu_m}{A}(\lambda - t)+2)}} + y_0$$

Richards

$$A(1 + ve^{1+v} + e^{\frac{\mu_m}{A}(1+v)(1+\frac{1}{v})(\lambda - t)})^{-\frac{1}{v}} + y0$$

Where:

- corresponds to the **plateau**

- corresponds to the **slope**
- corresponds to the **lag**

These functions have been derived from this publication. The fit method by default tries first to fit the gompertz function: if it fails it will then try to fit the logistic and then the richards function. The user can also specify one of the three functions to be applied.

```
>>> from Bio import phenotype
>>> record = list(phenotype.parse("Plates.csv", "pm-csv"))[-1]
>>> well = record["A02"]
>>> well.fit()
>>> print("Function fitted: %s" % well.model)
Function fitted: gompertz
>>> for param in ["area", "average_height", "lag", "max", "min", "plateau", "slope"]:
...     print("%s\t%.2f" % (param, getattr(well, param)))
...
area    4414.38
average_height  61.58
lag     48.60
max     143.00
min     12.00
plateau 120.02
slope   4.99
```

22.1.3 Writing Phenotype Microarray data

PlateRecord objects can be written to file in the form of JSON files, a format compatible with other software packages such as opm or DuctApe.

```
>>> phenotype.write(record, "out.json", "pm-json")
1
```

CHAPTER

TWENTYTHREE

COOKBOOK – COOL THINGS TO DO WITH IT

Biopython now has two collections of "cookbook" examples – this chapter (which has been included in this tutorial for many years and has gradually grown), and http://biopython.org/wiki/Category:Cookbook which is a user contributed collection on our wiki.

We're trying to encourage Biopython users to contribute their own examples to the wiki. In addition to helping the community, one direct benefit of sharing an example like this is that you could also get some feedback on the code from other Biopython users and developers - which could help you improve all your Python code.

In the long term, we may end up moving all of the examples in this chapter to the wiki, or elsewhere within the tutorial.

23.1 Working with sequence files

This section shows some more examples of sequence input/output, using the `Bio.SeqIO` module described in Chapter *Sequence Input/Output*.

23.1.1 Filtering a sequence file

Often you'll have a large file with many sequences in it (e.g. FASTA file or genes, or a FASTQ or SFF file of reads), a separate shorter list of the IDs for a subset of sequences of interest, and want to make a new sequence file for this subset.

Let's say the list of IDs is in a simple text file, as the first word on each line. This could be a tabular file where the first column is the ID. Try something like this:

```
from Bio import SeqIO

input_file = "big_file.sff"
id_file = "short_list.txt"
output_file = "short_list.sff"

with open(id_file) as id_handle:
    wanted = set(line.rstrip("\n").split(None, 1)[0] for line in id_handle)
print("Found %i unique identifiers in %s" % (len(wanted), id_file))

records = (r for r in SeqIO.parse(input_file, "sff") if r.id in wanted)
count = SeqIO.write(records, output_file, "sff")
print("Saved %i records from %s to %s" % (count, input_file, output_file))
if count < len(wanted):
    print("Warning %i IDs not found in %s" % (len(wanted) - count, input_file))
```

Note that we use a Python `set` rather than a `list`, this makes testing membership faster.

As discussed in Section *Low level FASTA and FASTQ parsers*, for a large FASTA or FASTQ file for speed you would be better off not using the high-level `SeqIO` interface, but working directly with strings. This next example shows how to do this with FASTQ files – it is more complicated:

```
from Bio.SeqIO.QualityIO import FastqGeneralIterator

input_file = "big_file.fastq"
id_file = "short_list.txt"
output_file = "short_list.fastq"

with open(id_file) as id_handle:
    # Taking first word on each line as an identifier
    wanted = set(line.rstrip("\n").split(None, 1)[0] for line in id_handle)
print("Found %i unique identifiers in %s" % (len(wanted), id_file))

with open(input_file) as in_handle:
    with open(output_file, "w") as out_handle:
        for title, seq, qual in FastqGeneralIterator(in_handle):
            # The ID is the first word in the title line (after the @ sign):
            if title.split(None, 1)[0] in wanted:
                # This produces a standard 4-line FASTQ entry:
                out_handle.write("@%s\n%s\n+\n%s\n" % (title, seq, qual))
                count += 1
print("Saved %i records from %s to %s" % (count, input_file, output_file))
if count < len(wanted):
    print("Warning %i IDs not found in %s" % (len(wanted) - count, input_file))
```

23.1.2 Producing randomized genomes

Let's suppose you are looking at genome sequence, hunting for some sequence feature – maybe extreme local GC% bias, or possible restriction digest sites. Once you've got your Python code working on the real genome it may be sensible to try running the same search on randomized versions of the same genome for statistical analysis (after all, any "features" you've found could just be there just by chance).

For this discussion, we'll use the GenBank file for the pPCP1 plasmid from *Yersinia pestis biovar Microtus*. The file is included with the Biopython unit tests under the GenBank folder, or you can get it from our website, NC_005816.gb. This file contains one and only one record, so we can read it in as a `SeqRecord` using the `Bio.SeqIO.read()` function:

```
>>> from Bio import SeqIO
>>> original_rec = SeqIO.read("NC_005816.gb", "genbank")
```

So, how can we generate a shuffled versions of the original sequence? I would use the built-in Python `random` module for this, in particular the function `random.shuffle` – but this works on a Python list. Our sequence is a `Seq` object, so in order to shuffle it we need to turn it into a list:

```
>>> import random
>>> nuc_list = list(original_rec.seq)
>>> random.shuffle(nuc_list)  # acts in situ!
```

Now, in order to use `Bio.SeqIO` to output the shuffled sequence, we need to construct a new `SeqRecord` with a new `Seq` object using this shuffled list. In order to do this, we need to turn the list of nucleotides (single letter strings) into a long string – the standard Python way to do this is with the string object's join method.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> shuffled_rec = SeqRecord(
...     Seq("".join(nuc_list)), id="Shuffled", description="Based on %s" % original_rec.
↪id
... )
```

Let's put all these pieces together to make a complete Python script which generates a single FASTA file containing 30 randomly shuffled versions of the original sequence.

This first version just uses a big for loop and writes out the records one by one (using the `SeqRecord`'s format method described in Section *Getting your SeqRecord objects as formatted strings*):

```
import random
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO

original_rec = SeqIO.read("NC_005816.gb", "genbank")

with open("shuffled.fasta", "w") as output_handle:
    for i in range(30):
        nuc_list = list(original_rec.seq)
        random.shuffle(nuc_list)
        shuffled_rec = SeqRecord(
            Seq("".join(nuc_list)),
            id="Shuffled%i" % (i + 1),
            description="Based on %s" % original_rec.id,
        )
        output_handle.write(shuffled_rec.format("fasta"))
```

Personally I prefer the following version using a function to shuffle the record and a generator expression instead of the for loop:

```
import random
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO


def make_shuffle_record(record, new_id):
    nuc_list = list(record.seq)
    random.shuffle(nuc_list)
    return SeqRecord(
        Seq("".join(nuc_list)),
        id=new_id,
        description="Based on %s" % original_rec.id,
    )


original_rec = SeqIO.read("NC_005816.gb", "genbank")
shuffled_recs = (
    make_shuffle_record(original_rec, "Shuffled%i" % (i + 1)) for i in range(30)
)
```

(continues on next page)

(continued from previous page)

```
SeqIO.write(shuffled_recs, "shuffled.fasta", "fasta")
```

23.1.3 Translating a FASTA file of CDS entries

Suppose you've got an input file of CDS entries for some organism, and you want to generate a new FASTA file containing their protein sequences. i.e. Take each nucleotide sequence from the original file, and translate it. Back in Section *Translation* we saw how to use the `Seq` object's `translate method`, and the optional `cds` argument which enables correct translation of alternative start codons.

We can combine this with `Bio.SeqIO` as shown in the reverse complement example in Section *Converting a file of sequences to their reverse complements*. The key point is that for each nucleotide `SeqRecord`, we need to create a protein `SeqRecord` - and take care of naming it.

You can write you own function to do this, choosing suitable protein identifiers for your sequences, and the appropriate genetic code. In this example we just use the default table and add a prefix to the identifier:

```
from Bio.SeqRecord import SeqRecord


def make_protein_record(nuc_record):
    """Returns a new SeqRecord with the translated sequence (default table)."""
    return SeqRecord(
        seq=nuc_record.seq.translate(cds=True),
        id="trans_" + nuc_record.id,
        description="translation of CDS, using default table",
    )
```

We can then use this function to turn the input nucleotide records into protein records ready for output. An elegant way and memory efficient way to do this is with a generator expression:

```
from Bio import SeqIO

proteins = (
    make_protein_record(nuc_rec)
    for nuc_rec in SeqIO.parse("coding_sequences.fasta", "fasta")
)
SeqIO.write(proteins, "translations.fasta", "fasta")
```

This should work on any FASTA file of complete coding sequences. If you are working on partial coding sequences, you may prefer to use `nuc_record.seq.translate(to_stop=True)` in the example above, as this wouldn't check for a valid start codon etc.

23.1.4 Making the sequences in a FASTA file upper case

Often you'll get data from collaborators as FASTA files, and sometimes the sequences can be in a mixture of upper and lower case. In some cases this is deliberate (e.g. lower case for poor quality regions), but usually it is not important. You may want to edit the file to make everything consistent (e.g. all upper case), and you can do this easily using the `upper()` method of the `SeqRecord` object (added in Biopython 1.55):

```
from Bio import SeqIO

records = (rec.upper() for rec in SeqIO.parse("mixed.fas", "fasta"))
count = SeqIO.write(records, "upper.fas", "fasta")
print("Converted %i records to upper case" % count)
```

How does this work? The first line is just importing the `Bio.SeqIO` module. The second line is the interesting bit – this is a Python generator expression which gives an upper case version of each record parsed from the input file (`mixed.fas`). In the third line we give this generator expression to the `Bio.SeqIO.write()` function and it saves the new upper cases records to our output file (`upper.fas`).

The reason we use a generator expression (rather than a list or list comprehension) is this means only one record is kept in memory at a time. This can be really important if you are dealing with large files with millions of entries.

23.1.5 Sorting a sequence file

Suppose you wanted to sort a sequence file by length (e.g. a set of contigs from an assembly), and you are working with a file format like FASTA or FASTQ which `Bio.SeqIO` can read, write (and index).

If the file is small enough, you can load it all into memory at once as a list of `SeqRecord` objects, sort the list, and save it:

```
from Bio import SeqIO

records = list(SeqIO.parse("ls_orchid.fasta", "fasta"))
records.sort(key=lambda r: len(r))
SeqIO.write(records, "sorted_orchids.fasta", "fasta")
```

The only clever bit is specifying a comparison method for how to sort the records (here we sort them by length). If you wanted the longest records first, you could flip the comparison or use the reverse argument:

```
from Bio import SeqIO

records = list(SeqIO.parse("ls_orchid.fasta", "fasta"))
records.sort(key=lambda r: -len(r))
SeqIO.write(records, "sorted_orchids.fasta", "fasta")
```

Now that's pretty straight forward - but what happens if you have a very large file and you can't load it all into memory like this? For example, you might have some next-generation sequencing reads to sort by length. This can be solved using the `Bio.SeqIO.index()` function.

```
from Bio import SeqIO

# Get the lengths and ids, and sort on length
len_and_ids = sorted(
    (len(rec), rec.id) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")
)
```

(continues on next page)

(continued from previous page)

```
ids = reversed([id for (length, id) in len_and_ids])
del len_and_ids  # free this memory
record_index = SeqIO.index("ls_orchid.fasta", "fasta")
records = (record_index[id] for id in ids)
SeqIO.write(records, "sorted.fasta", "fasta")
```

First we scan through the file once using `Bio.SeqIO.parse()`, recording the record identifiers and their lengths in a list of tuples. We then sort this list to get them in length order, and discard the lengths. Using this sorted list of identifiers `Bio.SeqIO.index()` allows us to retrieve the records one by one, and we pass them to `Bio.SeqIO.write()` for output.

These examples all use `Bio.SeqIO` to parse the records into `SeqRecord` objects which are output using `Bio.SeqIO.write()`. What if you want to sort a file format which `Bio.SeqIO.write()` doesn't support, like the plain text SwissProt format? Here is an alternative solution using the `get_raw()` method added to `Bio.SeqIO.index()` in Biopython 1.54 (see Section *Getting the raw data for a record*).

```
from Bio import SeqIO

# Get the lengths and ids, and sort on length
len_and_ids = sorted(
    (len(rec), rec.id) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")
)
ids = reversed([id for (length, id) in len_and_ids])
del len_and_ids  # free this memory

record_index = SeqIO.index("ls_orchid.fasta", "fasta")
with open("sorted.fasta", "wb") as out_handle:
    for id in ids:
        out_handle.write(record_index.get_raw(id))
```

Note with Python 3 onwards, we have to open the file for writing in binary mode because the `get_raw()` method returns `bytes` objects.

As a bonus, because it doesn't parse the data into `SeqRecord` objects a second time it should be faster. If you only want to use this with FASTA format, we can speed this up one step further by using the low-level FASTA parser to get the record identifiers and lengths:

```
from Bio.SeqIO.FastaIO import SimpleFastaParser
from Bio import SeqIO

# Get the lengths and ids, and sort on length
with open("ls_orchid.fasta") as in_handle:
    len_and_ids = sorted(
        (len(seq), title.split(None, 1)[0])
        for title, seq in SimpleFastaParser(in_handle)
    )
ids = reversed([id for (length, id) in len_and_ids])
del len_and_ids  # free this memory

record_index = SeqIO.index("ls_orchid.fasta", "fasta")
with open("sorted.fasta", "wb") as out_handle:
    for id in ids:
        out_handle.write(record_index.get_raw(id))
```

23.1.6 Simple quality filtering for FASTQ files

The FASTQ file format was introduced at Sanger and is now widely used for holding nucleotide sequencing reads together with their quality scores. FASTQ files (and the related QUAL files) are an excellent example of per-letter-annotation, because for each nucleotide in the sequence there is an associated quality score. Any per-letter-annotation is held in a `SeqRecord` in the `letter_annotations` dictionary as a list, tuple or string (with the same number of elements as the sequence length).

One common task is taking a large set of sequencing reads and filtering them (or cropping them) based on their quality scores. The following example is very simplistic, but should illustrate the basics of working with quality data in a `SeqRecord` object. All we are going to do here is read in a file of FASTQ data, and filter it to pick out only those records whose PHRED quality scores are all above some threshold (here 20).

For this example we'll use some real data downloaded from the ENA sequence read archive, ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz (2MB) which unzips to a 19MB file `SRR020192.fastq`. This is some Roche 454 GS FLX single end data from virus infected California sea lions (see https://www.ebi.ac.uk/ena/data/view/SRS004476 for details).

First, let's count the reads:

```
from Bio import SeqIO

count = 0
for rec in SeqIO.parse("SRR020192.fastq", "fastq"):
    count += 1
print("%i reads" % count)
```

Now let's do a simple filtering for a minimum PHRED quality of 20:

```
from Bio import SeqIO

good_reads = (
    rec
    for rec in SeqIO.parse("SRR020192.fastq", "fastq")
    if min(rec.letter_annotations["phred_quality"]) >= 20
)
count = SeqIO.write(good_reads, "good_quality.fastq", "fastq")
print("Saved %i reads" % count)
```

This pulled out only 14580 reads out of the 41892 present. A more sensible thing to do would be to quality trim the reads, but this is intended as an example only.

FASTQ files can contain millions of entries, so it is best to avoid loading them all into memory at once. This example uses a generator expression, which means only one `SeqRecord` is created at a time - avoiding any memory limitations.

Note that it would be faster to use the low-level `FastqGeneralIterator` parser here (see Section *Low level FASTA and FASTQ parsers*), but that does not turn the quality string into integer scores.

23.1.7 Trimming off primer sequences

For this example we're going to pretend that `GATGACGGTGT` is a 5' primer sequence we want to look for in some FASTQ formatted read data. As in the example above, we'll use the `SRR020192.fastq` file downloaded from the ENA (ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz).

By using the main `Bio.SeqIO` interface, the same approach would work with any other supported file format (e.g. FASTA files). However, for large FASTQ files it would be faster the low-level `FastqGeneralIterator` parser here (see the earlier example, and Section *Low level FASTA and FASTQ parsers*).

This code uses `Bio.SeqIO` with a generator expression (to avoid loading all the sequences into memory at once), and the `Seq` object's `startswith` method to see if the read starts with the primer sequence:

```
from Bio import SeqIO

primer_reads = (
    rec
    for rec in SeqIO.parse("SRR020192.fastq", "fastq")
    if rec.seq.startswith("GATGACGGTGT")
)
count = SeqIO.write(primer_reads, "with_primer.fastq", "fastq")
print("Saved %i reads" % count)
```

That should find 13819 reads from `SRR014849.fastq` and save them to a new FASTQ file, `with_primer.fastq`.

Now suppose that instead you wanted to make a FASTQ file containing these reads but with the primer sequence removed? That's just a small change as we can slice the `SeqRecord` (see Section *Slicing a SeqRecord*) to remove the first eleven letters (the length of our primer):

```
from Bio import SeqIO

trimmed_primer_reads = (
    rec[11:]
    for rec in SeqIO.parse("SRR020192.fastq", "fastq")
    if rec.seq.startswith("GATGACGGTGT")
)
count = SeqIO.write(trimmed_primer_reads, "with_primer_trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

Again, that should pull out the 13819 reads from `SRR020192.fastq`, but this time strip off the first ten characters, and save them to another new FASTQ file, `with_primer_trimmed.fastq`.

Now, suppose you want to create a new FASTQ file where these reads have their primer removed, but all the other reads are kept as they were? If we want to still use a generator expression, it is probably clearest to define our own trim function:

```
from Bio import SeqIO


def trim_primer(record, primer):
    if record.seq.startswith(primer):
        return record[len(primer) :]
    else:
        return record
```

(continues on next page)

(continued from previous page)

```
trimmed_reads = (
    trim_primer(record, "GATGACGGTGT")
    for record in SeqIO.parse("SRR020192.fastq", "fastq")
)
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

This takes longer, as this time the output file contains all 41892 reads. Again, we're used a generator expression to avoid any memory problems. You could alternatively use a generator function rather than a generator expression.

```
from Bio import SeqIO


def trim_primers(records, primer):
    """Removes perfect primer sequences at start of reads.

    This is a generator function, the records argument should
    be a list or iterator returning SeqRecord objects.
    """
    len_primer = len(primer)  # cache this for later
    for record in records:
        if record.seq.startswith(primer):
            yield record[len_primer:]
        else:
            yield record


original_reads = SeqIO.parse("SRR020192.fastq", "fastq")
trimmed_reads = trim_primers(original_reads, "GATGACGGTGT")
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

This form is more flexible if you want to do something more complicated where only some of the records are retained – as shown in the next example.

23.1.8 Trimming off adaptor sequences

This is essentially a simple extension to the previous example. We are going to going to pretend `GATGACGGTGT` is an adaptor sequence in some FASTQ formatted read data, again the `SRR020192.fastq` file from the NCBI (ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz).

This time however, we will look for the sequence *anywhere* in the reads, not just at the very beginning:

```
from Bio import SeqIO


def trim_adaptors(records, adaptor):
    """Trims perfect adaptor sequences.

    This is a generator function, the records argument should
    be a list or iterator returning SeqRecord objects.
    """
```

(continues on next page)

(continued from previous page)

```
    len_adaptor = len(adaptor)  # cache this for later
    for record in records:
        index = record.seq.find(adaptor)
        if index == -1:
            # adaptor not found, so won't trim
            yield record
        else:
            # trim off the adaptor
            yield record[index + len_adaptor :]


original_reads = SeqIO.parse("SRR020192.fastq", "fastq")
trimmed_reads = trim_adaptors(original_reads, "GATGACGGTGT")
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

Because we are using a FASTQ input file in this example, the `SeqRecord` objects have per-letter-annotation for the quality scores. By slicing the `SeqRecord` object the appropriate scores are used on the trimmed records, so we can output them as a FASTQ file too.

Compared to the output of the previous example where we only looked for a primer/adaptor at the start of each read, you may find some of the trimmed reads are quite short after trimming (e.g. if the adaptor was found in the middle rather than near the start). So, let's add a minimum length requirement as well:

```
from Bio import SeqIO


def trim_adaptors(records, adaptor, min_len):
    """Trims perfect adaptor sequences, checks read length.

    This is a generator function, the records argument should
    be a list or iterator returning SeqRecord objects.
    """
    len_adaptor = len(adaptor)  # cache this for later
    for record in records:
        len_record = len(record)  # cache this for later
        if len(record) < min_len:
            # Too short to keep
            continue
        index = record.seq.find(adaptor)
        if index == -1:
            # adaptor not found, so won't trim
            yield record
        elif len_record - index - len_adaptor >= min_len:
            # after trimming this will still be long enough
            yield record[index + len_adaptor :]


original_reads = SeqIO.parse("SRR020192.fastq", "fastq")
trimmed_reads = trim_adaptors(original_reads, "GATGACGGTGT", 100)
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

By changing the format names, you could apply this to FASTA files instead. This code also could be extended to do a fuzzy match instead of an exact match (maybe using a pairwise alignment, or taking into account the read quality scores), but that will be much slower.

23.1.9 Converting FASTQ files

Back in Section *Converting between sequence file formats* we showed how to use `Bio.SeqIO` to convert between two file formats. Here we'll go into a little more detail regarding FASTQ files which are used in second generation DNA sequencing. Please refer to Cock *et al.* (2010) [Cock2010] for a longer description. FASTQ files store both the DNA sequence (as a string) and the associated read qualities.

PHRED scores (used in most FASTQ files, and also in QUAL files, ACE files and SFF files) have become a *de facto* standard for representing the probability of a sequencing error (here denoted by P_e) at a given base using a simple base ten log transformation:

$$Q_{\text{PHRED}} = -10 \times \log_{10}(P_e)$$

This means a wrong read ($P_e = 1$) gets a PHRED quality of 0, while a very good read like $P_e = 0.00001$ gets a PHRED quality of 50. While for raw sequencing data qualities higher than this are rare, with post processing such as read mapping or assembly, qualities of up to about 90 are possible (indeed, the MAQ tool allows for PHRED scores in the range 0 to 93 inclusive).

The FASTQ format has the potential to become a *de facto* standard for storing the letters and quality scores for a sequencing read in a single plain text file. The only fly in the ointment is that there are at least three versions of the FASTQ format which are incompatible and difficult to distinguish...

1. The original Sanger FASTQ format uses PHRED qualities encoded with an ASCII offset of 33. The NCBI are using this format in their Short Read Archive. We call this the `fastq` (or `fastq-sanger`) format in `Bio.SeqIO`.
2. Solexa (later bought by Illumina) introduced their own version using Solexa qualities encoded with an ASCII offset of 64. We call this the `fastq-solexa` format.
3. Illumina pipeline 1.3 onwards produces FASTQ files with PHRED qualities (which is more consistent), but encoded with an ASCII offset of 64. We call this the `fastq-illumina` format.

The Solexa quality scores are defined using a different log transformation:

$$Q_{\text{Solexa}} = -10 \times \log_{10}\left(\frac{P_e}{1 - P_e}\right)$$

Given Solexa/Illumina have now moved to using PHRED scores in version 1.3 of their pipeline, the Solexa quality scores will gradually fall out of use. If you equate the error estimates (P_e) these two equations allow conversion between the two scoring systems - and Biopython includes functions to do this in the `Bio.SeqIO.QualityIO` module, which are called if you use `Bio.SeqIO` to convert an old Solexa/Illumina file into a standard Sanger FASTQ file:

```
from Bio import SeqIO

SeqIO.convert("solexa.fastq", "fastq-solexa", "standard.fastq", "fastq")
```

If you want to convert a new Illumina 1.3+ FASTQ file, all that gets changed is the ASCII offset because although encoded differently the scores are all PHRED qualities:

```
from Bio import SeqIO

SeqIO.convert("illumina.fastq", "fastq-illumina", "standard.fastq", "fastq")
```

Note that using `Bio.SeqIO.convert()` like this is *much* faster than combining `Bio.SeqIO.parse()` and `Bio.SeqIO.write()` because optimized code is used for converting between FASTQ variants (and also for FASTQ to FASTA conversion).

For good quality reads, PHRED and Solexa scores are approximately equal, which means since both the `fasta-solexa` and `fastq-illumina` formats use an ASCII offset of 64 the files are almost the same. This was a deliberate design choice by Illumina, meaning applications expecting the old `fasta-solexa` style files will probably be OK using the newer `fastq-illumina` files (on good data). Of course, both variants are very different from the original FASTQ standard as used by Sanger, the NCBI, and elsewhere (format name `fastq` or `fastq-sanger`).

For more details, see the built-in help (also at *Bio.SeqIO.QualityIO*):

```
>>> from Bio.SeqIO import QualityIO
>>> help(QualityIO)
```

23.1.10 Converting FASTA and QUAL files into FASTQ files

FASTQ files hold *both* sequences and their quality strings. FASTA files hold *just* sequences, while QUAL files hold *just* the qualities. Therefore a single FASTQ file can be converted to or from *paired* FASTA and QUAL files.

Going from FASTQ to FASTA is easy:

```
from Bio import SeqIO

SeqIO.convert("example.fastq", "fastq", "example.fasta", "fasta")
```

Going from FASTQ to QUAL is also easy:

```
from Bio import SeqIO

SeqIO.convert("example.fastq", "fastq", "example.qual", "qual")
```

However, the reverse is a little more tricky. You can use `Bio.SeqIO.parse()` to iterate over the records in a *single* file, but in this case we have two input files. There are several strategies possible, but assuming that the two files are really paired the most memory efficient way is to loop over both together. The code is a little fiddly, so we provide a function called `PairedFastaQualIterator` in the `Bio.SeqIO.QualityIO` module to do this. This takes two handles (the FASTA file and the QUAL file) and returns a `SeqRecord` iterator:

```
from Bio.SeqIO.QualityIO import PairedFastaQualIterator

for record in PairedFastaQualIterator(open("example.fasta"), open("example.qual")):
    print(record)
```

This function will check that the FASTA and QUAL files are consistent (e.g. the records are in the same order, and have the same sequence length). You can combine this with the `Bio.SeqIO.write()` function to convert a pair of FASTA and QUAL files into a single FASTQ files:

```
from Bio import SeqIO
from Bio.SeqIO.QualityIO import PairedFastaQualIterator

with open("example.fasta") as f_handle, open("example.qual") as q_handle:
    records = PairedFastaQualIterator(f_handle, q_handle)
    count = SeqIO.write(records, "temp.fastq", "fastq")
print("Converted %i records" % count)
```

23.1.11 Indexing a FASTQ file

FASTQ files are usually very large, with millions of reads in them. Due to the sheer amount of data, you can't load all the records into memory at once. This is why the examples above (filtering and trimming) iterate over the file looking at just one `SeqRecord` at a time.

However, sometimes you can't use a big loop or an iterator - you may need random access to the reads. Here the `Bio.SeqIO.index()` function may prove very helpful, as it allows you to access any read in the FASTQ file by its name (see Section *Sequence files as Dictionaries – Indexed files*).

Again we'll use the `SRR020192.fastq` file from the ENA (ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz), although this is actually quite a small FASTQ file with less than 50,000 reads:

```
>>> from Bio import SeqIO
>>> fq_dict = SeqIO.index("SRR020192.fastq", "fastq")
>>> len(fq_dict)
41892
>>> list(fq_dict.keys())[:4]
['SRR020192.38240', 'SRR020192.23181', 'SRR020192.40568', 'SRR020192.23186']
>>> fq_dict["SRR020192.23186"].seq
Seq('GTCCCAGTATTCGGATTTGTCTGCCAAAACAATGAAATTGACACAGTTTACAAC...CCG')
```

When testing this on a FASTQ file with seven million reads, indexing took about a minute, but record access was almost instant.

The sister function `Bio.SeqIO.index_db()` lets you save the index to an SQLite3 database file for near instantaneous reuse - see Section *Sequence files as Dictionaries – Indexed files* for more details.

The example in Section *Sorting a sequence file* show how you can use the `Bio.SeqIO.index()` function to sort a large FASTA file – this could also be used on FASTQ files.

23.1.12 Converting SFF files

If you work with 454 (Roche) sequence data, you will probably have access to the raw data as a Standard Flowgram Format (SFF) file. This contains the sequence reads (called bases) with quality scores and the original flow information.

A common task is to convert from SFF to a pair of FASTA and QUAL files, or to a single FASTQ file. These operations are trivial using the `Bio.SeqIO.convert()` function (see Section *Converting between sequence file formats*):

```
>>> from Bio import SeqIO
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff", "reads.fasta", "fasta")
10
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff", "reads.qual", "qual")
10
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff", "reads.fastq", "fastq")
10
```

Remember the convert function returns the number of records, in this example just ten. This will give you the *untrimmed* reads, where the leading and trailing poor quality sequence or adaptor will be in lower case. If you want the *trimmed* reads (using the clipping information recorded within the SFF file) use this:

```
>>> from Bio import SeqIO
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff-trim", "trimmed.fasta", "fasta")
10
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff-trim", "trimmed.qual", "qual")
```

(continues on next page)

(continued from previous page)

```
10
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff-trim", "trimmed.fastq", "fastq")
10
```

If you run Linux, you could ask Roche for a copy of their "off instrument" tools (often referred to as the Newbler tools). This offers an alternative way to do SFF to FASTA or QUAL conversion at the command line (but currently FASTQ output is not supported), e.g.

```
$ sffinfo -seq -notrim E3MFGYR02_random_10_reads.sff > reads.fasta
$ sffinfo -qual -notrim E3MFGYR02_random_10_reads.sff > reads.qual
$ sffinfo -seq -trim E3MFGYR02_random_10_reads.sff > trimmed.fasta
$ sffinfo -qual -trim E3MFGYR02_random_10_reads.sff > trimmed.qual
```

The way Biopython uses mixed case sequence strings to represent the trimming points deliberately mimics what the Roche tools do.

For more information on the Biopython SFF support, consult the built-in help:

```
>>> from Bio.SeqIO import SffIO
>>> help(SffIO)
```

23.1.13 Identifying open reading frames

A very simplistic first step at identifying possible genes is to look for open reading frames (ORFs). By this we mean look in all six frames for long regions without stop codons – an ORF is just a region of nucleotides with no in frame stop codons.

Of course, to find a gene you would also need to worry about locating a start codon, possible promoters – and in Eukaryotes there are introns to worry about too. However, this approach is still useful in viruses and Prokaryotes.

To show how you might approach this with Biopython, we'll need a sequence to search, and as an example we'll again use the bacterial plasmid – although this time we'll start with a plain FASTA file with no pre-marked genes: NC_005816.fna. This is a bacterial sequence, so we'll want to use NCBI codon table 11 (see Section *Translation* about translation).

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.fna", "fasta")
>>> table = 11
>>> min_pro_len = 100
```

Here is a neat trick using the `Seq` object's `split` method to get a list of all the possible ORF translations in the six reading frames:

```
>>> for strand, nuc in [(+1, record.seq), (-1, record.seq.reverse_complement())]:
...     for frame in range(3):
...         length = 3 * ((len(record) - frame) // 3)  # Multiple of three
...         for pro in nuc[frame : frame + length].translate(table).split("*"):
...             if len(pro) >= min_pro_len:
...                 print(
...                     "%s...%s - length %i, strand %i, frame %i"
...                     % (pro[:30], pro[-3:], len(pro), strand, frame)
...                 )
...
```

(continues on next page)

(continued from previous page)

```
GCLMKKSSIVATIITILSGSANAASSQLIP...YRF - length 315, strand 1, frame 0
KSGELRQTPPASSTLHLRLILQRSGVMMEL...NPE - length 285, strand 1, frame 1
GLNCSFFSICNWKFIDYINRLFQIIYLCKN...YYH - length 176, strand 1, frame 1
VKKILYIKALFLCTVIKLRRFIFSVNNMKF...DLP - length 165, strand 1, frame 1
NQIQGVICSPDSGEFMVTFETVMEIKILHK...GVA - length 355, strand 1, frame 2
RRKEHVSKKRRPQKRPRRRRFFHRLRPPDE...PTR - length 128, strand 1, frame 2
TGKQNSCQMSAIWQLRQNTATKTRQNRARI...AIK - length 100, strand 1, frame 2
QGSGYAFPHASILSGIAMSHFYFLVLHAVK...CSD - length 114, strand -1, frame 0
IYSTSEHTGEQVMRTLDEVIASRSPESQTR...FHV - length 111, strand -1, frame 0
WGKLQVIGLSMWMVLFSQRFDDWLNEQEDA...ESK - length 125, strand -1, frame 1
RGIFMSDTMVVNGSGGVPAFLFSGSTLSSY...LLK - length 361, strand -1, frame 1
WDVKTVTGVLHHPFHLTFSLCPEGATQSGR...VKR - length 111, strand -1, frame 1
LSHTVTDFTDQMAQVGLCQCVNVFLDEVTG...KAA - length 107, strand -1, frame 2
RALTGLSAPGIRSQTSCDRLRELRYVPVSL...PLQ - length 119, strand -1, frame 2
```

Note that here we are counting the frames from the 5' end (start) of *each* strand. It is sometimes easier to always count from the 5' end (start) of the *forward* strand.

You could easily edit the above loop based code to build up a list of the candidate proteins, or convert this to a list comprehension. Now, one thing this code doesn't do is keep track of where the proteins are.

You could tackle this in several ways. For example, the following code tracks the locations in terms of the protein counting, and converts back to the parent sequence by multiplying by three, then adjusting for the frame and strand:

```
from Bio import SeqIO

record = SeqIO.read("NC_005816.gb", "genbank")
table = 11
min_pro_len = 100


def find_orfs_with_trans(seq, trans_table, min_protein_length):
    answer = []
    seq_len = len(seq)
    for strand, nuc in [(+1, seq), (-1, seq.reverse_complement())]:
        for frame in range(3):
            trans = nuc[frame:].translate(trans_table)
            trans_len = len(trans)
            aa_start = 0
            aa_end = 0
            while aa_start < trans_len:
                aa_end = trans.find("*", aa_start)
                if aa_end == -1:
                    aa_end = trans_len
                if aa_end - aa_start >= min_protein_length:
                    if strand == 1:
                        start = frame + aa_start * 3
                        end = min(seq_len, frame + aa_end * 3 + 3)
                    else:
                        start = seq_len - frame - aa_end * 3 - 3
                        end = seq_len - frame - aa_start * 3
                    answer.append((start, end, strand, trans[aa_start:aa_end]))
                aa_start = aa_end + 1
```

(continues on next page)

(continued from previous page)

```
    answer.sort()
    return answer


orf_list = find_orfs_with_trans(record.seq, table, min_pro_len)
for start, end, strand, pro in orf_list:
    print(
        "%s...%s - length %i, strand %i, %i:%i"
        % (pro[:30], pro[-3:], len(pro), strand, start, end)
    )
```

And the output:

```
NQIQGVICSPDSGEFMVTFETVMEIKILHK...GVA - length 355, strand 1, 41:1109
WDVKTVTGVLHHPFHLTFSLCPEGATQSGR...VKR - length 111, strand -1, 491:827
KSGELRQTPPASSTLHLRLILQRSGVMMEL...NPE - length 285, strand 1, 1030:1888
RALTGLSAPGIRSQTSCDRLRELRYVPVSL...PLQ - length 119, strand -1, 2830:3190
RRKEHVSKKRRPQKRPRRRRFFHRLRPPDE...PTR - length 128, strand 1, 3470:3857
GLNCSFFSICNWKFIDYINRLFQIIYLCKN...YYH - length 176, strand 1, 4249:4780
RGIFMSDTMVVNGSGGVPAFLFSGSTLSSY...LLK - length 361, strand -1, 4814:5900
VKKILYIKALFLCTVIKLRRFIFSVNNMKF...DLP - length 165, strand 1, 5923:6421
LSHTVTDFTDQMAQVGLCQCVNVFLDEVTG...KAA - length 107, strand -1, 5974:6298
GCLMKKSSIVATIITILSGSANAASSQLIP...YRF - length 315, strand 1, 6654:7602
IYSTSEHTGEQVMRTLDEVIASRSPESQTR...FHV - length 111, strand -1, 7788:8124
WGKLQVIGLSMWMVLFSQRFDDWLNEQEDA...ESK - length 125, strand -1, 8087:8465
TGKQNSCQMSAIWQLRQNTATKTRQNRARI...AIK - length 100, strand 1, 8741:9044
QGSGYAFPHASILSGIAMSHFYFLVLHAVK...CSD - length 114, strand -1, 9264:9609
```

If you comment out the sort statement, then the protein sequences will be shown in the same order as before, so you can check this is doing the same thing. Here we have sorted them by location to make it easier to compare to the actual annotation in the GenBank file (as visualized in Section *A nice example*).

If however all you want to find are the locations of the open reading frames, then it is a waste of time to translate every possible codon, including doing the reverse complement to search the reverse strand too. All you need to do is search for the possible stop codons (and their reverse complements). Using regular expressions is an obvious approach here (see the Python module `re`). These are an extremely powerful (but rather complex) way of describing search strings, which are supported in lots of programming languages and also command line tools like `grep` as well). You can find whole books about this topic!

23.2 Sequence parsing plus simple plots

This section shows some more examples of sequence parsing, using the `Bio.SeqIO` module described in Chapter *Sequence Input/Output*, plus the Python library matplotlib's `pylab` plotting interface (see the matplotlib website for a tutorial). Note that to follow these examples you will need matplotlib installed - but without it you can still try the data parsing bits.

23.2.1 Histogram of sequence lengths

There are lots of times when you might want to visualize the distribution of sequence lengths in a dataset – for example the range of contig sizes in a genome assembly project. In this example we'll reuse our orchid FASTA file ls_orchid.fasta which has only 94 sequences.

First of all, we will use `Bio.SeqIO` to parse the FASTA file and compile a list of all the sequence lengths. You could do this with a for loop, but I find a list comprehension more pleasing:

```
>>> from Bio import SeqIO
>>> sizes = [len(rec) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")]
>>> len(sizes), min(sizes), max(sizes)
(94, 572, 789)
>>> sizes
[740, 753, 748, 744, 733, 718, 730, 704, 740, 709, 700, 726, ..., 592]
```

Now that we have the lengths of all the genes (as a list of integers), we can use the matplotlib histogram function to display it.

```
from Bio import SeqIO

sizes = [len(rec) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")]

import pylab

pylab.hist(sizes, bins=20)
pylab.title(
    "%i orchid sequences\nLengths %i to %i" % (len(sizes), min(sizes), max(sizes))
)
pylab.xlabel("Sequence length (bp)")
pylab.ylabel("Count")
pylab.show()
```

That should pop up a new window containing the graph shown in Fig. 1. Notice that most of these orchid sequences are about 740 bp long, and there could be two distinct classes of sequence here with a subset of shorter sequences.

Tip: Rather than using `pylab.show()` to show the plot in a window, you can also use `pylab.savefig(...)` to save the figure to a file (e.g. as a PNG or PDF).

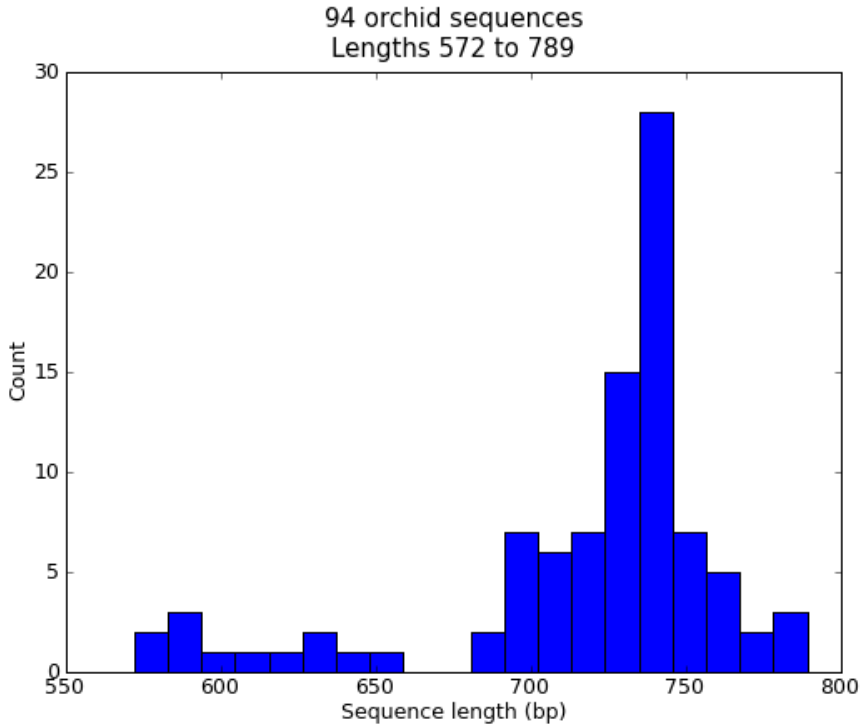


Fig. 1: Histogram of orchid sequence lengths.

23.2.2 Plot of sequence GC%

Another easily calculated quantity of a nucleotide sequence is the GC%. You might want to look at the GC% of all the genes in a bacterial genome for example, and investigate any outliers which could have been recently acquired by horizontal gene transfer. Again, for this example we'll reuse our orchid FASTA file ls_orchid.fasta.

First of all, we will use `Bio.SeqIO` to parse the FASTA file and compile a list of all the GC percentages. Again, you could do this with a for loop, but I prefer this:

```
from Bio import SeqIO
from Bio.SeqUtils import gc_fraction

gc_values = sorted(
    100 * gc_fraction(rec.seq) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")
)
```

Having read in each sequence and calculated the GC%, we then sorted them into ascending order. Now we'll take this list of floating point values and plot them with matplotlib:

```
import pylab

pylab.plot(gc_values)
pylab.title(
    "%i orchid sequences\nGC%% %0.1f to %0.1f"
    % (len(gc_values), min(gc_values), max(gc_values))
)
pylab.xlabel("Genes")
```

(continues on next page)

(continued from previous page)

```
pylab.ylabel("GC%")
pylab.show()
```

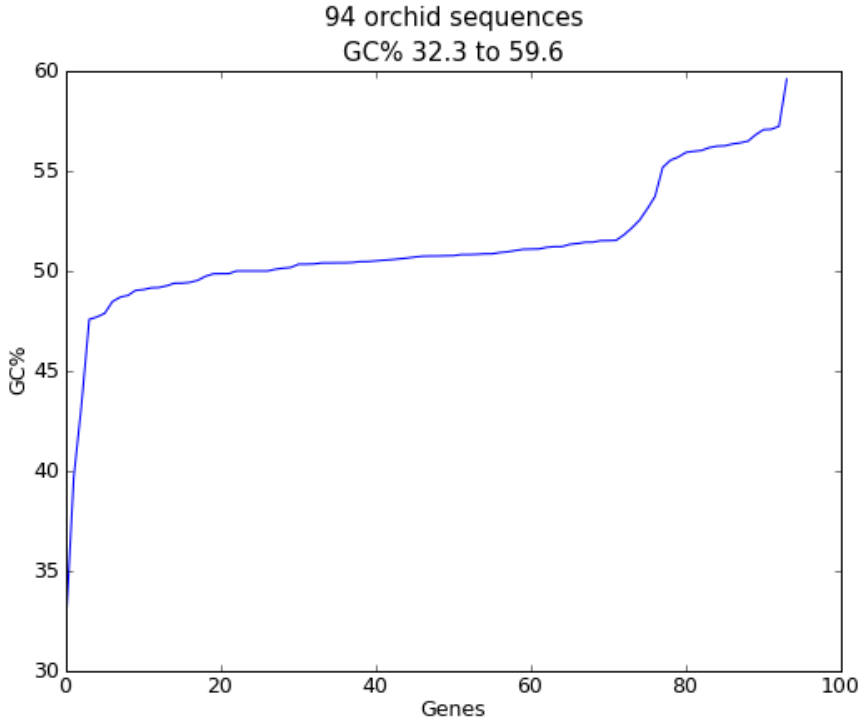


Fig. 2: Histogram of orchid sequence lengths.

As in the previous example, that should pop up a new window with the graph shown in Fig. 2. If you tried this on the full set of genes from one organism, you'd probably get a much smoother plot than this.

23.2.3 Nucleotide dot plots

A dot plot is a way of visually comparing two nucleotide sequences for similarity to each other. A sliding window is used to compare short sub-sequences to each other, often with a mismatch threshold. Here for simplicity we'll only look for perfect matches (shown in black in Fig. 3).

To start off, we'll need two sequences. For the sake of argument, we'll just take the first two from our orchid FASTA file ls_orchid.fasta:

```
from Bio import SeqIO

with open("ls_orchid.fasta") as in_handle:
    record_iterator = SeqIO.parse(in_handle, "fasta")
    rec_one = next(record_iterator)
    rec_two = next(record_iterator)
```

We're going to show two approaches. Firstly, a simple naive implementation which compares all the window sized sub-sequences to each other to compiles a similarity matrix. You could construct a matrix or array object, but here we just use a list of lists of booleans created with a nested list comprehension:

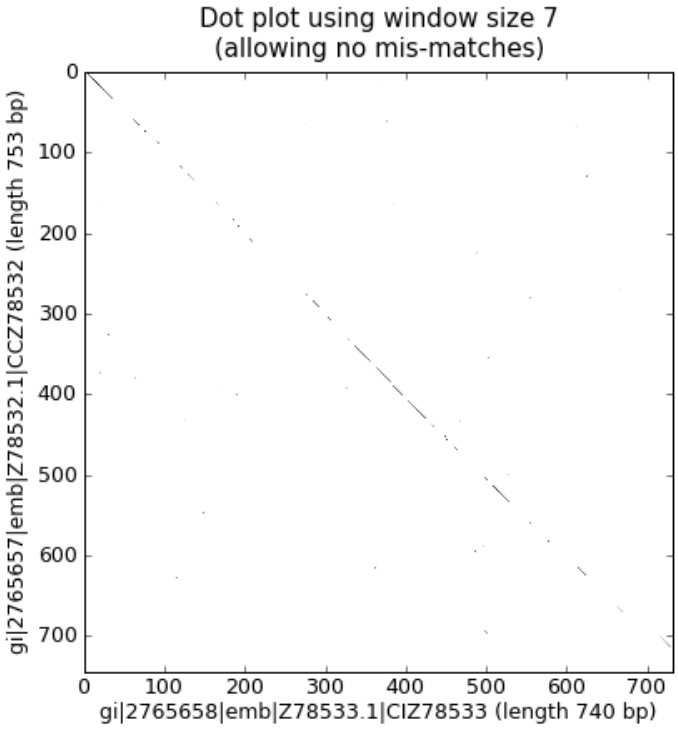


Fig. 3: Nucleotide dot plot of two orchid sequences using image show.

```
window = 7
seq_one = rec_one.seq.upper()
seq_two = rec_two.seq.upper()
data = [
    [
        (seq_one[i : i + window] != seq_two[j : j + window])
        for j in range(len(seq_one) - window)
    ]
    for i in range(len(seq_two) - window)
]
```

Note that we have *not* checked for reverse complement matches here. Now we'll use the matplotlib's `pylab.imshow()` function to display this data, first requesting the gray color scheme so this is done in black and white:

```
import pylab

pylab.gray()
pylab.imshow(data)
pylab.xlabel("%s (length %i bp)" % (rec_one.id, len(rec_one)))
pylab.ylabel("%s (length %i bp)" % (rec_two.id, len(rec_two)))
pylab.title("Dot plot using window size %i\n(allowing no mis-matches)" % window)
pylab.show()
```

That should pop up a new window showing the graph in Fig. 3. As you might have expected, these two sequences are very similar with a partial line of window sized matches along the diagonal. There are no off diagonal matches which would be indicative of inversions or other interesting events.

The above code works fine on small examples, but there are two problems applying this to larger sequences, which we will address below. First off all, this brute force approach to the all against all comparisons is very slow. Instead, we'll compile dictionaries mapping the window sized sub-sequences to their locations, and then take the set intersection to find those sub-sequences found in both sequences. This uses more memory, but is *much* faster. Secondly, the `pylab.imshow()` function is limited in the size of matrix it can display. As an alternative, we'll use the `pylab.scatter()` function.

We start by creating dictionaries mapping the window-sized sub-sequences to locations:

```
window = 7
dict_one = {}
dict_two = {}
for seq, section_dict in [
    (rec_one.seq.upper(), dict_one),
    (rec_two.seq.upper(), dict_two),
]:
    for i in range(len(seq) - window):
        section = seq[i : i + window]
        try:
            section_dict[section].append(i)
        except KeyError:
            section_dict[section] = [i]
# Now find any sub-sequences found in both sequences
matches = set(dict_one).intersection(dict_two)
print("%i unique matches" % len(matches))
```

In order to use the `pylab.scatter()` we need separate lists for the x and y coordinates:

```
# Create lists of x and y coordinates for scatter plot
x = []
y = []
for section in matches:
    for i in dict_one[section]:
        for j in dict_two[section]:
            x.append(i)
            y.append(j)
```

We are now ready to draw the revised dot plot as a scatter plot:

```
import pylab

pylab.cla()  # clear any prior graph
pylab.gray()
pylab.scatter(x, y)
pylab.xlim(0, len(rec_one) - window)
pylab.ylim(0, len(rec_two) - window)
pylab.xlabel("%s (length %i bp)" % (rec_one.id, len(rec_one)))
pylab.ylabel("%s (length %i bp)" % (rec_two.id, len(rec_two)))
pylab.title("Dot plot using window size %i\n(allowing no mis-matches)" % window)
pylab.show()
```

That should pop up a new window showing the graph in Fig. 4.

Personally I find this second plot much easier to read! Again note that we have *not* checked for reverse complement matches here – you could extend this example to do this, and perhaps plot the forward matches in one color and the reverse matches in another.

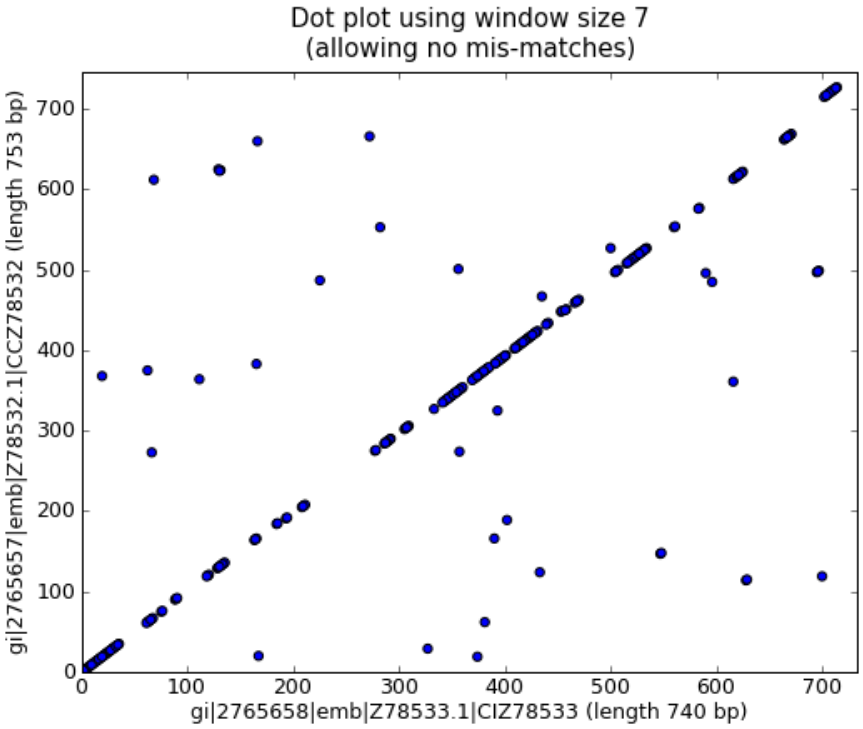


Fig. 4: Nucleotide dot plot of two orchid sequence using scatter.

23.2.4 Plotting the quality scores of sequencing read data

If you are working with second generation sequencing data, you may want to try plotting the quality data. Here is an example using two FASTQ files containing paired end reads, `SRR001666_1.fastq` for the forward reads, and `SRR001666_2.fastq` for the reverse reads. These were downloaded from the ENA sequence read archive FTP site (ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR001/SRR001666/SRR001666_1.fastq.gz and ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR001/SRR001666/SRR001666_2.fastq.gz), and are from *E. coli* – see https://www.ebi.ac.uk/ena/data/view/SRR001666 for details.

In the following code the `pylab.subplot(...)` function is used in order to show the forward and reverse qualities on two subplots, side by side. There is also a little bit of code to only plot the first fifty reads.

```
import pylab
from Bio import SeqIO

for subfigure in [1, 2]:
    filename = "SRR001666_%i.fastq" % subfigure
    pylab.subplot(1, 2, subfigure)
    for i, record in enumerate(SeqIO.parse(filename, "fastq")):
        if i >= 50:
            break  # trick!
        pylab.plot(record.letter_annotations["phred_quality"])
    pylab.ylim(0, 45)
    pylab.ylabel("PHRED quality score")
    pylab.xlabel("Position")
pylab.savefig("SRR001666.png")
print("Done")
```

You should note that we are using the `Bio.SeqIO` format name `fastq` here because the NCBI has saved these reads using the standard Sanger FASTQ format with PHRED scores. However, as you might guess from the read lengths, this data was from an Illumina Genome Analyzer and was probably originally in one of the two Solexa/Illumina FASTQ variant file formats instead.

This example uses the `pylab.savefig(...)` function instead of `pylab.show(...)`, but as mentioned before both are useful.

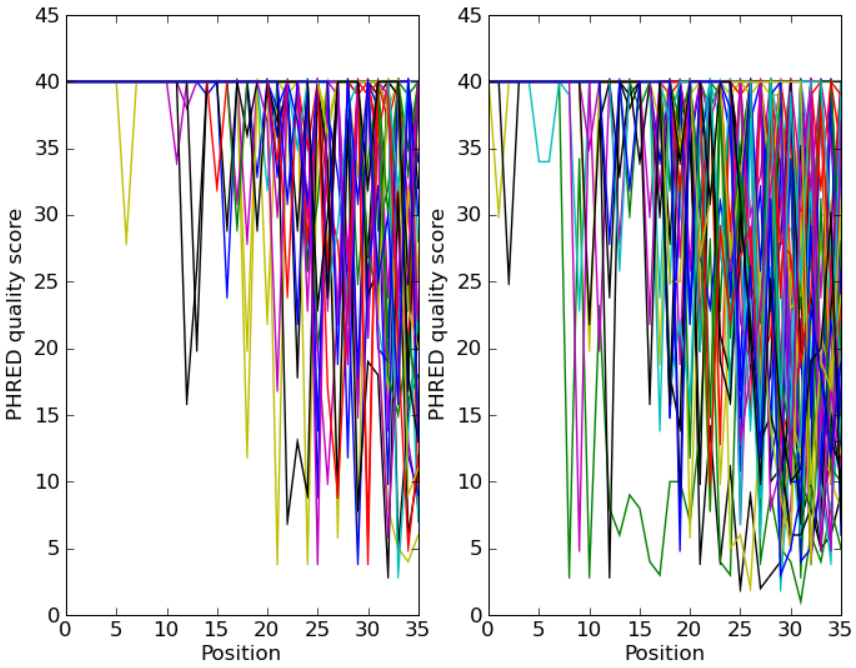


Fig. 5: Quality plot for some paired end reads.

The result is shown in Fig. 5.

23.3 BioSQL – storing sequences in a relational database

BioSQL is a joint effort between the OBF projects (BioPerl, BioJava etc) to support a shared database schema for storing sequence data. In theory, you could load a GenBank file into the database with BioPerl, then using Biopython extract this from the database as a record object with features - and get more or less the same thing as if you had loaded the GenBank file directly as a SeqRecord using `Bio.SeqIO` (Chapter *Sequence Input/Output*).

Biopython's BioSQL module is currently documented at http://biopython.org/wiki/BioSQL which is part of our wiki pages.

CHAPTER

TWENTYFOUR

THE BIOPYTHON TESTING FRAMEWORK

Biopython has a regression testing framework (the file `run_tests.py`) based on unittest, the standard unit testing framework for Python. Providing comprehensive tests for modules is one of the most important aspects of making sure that the Biopython code is as bug-free as possible before going out. It also tends to be one of the most undervalued aspects of contributing. This chapter is designed to make running the Biopython tests and writing good test code as easy as possible. Ideally, every module that goes into Biopython should have a test (and should also have documentation!). All our developers, and anyone installing Biopython from source, are strongly encouraged to run the unit tests.

24.1 Running the tests

When you download the Biopython source code, or check it out from our source code repository, you should find a subdirectory call `Tests`. This contains the key script `run_tests.py`, lots of individual scripts named `test_XXX.py`, and lots of other subdirectories which contain input files for the test suite.

As part of building and installing Biopython you will typically run the full test suite at the command line from the Biopython source top level directory using the following:

```
$ python setup.py test
```

This is actually equivalent to going to the `Tests` subdirectory and running:

```
$ python run_tests.py
```

You'll often want to run just some of the tests, and this is done like this:

```
$ python run_tests.py test_SeqIO.py test_AlignIO.py
```

When giving the list of tests, the `.py` extension is optional, so you can also just type:

```
$ python run_tests.py test_SeqIO test_AlignIO
```

To run the docstring tests (see section *Writing doctests*), you can use

```
$ python run_tests.py doctest
```

You can also skip any tests which have been setup with an explicit online component by adding `--offline`, e.g.

```
$ python run_tests.py --offline
```

By default, `run_tests.py` runs all tests, including the docstring tests.

If an individual test is failing, you can also try running it directly, which may give you more information.

Tests based on Python's standard `unittest` framework will `import unittest` and then define `unittest.TestCase` classes, each with one or more sub-tests as methods starting with `test_` which check some specific aspect of the code.

24.1.1 Running the tests using Tox

Like most Python projects, you can also use Tox to run the tests on multiple Python versions, provided they are already installed in your system.

We do not provide the configuration `tox.ini` file in our code base because of difficulties pinning down user-specific settings (e.g. executable names of the Python versions). You may also only be interested in testing Biopython only against a subset of the Python versions that we support.

If you are interested in using Tox, you could start with the example `tox.ini` shown below:

```
[tox]
envlist = pypy,py38,py39

[testenv]
changedir = Tests
commands = {envpython} run_tests.py --offline
deps =
    numpy
    reportlab
```

Using the template above, executing `tox` will test your Biopython code against PyPy, Python 3.8 and 3.9. It assumes that those Pythons' executables are named "python3.8" for Python 3.8, and so on.

24.2 Writing tests

Let's say you want to write some tests for a module called `Biospam`. This can be a module you wrote, or an existing module that doesn't have any tests yet. In the examples below, we assume that `Biospam` is a module that does simple math.

Each Biopython test consists of a script containing the test itself, and optionally a directory with input files used by the test:

1. `test_Biospam.py` – The actual test code for your module.
2. `Biospam` [optional]– A directory where any necessary input files will be located. If you have any output files that should be manually reviewed, output them here (but this is discouraged) to prevent clogging up the main Tests directory. In general, use a temporary file/folder.

Any script with a `test_` prefix in the `Tests` directory will be found and run by `run_tests.py`. Below, we show an example test script `test_Biospam.py`. If you put this script in the Biopython `Tests` directory, then `run_tests.py` will find it and execute the tests contained in it:

```
$ python run_tests.py
test_Ace ... ok
test_AlignIO ... ok
test_BioSQL ... ok
test_BioSQL_SeqIO ... ok
test_Biospam ... ok
test_CAPS ... ok
test_Clustalw ... ok
```

(continues on next page)

(continued from previous page)

```
...
----------------------------------------------------------------------
Ran 107 tests in 86.127 seconds
```

24.2.1 Writing a test using `unittest`

The `unittest`-framework has been included with Python since version 2.1, and is documented in the Python Library Reference (which I know you are keeping under your pillow, as recommended). There is also online documentation for unittest. If you are familiar with the `unittest` system (or something similar like the nose test framework), you shouldn't have any trouble. You may find looking at the existing examples within Biopython helpful too.

Here's a minimal `unittest`-style test script for `Biospam`, which you can copy and paste to get started:

```
import unittest
from Bio import Biospam


class BiospamTestAddition(unittest.TestCase):
    def test_addition1(self):
        result = Biospam.addition(2, 3)
        self.assertEqual(result, 5)

    def test_addition2(self):
        result = Biospam.addition(9, -1)
        self.assertEqual(result, 8)


class BiospamTestDivision(unittest.TestCase):
    def test_division1(self):
        result = Biospam.division(3.0, 2.0)
        self.assertAlmostEqual(result, 1.5)

    def test_division2(self):
        result = Biospam.division(10.0, -2.0)
        self.assertAlmostEqual(result, -5.0)


if __name__ == "__main__":
    runner = unittest.TextTestRunner(verbosity=2)
    unittest.main(testRunner=runner)
```

In the division tests, we use `assertAlmostEqual` instead of `assertEqual` to avoid tests failing due to roundoff errors; see the `unittest` chapter in the Python documentation for details and for other functionality available in `unittest` (online reference).

These are the key points of `unittest`-based tests:

- Test cases are stored in classes that derive from `unittest.TestCase` and cover one basic aspect of your code
- You can use methods `setUp` and `tearDown` for any repeated code which should be run before and after each test method. For example, the `setUp` method might be used to create an instance of the object you are testing, or open a file handle. The `tearDown` should do any "tidying up", for example closing the file handle.

- The tests are prefixed with `test_` and each test should cover one specific part of what you are trying to test. You can have as many tests as you want in a class.
- At the end of the test script, you can use

```
if __name__ == "__main__":
    runner = unittest.TextTestRunner(verbosity=2)
    unittest.main(testRunner=runner)
```

to execute the tests when the script is run by itself (rather than imported from `run_tests.py`). If you run this script, then you'll see something like the following:

```
$ python test_BiospamMyModule.py
test_addition1 (__main__.TestAddition) ... ok
test_addition2 (__main__.TestAddition) ... ok
test_division1 (__main__.TestDivision) ... ok
test_division2 (__main__.TestDivision) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.059s

OK
```

- To indicate more clearly what each test is doing, you can add docstrings to each test. These are shown when running the tests, which can be useful information if a test is failing.

```
import unittest
from Bio import Biospam


class BiospamTestAddition(unittest.TestCase):
    def test_addition1(self):
        """An addition test"""
        result = Biospam.addition(2, 3)
        self.assertEqual(result, 5)

    def test_addition2(self):
        """A second addition test"""
        result = Biospam.addition(9, -1)
        self.assertEqual(result, 8)


class BiospamTestDivision(unittest.TestCase):
    def test_division1(self):
        """Now let's check division"""
        result = Biospam.division(3.0, 2.0)
        self.assertAlmostEqual(result, 1.5)

    def test_division2(self):
        """A second division test"""
        result = Biospam.division(10.0, -2.0)
        self.assertAlmostEqual(result, -5.0)
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    runner = unittest.TextTestRunner(verbosity=2)
    unittest.main(testRunner=runner)
```

Running the script will now show you:

```
$ python test_BiospamMyModule.py
An addition test ... ok
A second addition test ... ok
Now let's check division ... ok
A second division test ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
```

If your module contains docstring tests (see section *Writing doctests*), you *may* want to include those in the tests to be run. You can do so as follows by modifying the code under `if __name__ == "__main__":` to look like this:

```
if __name__ == "__main__":
    unittest_suite = unittest.TestLoader().loadTestsFromName("test_Biospam")
    doctest_suite = doctest.DocTestSuite(Biospam)
    suite = unittest.TestSuite((unittest_suite, doctest_suite))
    runner = unittest.TextTestRunner(sys.stdout, verbosity=2)
    runner.run(suite)
```

This is only relevant if you want to run the docstring tests when you execute `python test_Biospam.py` if it has some complex run-time dependency checking.

In general instead include the docstring tests by adding them to the `run_tests.py` as explained below.

24.3 Writing doctests

Python modules, classes and functions support built-in documentation using docstrings. The doctest framework (included with Python) allows the developer to embed working examples in the docstrings, and have these examples automatically tested.

Currently only part of Biopython includes doctests. The `run_tests.py` script takes care of running the doctests. For this purpose, at the top of the `run_tests.py` script is a manually compiled list of modules to skip, important where optional external dependencies which may not be installed (e.g. the Reportlab and NumPy libraries). So, if you've added some doctests to the docstrings in a Biopython module, in order to have them excluded in the Biopython test suite, you must update `run_tests.py` to include your module. Currently, the relevant part of `run_tests.py` looks as follows:

```
# Following modules have historic failures. If you fix one of these
# please remove here!
EXCLUDE_DOCTEST_MODULES = [
    "Bio.PDB",
    "Bio.PDB.AbstractPropertyMap",
    "Bio.Phylo.Applications._Fasttree",
    "Bio.Phylo._io",
```

(continues on next page)

(continued from previous page)

```
    "Bio.Phylo.TreeConstruction",
    "Bio.Phylo._utils",
]

# Exclude modules with online activity
# They are not excluded by default, use --offline to exclude them
ONLINE_DOCTEST_MODULES = ["Bio.Entrez", "Bio.ExPASy", "Bio.TogoWS"]

# Silently ignore any doctests for modules requiring numpy!
if numpy is None:
    EXCLUDE_DOCTEST_MODULES.extend(
        [
            "Bio.Affy.CelFile",
            "Bio.Cluster",
            # ...
        ]
    )
```

Note that we regard doctests primarily as documentation, so you should stick to typical usage. Generally complicated examples dealing with error conditions and the like would be best left to a dedicated unit test.

Note that if you want to write doctests involving file parsing, defining the file location complicates matters. Ideally use relative paths assuming the code will be run from the `Tests` directory, see the `Bio.SeqIO` doctests for an example of this.

To run the docstring tests only, use

```
$ python run_tests.py doctest
```

Note that the doctest system is fragile and care is needed to ensure your output will match on all the different versions of Python that Biopython supports (e.g. differences in floating point numbers).

24.4 Writing doctests in the Tutorial

This Tutorial you are reading has a lot of code snippets, which are often formatted like a doctest. We have our own system in file `test_Tutorial.py` to allow tagging code snippets in the Tutorial source to be run as Python doctests. This works by adding special `.. doctest` comment lines before each Python Console (pycon) block, e.g.

```
.. doctest

.. code:: pycon

   >>> from Bio.Seq import Seq
   >>> s = Seq("ACGT")
   >>> len(s)
   4
```

Often code examples are not self-contained, but continue from the previous Python block. Here we use the magic comment `.. cont-doctest` as shown here:

```
.. cont-doctest
```

(continues on next page)

(continued from previous page)

```
.. code:: pycon

   >>> s == "ACGT"
   True
```

The special `.. doctest` comment line can take a working directory (relative to the `Doc/` folder) to use if you have any example data files, e.g. `.. doctest examples` will use the `Doc/examples` folder, while `.. doctest ../Tests/GenBank` will use the `Tests/GenBank` folder.

After the directory argument, you can specify any Python dependencies which must be present in order to run the test by adding `lib:XXX` to indicate `import XXX` must work, e.g. `.. doctest examples lib:numpy`

You can run the Tutorial doctests via:

```
$ python test_Tutorial.py
```

or:

```
$ python run_tests.py test_Tutorial.py
```

CHAPTER

TWENTYFIVE

WHERE TO GO FROM HERE – CONTRIBUTING TO BIOPYTHON

25.1 Bug Reports + Feature Requests

Getting feedback on the Biopython modules is very important to us. Open-source projects like this benefit greatly from feedback, bug-reports (and patches!) from a wide variety of contributors.

The main forums for discussing feature requests and potential bugs are the Biopython mailing list and issues or pull requests on GitHub.

Additionally, if you think you've found a new bug, you can submit it to our issue tracker at https://github.com/biopython/biopython/issues (this replaced the older Open Bioinformatics Foundation hosted RedMine tracker). This way, it won't get buried in anyone's Inbox and forgotten about.

25.2 Mailing lists and helping newcomers

We encourage all our uses to sign up to the main Biopython mailing list. Once you've got the hang of an area of Biopython, we'd encourage you to help answer questions from beginners. After all, you were a beginner once.

25.3 Contributing Documentation

We're happy to take feedback or contributions - either via a bug-report or on the Mailing List. While reading this tutorial, perhaps you noticed some topics you were interested in which were missing, or not clearly explained. There is also Biopython's built-in documentation (the docstrings, these are also online), where again, you may be able to help fill in any blanks.

25.4 Contributing cookbook examples

As explained in Chapter *Cookbook – Cool things to do with it*, Biopython now has a wiki collection of user contributed "cookbook" examples, http://biopython.org/wiki/Category:Cookbook – maybe you can add to this?

25.5 Maintaining a distribution for a platform

We currently provide source code archives (suitable for any OS, if you have the right build tools installed), and pre-compiled wheels via https://github.com/biopython/biopython-wheels to cover the major operating systems.

Most major Linux distributions have volunteers who take these source code releases, and compile them into packages for Linux users to easily install (taking care of dependencies etc). This is really great and we are of course very grateful. If you would like to contribute to this work, please find out more about how your Linux distribution handles this. There is a similar process for conda packages via https://github.com/conda-forge/biopython-feedstock thanks to the conda-forge team.

Below are some tips for certain platforms to maybe get people started with helping out:

Windows
: – You must first make sure you have a C compiler on your Windows computer, and that you can compile and install things (this is the hard bit - see the Biopython installation instructions for info on how to do this).

RPMs
: – RPMs are pretty popular package systems on some Linux platforms. There is lots of documentation on RPMs available at http://www.rpm.org to help you get started with them. To create an RPM for your platform is really easy. You just need to be able to build the package from source (having a C compiler that works is thus essential) – see the Biopython installation instructions for more info on this.

  To make the RPM, you just need to do:

```
$ python setup.py bdist_rpm
```

  This will create an RPM for your specific platform and a source RPM in the directory `dist`. This RPM should be good and ready to go, so this is all you need to do! Nice and easy.

Macintosh
: – Since Apple moved to Mac OS X, things have become much easier on the Mac. We generally treat it as just another Unix variant, and installing Biopython from source is just as easy as on Linux. The easiest way to get all the GCC compilers etc installed is to install Apple's X-Code. We might be able to provide click and run installers for Mac OS X, but to date there hasn't been any demand.

Once you've got a package, please test it on your system to make sure it installs everything in a good way and seems to work properly. Once you feel good about it, make a pull request on GitHub and write to our Biopython mailing list. You've done it. Thanks!

25.6 Contributing Unit Tests

Even if you don't have any new functionality to add to Biopython, but you want to write some code, please consider extending our unit test coverage. We've devoted all of Chapter *The Biopython testing framework* to this topic.

25.7 Contributing Code

There are no barriers to joining Biopython code development other than an interest in creating biology-related code in Python. The best place to express an interest is on the Biopython mailing lists – just let us know you are interested in coding and what kind of stuff you want to work on. Normally, we try to have some discussion on modules before coding them, since that helps generate good ideas – then just feel free to jump right in and start coding!

The main Biopython release tries to be fairly uniform and interworkable, to make it easier for users. You can read about some of (fairly informal) coding style guidelines we try to use in Biopython in the contributing documentation at http://biopython.org/wiki/Contributing. We also try to add code to the distribution along with tests (see Chapter *The Biopython testing framework* for more info on the regression testing framework) and documentation, so that everything can stay as workable and well documented as possible (including docstrings). This is, of course, the most ideal situation, under many situations you'll be able to find other people on the list who will be willing to help add documentation or more tests for your code once you make it available. So, to end this paragraph like the last, feel free to start working!

Please note that to make a code contribution you must have the legal right to contribute it and license it under the Biopython license. If you wrote it all yourself, and it is not based on any other code, this shouldn't be a problem. However, there are issues if you want to contribute a derivative work - for example something based on GPL or LPGL licensed code would not be compatible with our license. If you have any queries on this, please discuss the issue on the mailing list or GitHub.

Another point of concern for any additions to Biopython regards any build time or run time dependencies. Generally speaking, writing code to interact with a standalone tool (like BLAST, EMBOSS or ClustalW) doesn't present a big problem. However, any dependency on another library - even a Python library (especially one needed in order to compile and install Biopython like NumPy) would need further discussion.

Additionally, if you have code that you don't think fits in the distribution, but that you want to make available, we maintain Script Central (http://biopython.org/wiki/Scriptcentral) which has pointers to freely available code in Python for bioinformatics.

Hopefully this documentation has got you excited enough about Biopython to try it out (and most importantly, contribute!). Thanks for reading all the way through!

CHAPTER

TWENTYSIX

APPENDIX: USEFUL STUFF ABOUT PYTHON

If you haven't spent a lot of time programming in Python, many questions and problems that come up in using Biopython are often related to Python itself. This section tries to present some ideas and code that come up often (at least for us!) while using the Biopython libraries. If you have any suggestions for useful pointers that could go here, please contribute!

26.1 What the heck is a handle?

Handles are mentioned quite frequently throughout this documentation, and are also fairly confusing (at least to me!). Basically, you can think of a handle as being a "wrapper" around text information.

Handles provide (at least) two benefits over plain text information:

1. They provide a standard way to deal with information stored in different ways. The text information can be in a file, or in a string stored in memory, or the output from a command line program, or at some remote website, but the handle provides a common way of dealing with information in all of these formats.
2. They allow text information to be read incrementally, instead of all at once. This is really important when you are dealing with huge text files which would use up all of your memory if you had to load them all.

Handles can deal with text information that is being read (e. g. reading from a file) or written (e. g. writing information to a file). In the case of a "read" handle, commonly used functions are `read()`, which reads the entire text information from the handle, and `readline()`, which reads information one line at a time. For "write" handles, the function `write()` is regularly used.

The most common usage for handles is reading information from a file, which is done using the built-in Python function `open`. Here, we handle to the file `m_cold.fasta` which you can download here (or find included in the Biopython source code as `Doc/examples/m_cold.fasta`).

```
>>> handle = open("m_cold.fasta", "r")
>>> handle.readline()
">gi|8332116|gb|BE037100.1|BE037100 MP14H09 MP Mesembryanthemum ...\n"
```

Handles are regularly used in Biopython for passing information to parsers. For example, since Biopython 1.54 the main functions in `Bio.SeqIO` and `Bio.AlignIO` have allowed you to use a filename instead of a handle:

```
from Bio import SeqIO

for record in SeqIO.parse("m_cold.fasta", "fasta"):
    print(record.id, len(record))
```

On older versions of Biopython you had to use a handle, e.g.

```
from Bio import SeqIO

handle = open("m_cold.fasta", "r")
for record in SeqIO.parse(handle, "fasta"):
    print(record.id, len(record))
handle.close()
```

This pattern is still useful - for example suppose you have a gzip compressed FASTA file you want to parse:

```
import gzip
from Bio import SeqIO

handle = gzip.open("m_cold.fasta.gz", "rt")
for record in SeqIO.parse(handle, "fasta"):
    print(record.id, len(record))
handle.close()
```

With our parsers for plain text files, it is essential to use gzip in text mode (the default is binary mode).

See Section *Parsing sequences from compressed files* for more examples like this, including reading bzip2 compressed files.

26.1.1 Creating a handle from a string

One useful thing is to be able to turn information contained in a string into a handle. The following example shows how to do this using `StringIO` from the Python standard library:

```
>>> my_info = "A string\n with multiple lines."
>>> print(my_info)
A string
 with multiple lines.
>>> from io import StringIO
>>> my_info_handle = StringIO(my_info)
>>> first_line = my_info_handle.readline()
>>> print(first_line)
A string

>>> second_line = my_info_handle.readline()
>>> print(second_line)
 with multiple lines.
```

CHAPTER

TWENTYSEVEN

BIBLIOGRAPHY

Part II

API documentation

This content is automatically extracted from the "docstrings" within the Biopython source code, using Sphinx with autodoc. You can view this documentation within an interactive Python session using the `help()` command.

CHAPTER

TWENTYEIGHT

BIO PACKAGE

28.1 Subpackages

28.1.1 Bio.Affy package

Submodules

Bio.Affy.CelFile module

Reading information from Affymetrix CEL files version 3 and 4.

exception Bio.Affy.CelFile.**ParserError**(**args*)

Bases: ValueError

Affymetrix parser error.

__init__(**args*)

Initialise class.

class Bio.Affy.CelFile.**Record**

Bases: object

Stores the information in a cel file.

Example usage:

```
>>> from Bio.Affy import CelFile
>>> with open("Affy/affy_v3_example.CEL") as handle:
...     c = CelFile.read(handle)
...
>>> print(c.ncols, c.nrows)
5 5
>>> print(c.intensities)
[[  234.   170. 22177.   164. 22104.]
 [  188.   188. 21871.   168. 21883.]
 [  188.   193. 21455.   198. 21300.]
 [  188.   182. 21438.   188. 20945.]
 [  193. 20370.   174. 20605.   168.]]
>>> print(c.stdevs)
[[  24.    34.5 2669.    19.7 3661.2]
 [  29.8   29.8 2795.9   67.9 2792.4]
 [  29.8   88.7 2976.5   62.  2914.5]
```

(continues on next page)

(continued from previous page)

```
 [  29.8   76.2 2759.5   49.2 2762. ]
 [  38.8 2611.8   26.6 2810.7   24.1]]
>>> print(c.npix)
[[25 25 25 25 25]
 [25 25 25 25 25]
 [25 25 25 25 25]
 [25 25 25 25 25]
 [25 25 25 25 25]]
```

__init__()

Initialize the class.

Bio.Affy.CelFile.**read**(*handle*, *version=None*)

Read Affymetrix CEL file and return Record object.

CEL files format versions 3 and 4 are supported. Please specify the CEL file format as 3 or 4 if known for the version argument. If the version number is not specified, the parser will attempt to detect the version from the file contents.

The Record object returned by this function stores the intensities from the CEL file in record.intensities. Currently, record.mask and record.outliers are not set in when parsing version 4 CEL files.

Example Usage:

```
>>> from Bio.Affy import CelFile
>>> with open("Affy/affy_v3_example.CEL") as handle:
...     record = CelFile.read(handle)
...
>>> record.version == 3
True
>>> print("%i by %i array" % record.intensities.shape)
5 by 5 array
```

```
>>> with open("Affy/affy_v4_example.CEL", "rb") as handle:
...     record = CelFile.read(handle, version=4)
...
>>> record.version == 4
True
>>> print("%i by %i array" % record.intensities.shape)
5 by 5 array
```

Module contents

Deal with Affymetrix related data such as cel files.

28.1.2 Bio.Align package

Subpackages

Bio.Align.Applications package

Module contents

Alignment command line tool wrappers (OBSOLETE).

We have decided to remove this module in future, and instead recommend building your command and invoking it via the subprocess module directly.

class Bio.Align.Applications.**MuscleCommandline**(*cmd='muscle'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for the multiple alignment program MUSCLE.

http://www.drive5.com/muscle/

Notes

Last checked against version: 3.7, briefly against 3.8

References

Edgar, Robert C. (2004), MUSCLE: multiple sequence alignment with high accuracy and high throughput, Nucleic Acids Research 32(5), 1792-97.

Edgar, R.C. (2004) MUSCLE: a multiple sequence alignment method with reduced time and space complexity. BMC Bioinformatics 5(1): 113.

Examples

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> muscle_exe = r"C:\Program Files\Alignments\muscle3.8.31_i86win32.exe"
>>> in_file = r"C:\My Documents\unaligned.fasta"
>>> out_file = r"C:\My Documents\aligned.fasta"
>>> muscle_cline = MuscleCommandline(muscle_exe, input=in_file, out=out_file)
>>> print(muscle_cline)
"C:\Program Files\Alignments\muscle3.8.31_i86win32.exe" -in "C:\My Documents\
↪unaligned.fasta" -out "C:\My Documents\aligned.fasta"
```

You would typically run the command line with muscle_cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='muscle'*, ***kwargs*)

Initialize the class.

property anchors

Use anchor optimisation in tree dependent refinement iterations

This property controls the addition of the -anchors switch, treat this property as a boolean.

property anchorspacing

Minimum spacing between anchor columns

This controls the addition of the -anchorspacing parameter and its associated value. Set this property to the argument value required.

property brenner

Use Steve Brenner's root alignment method

This property controls the addition of the -brenner switch, treat this property as a boolean.

property center

Center parameter - should be negative

This controls the addition of the -center parameter and its associated value. Set this property to the argument value required.

property cluster

Perform fast clustering of input sequences, use -tree1 to save tree

This property controls the addition of the -cluster switch, treat this property as a boolean.

property cluster1

Clustering method used in iteration 1

This controls the addition of the -cluster1 parameter and its associated value. Set this property to the argument value required.

property cluster2

Clustering method used in iteration 2

This controls the addition of the -cluster2 parameter and its associated value. Set this property to the argument value required.

property clw

Write output in CLUSTALW format (with a MUSCLE header)

This property controls the addition of the -clw switch, treat this property as a boolean.

property clwout

Write CLUSTALW output (with MUSCLE header) to specified filename

This controls the addition of the -clwout parameter and its associated value. Set this property to the argument value required.

property clwstrict

Write output in CLUSTALW format with version 1.81 header

This property controls the addition of the -clwstrict switch, treat this property as a boolean.

property clwstrictout

Write CLUSTALW output (with version 1.81 header) to specified filename

This controls the addition of the -clwstrictout parameter and its associated value. Set this property to the argument value required.

property core

Do not catch exceptions

This property controls the addition of the -core switch, treat this property as a boolean.

property diagbreak

Maximum distance between two diagonals that allows them to merge into one diagonal

This controls the addition of the -diagbreak parameter and its associated value. Set this property to the argument value required.

property diaglength

Minimum length of diagonal

This controls the addition of the -diaglength parameter and its associated value. Set this property to the argument value required.

property diagmargin

Discard this many positions at ends of diagonal

This controls the addition of the -diagmargin parameter and its associated value. Set this property to the argument value required.

property diags

Find diagonals (faster for similar sequences)

This property controls the addition of the -diags switch, treat this property as a boolean.

property dimer

Use faster (slightly less accurate) dimer approximationfor the SP score

This property controls the addition of the -dimer switch, treat this property as a boolean.

property distance1

Distance measure for iteration 1

This controls the addition of the -distance1 parameter and its associated value. Set this property to the argument value required.

property distance2

Distance measure for iteration 2

This controls the addition of the -distance2 parameter and its associated value. Set this property to the argument value required.

property fasta

Write output in FASTA format

This property controls the addition of the -fasta switch, treat this property as a boolean.

property fastaout

Write FASTA format output to specified filename

This controls the addition of the -fastaout parameter and its associated value. Set this property to the argument value required.

property gapextend

Gap extension penalty

This controls the addition of the -gapextend parameter and its associated value. Set this property to the argument value required.

property gapopen

Gap open score - negative number

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property group

Group similar sequences in output

This property controls the addition of the -group switch, treat this property as a boolean.

property html

Write output in HTML format

This property controls the addition of the -html switch, treat this property as a boolean.

property htmlout

Write HTML output to specified filename

This controls the addition of the -htmlout parameter and its associated value. Set this property to the argument value required.

property hydro

Window size for hydrophobic region

This controls the addition of the -hydro parameter and its associated value. Set this property to the argument value required.

property hydrofactor

Multiplier for gap penalties in hydrophobic regions

This controls the addition of the -hydrofactor parameter and its associated value. Set this property to the argument value required.

property in1

First input filename for profile alignment

This controls the addition of the -in1 parameter and its associated value. Set this property to the argument value required.

property in2

Second input filename for a profile alignment

This controls the addition of the -in2 parameter and its associated value. Set this property to the argument value required.

property input

Input filename

This controls the addition of the -in parameter and its associated value. Set this property to the argument value required.

property le

Use log-expectation profile score (VTML240)

This property controls the addition of the -le switch, treat this property as a boolean.

property log

Log file name

This controls the addition of the -log parameter and its associated value. Set this property to the argument value required.

property loga

Log file name (append to existing file)

This controls the addition of the -loga parameter and its associated value. Set this property to the argument value required.

property matrix

path to NCBI or WU-BLAST format protein substitution matrix - also set -gapopen, -gapextend and -center

This controls the addition of the -matrix parameter and its associated value. Set this property to the argument value required.

property maxdiagbreak

Deprecated in v3.8, use -diagbreak instead.

This controls the addition of the -maxdiagbreak parameter and its associated value. Set this property to the argument value required.

property maxhours

Maximum time to run in hours

This controls the addition of the -maxhours parameter and its associated value. Set this property to the argument value required.

property maxiters

Maximum number of iterations

This controls the addition of the -maxiters parameter and its associated value. Set this property to the argument value required.

property maxtrees

Maximum number of trees to build in iteration 2

This controls the addition of the -maxtrees parameter and its associated value. Set this property to the argument value required.

property minbestcolscore

Minimum score a column must have to be an anchor

This controls the addition of the -minbestcolscore parameter and its associated value. Set this property to the argument value required.

property minsmoothscore

Minimum smoothed score a column must have to be an anchor

This controls the addition of the -minsmoothscore parameter and its associated value. Set this property to the argument value required.

property msf

Write output in MSF format

This property controls the addition of the -msf switch, treat this property as a boolean.

property msfout

Write MSF format output to specified filename

This controls the addition of the -msfout parameter and its associated value. Set this property to the argument value required.

property noanchors

Do not use anchor optimisation in tree dependent refinement iterations

This property controls the addition of the -noanchors switch, treat this property as a boolean.

property nocore

Catch exceptions

This property controls the addition of the -nocore switch, treat this property as a boolean.

property objscore

Objective score used by tree dependent refinement

This controls the addition of the -objscore parameter and its associated value. Set this property to the argument value required.

property out

Output filename

This controls the addition of the -out parameter and its associated value. Set this property to the argument value required.

property phyi

Write output in PHYLIP interleaved format

This property controls the addition of the -phyi switch, treat this property as a boolean.

property phyiout

Write PHYLIP interleaved output to specified filename

This controls the addition of the -phyiout parameter and its associated value. Set this property to the argument value required.

property phys

Write output in PHYLIP sequential format

This property controls the addition of the -phys switch, treat this property as a boolean.

property physout

Write PHYLIP sequential format to specified filename

This controls the addition of the -physout parameter and its associated value. Set this property to the argument value required.

property profile

Perform a profile alignment

This property controls the addition of the -profile switch, treat this property as a boolean.

property quiet

Do not display progress messages

This property controls the addition of the -quiet switch, treat this property as a boolean.

property refine

Only do tree dependent refinement

This property controls the addition of the -refine switch, treat this property as a boolean.

property refinew

Only do tree dependent refinement using sliding window approach

This property controls the addition of the -refinew switch, treat this property as a boolean.

property refinewindow

Length of window for -refinew

This controls the addition of the -refinewindow parameter and its associated value. Set this property to the argument value required.

property root1

Method used to root tree in iteration 1

This controls the addition of the -root1 parameter and its associated value. Set this property to the argument value required.

property root2

Method used to root tree in iteration 2

This controls the addition of the -root2 parameter and its associated value. Set this property to the argument value required.

property scorefile

Score file name, contains one line for each column in the alignment with average BLOSUM62 score

This controls the addition of the -scorefile parameter and its associated value. Set this property to the argument value required.

property seqtype

Sequence type

This controls the addition of the -seqtype parameter and its associated value. Set this property to the argument value required.

property smoothscoreceil

Maximum value of column score for smoothing

This controls the addition of the -smoothscoreceil parameter and its associated value. Set this property to the argument value required.

property smoothwindow

Window used for anchor column smoothing

This controls the addition of the -smoothwindow parameter and its associated value. Set this property to the argument value required.

property sp

Use sum-of-pairs protein profile score (PAM200)

This property controls the addition of the -sp switch, treat this property as a boolean.

property spn

Use sum-of-pairs protein nucleotide profile score

This property controls the addition of the -spn switch, treat this property as a boolean.

property spscore

Compute SP objective score of multiple alignment

This controls the addition of the -spscore parameter and its associated value. Set this property to the argument value required.

property stable

Do not group similar sequences in output (not supported in v3.8)

This property controls the addition of the -stable switch, treat this property as a boolean.

property sueff

Constant used in UPGMB clustering

This controls the addition of the -sueff parameter and its associated value. Set this property to the argument value required.

property sv

Use sum-of-pairs profile score (VTML240)

This property controls the addition of the -sv switch, treat this property as a boolean.

property tree1

Save Newick tree from iteration 1

This controls the addition of the -tree1 parameter and its associated value. Set this property to the argument value required.

property tree2

Save Newick tree from iteration 2

This controls the addition of the -tree2 parameter and its associated value. Set this property to the argument value required.

property usetree

Use given Newick tree as guide tree

This controls the addition of the -usetree parameter and its associated value. Set this property to the argument value required.

property verbose

Write parameter settings and progress

This property controls the addition of the -verbose switch, treat this property as a boolean.

property version

Write version string to stdout and exit

This property controls the addition of the -version switch, treat this property as a boolean.

property weight1

Weighting scheme used in iteration 1

This controls the addition of the -weight1 parameter and its associated value. Set this property to the argument value required.

property weight2

Weighting scheme used in iteration 2

This controls the addition of the -weight2 parameter and its associated value. Set this property to the argument value required.

class Bio.Align.Applications.**ClustalwCommandline**(*cmd='clustalw'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for clustalw (version one or two).

http://www.clustal.org/

Notes

Last checked against versions: 1.83 and 2.1

References

Larkin MA, Blackshields G, Brown NP, Chenna R, McGettigan PA, McWilliam H, Valentin F, Wallace IM, Wilm A, Lopez R, Thompson JD, Gibson TJ, Higgins DG. (2007). Clustal W and Clustal X version 2.0. Bioinformatics, 23, 2947-2948.

Examples

```
>>> from Bio.Align.Applications import ClustalwCommandline
>>> in_file = "unaligned.fasta"
>>> clustalw_cline = ClustalwCommandline("clustalw2", infile=in_file)
>>> print(clustalw_cline)
clustalw2 -infile=unaligned.fasta
```

You would typically run the command line with clustalw_cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='clustalw'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property align

Do full multiple alignment.

This property controls the addition of the -align switch, treat this property as a boolean.

property bootlabels

Node OR branch position of bootstrap values in tree display

This controls the addition of the -bootlabels parameter and its associated value. Set this property to the argument value required.

property bootstrap

Bootstrap a NJ tree (n= number of bootstraps; def. = 1000).

This controls the addition of the -bootstrap parameter and its associated value. Set this property to the argument value required.

property case

LOWER or UPPER (for GDE output only)

This controls the addition of the -case parameter and its associated value. Set this property to the argument value required.

property check

Outline the command line params.

This property controls the addition of the -check switch, treat this property as a boolean.

property clustering

NJ or UPGMA

This controls the addition of the -clustering parameter and its associated value. Set this property to the argument value required.

property convert

Output the input sequences in a different file format.

This property controls the addition of the -convert switch, treat this property as a boolean.

property dnamatrix

DNA weight matrix=IUB, CLUSTALW or filename

This controls the addition of the -dnamatrix parameter and its associated value. Set this property to the argument value required.

property endgaps

No end gap separation pen.

This property controls the addition of the -endgaps switch, treat this property as a boolean.

property fullhelp

Output full help content.

This property controls the addition of the -fullhelp switch, treat this property as a boolean.

property gapdist

Gap separation pen. range

This controls the addition of the -gapdist parameter and its associated value. Set this property to the argument value required.

property gapext

Gap extension penalty

This controls the addition of the -gapext parameter and its associated value. Set this property to the argument value required.

property gapopen

Gap opening penalty

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property helixendin

Number of residues inside helix to be treated as terminal

This controls the addition of the -helixendin parameter and its associated value. Set this property to the argument value required.

property helixendout

Number of residues outside helix to be treated as terminal

This controls the addition of the -helixendout parameter and its associated value. Set this property to the argument value required.

property helixgap

Gap penalty for helix core residues

This controls the addition of the -helixgap parameter and its associated value. Set this property to the argument value required.

property help

Outline the command line params.

This property controls the addition of the -help switch, treat this property as a boolean.

property hgapresidues

List hydrophilic res.

This property controls the addition of the -hgapresidues switch, treat this property as a boolean.

property infile

Input sequences.

This controls the addition of the -infile parameter and its associated value. Set this property to the argument value required.

property iteration

NONE or TREE or ALIGNMENT

This controls the addition of the -iteration parameter and its associated value. Set this property to the argument value required.

property kimura

Use Kimura's correction.

This property controls the addition of the -kimura switch, treat this property as a boolean.

property ktuple

Word size

This controls the addition of the -ktuple parameter and its associated value. Set this property to the argument value required.

property loopgap

Gap penalty for loop regions

This controls the addition of the -loopgap parameter and its associated value. Set this property to the argument value required.

property matrix

Protein weight matrix=BLOSUM, PAM, GONNET, ID or filename

This controls the addition of the -matrix parameter and its associated value. Set this property to the argument value required.

property maxdiv

% ident. for delay

This controls the addition of the -maxdiv parameter and its associated value. Set this property to the argument value required.

property maxseqlen

Maximum allowed input sequence length

This controls the addition of the -maxseqlen parameter and its associated value. Set this property to the argument value required.

property negative

Protein alignment with negative values in matrix

This property controls the addition of the -negative switch, treat this property as a boolean.

property newtree

Output file name for newly created guide tree

This controls the addition of the -newtree parameter and its associated value. Set this property to the argument value required.

property newtree1

Output file name for new guide tree of profile1

This controls the addition of the -newtree1 parameter and its associated value. Set this property to the argument value required.

property newtree2

Output file for new guide tree of profile2

This controls the addition of the -newtree2 parameter and its associated value. Set this property to the argument value required.

property nohgap

Hydrophilic gaps off

This property controls the addition of the -nohgap switch, treat this property as a boolean.

property nopgap

Residue-specific gaps off

This property controls the addition of the -nopgap switch, treat this property as a boolean.

property nosecstr1

Do not use secondary structure-gap penalty mask for profile 1

This property controls the addition of the -nosecstr1 switch, treat this property as a boolean.

property nosecstr2

Do not use secondary structure-gap penalty mask for profile 2

This property controls the addition of the -nosecstr2 switch, treat this property as a boolean.

property noweights

Disable sequence weighting

This property controls the addition of the -noweights switch, treat this property as a boolean.

property numiter

maximum number of iterations to perform

This controls the addition of the -numiter parameter and its associated value. Set this property to the argument value required.

property options

List the command line parameters

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output sequence alignment file name

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property outorder

Output taxon order: INPUT or ALIGNED

This controls the addition of the -outorder parameter and its associated value. Set this property to the argument value required.

property output

Output format: CLUSTAL(default), GCG, GDE, PHYLIP, PIR, NEXUS and FASTA

This controls the addition of the -output parameter and its associated value. Set this property to the argument value required.

property outputtree

nj OR phylip OR dist OR nexus

This controls the addition of the -outputtree parameter and its associated value. Set this property to the argument value required.

property pairgap

Gap penalty

This controls the addition of the -pairgap parameter and its associated value. Set this property to the argument value required.

property pim

Output percent identity matrix (while calculating the tree).

This property controls the addition of the -pim switch, treat this property as a boolean.

property profile

Merge two alignments by profile alignment

This property controls the addition of the -profile switch, treat this property as a boolean.

property profile1

Profiles (old alignment).

This controls the addition of the -profile1 parameter and its associated value. Set this property to the argument value required.

property profile2

Profiles (old alignment).

This controls the addition of the -profile2 parameter and its associated value. Set this property to the argument value required.

property pwdnamatrix

DNA weight matrix=IUB, CLUSTALW or filename

This controls the addition of the -pwdnamatrix parameter and its associated value. Set this property to the argument value required.

property pwgapext

Gap extension penalty

This controls the addition of the -pwgapext parameter and its associated value. Set this property to the argument value required.

property pwgapopen

Gap opening penalty

This controls the addition of the -pwgapopen parameter and its associated value. Set this property to the argument value required.

property pwmatrix

Protein weight matrix=BLOSUM, PAM, GONNET, ID or filename

This controls the addition of the -pwmatrix parameter and its associated value. Set this property to the argument value required.

property quicktree

Use FAST algorithm for the alignment guide tree

This property controls the addition of the -quicktree switch, treat this property as a boolean.

property quiet

Reduce console output to minimum

This property controls the addition of the -quiet switch, treat this property as a boolean.

property range

Sequence range to write starting m to m+n. Input as string eg. '24,200'

This controls the addition of the -range parameter and its associated value. Set this property to the argument value required.

property score

Either: PERCENT or ABSOLUTE

This controls the addition of the -score parameter and its associated value. Set this property to the argument value required.

property secstrout

STRUCTURE or MASK or BOTH or NONE output in alignment file

This controls the addition of the -secstrout parameter and its associated value. Set this property to the argument value required.

property seed

Seed number for bootstraps.

This controls the addition of the -seed parameter and its associated value. Set this property to the argument value required.

property seqno_range

OFF or ON (NEW- for all output formats)

This controls the addition of the -seqno_range parameter and its associated value. Set this property to the argument value required.

property seqnos

OFF or ON (for Clustal output only)

This controls the addition of the -seqnos parameter and its associated value. Set this property to the argument value required.

property sequences

Sequentially add profile2 sequences to profile1 alignment

This property controls the addition of the -sequences switch, treat this property as a boolean.

property stats

Log some alignment statistics to file

This controls the addition of the -stats parameter and its associated value. Set this property to the argument value required.

property strandendin

Number of residues inside strand to be treated as terminal

This controls the addition of the -strandendin parameter and its associated value. Set this property to the argument value required.

property strandendout

Number of residues outside strand to be treated as terminal

This controls the addition of the -strandendout parameter and its associated value. Set this property to the argument value required.

property strandgap

gap penalty for strand core residues

This controls the addition of the -strandgap parameter and its associated value. Set this property to the argument value required.

property terminalgap

Gap penalty for structure termini

This controls the addition of the -terminalgap parameter and its associated value. Set this property to the argument value required.

property topdiags

Number of best diags.

This controls the addition of the -topdiags parameter and its associated value. Set this property to the argument value required.

property tossgaps

Ignore positions with gaps.

This property controls the addition of the -tossgaps switch, treat this property as a boolean.

property transweight

Transitions weighting

This controls the addition of the -transweight parameter and its associated value. Set this property to the argument value required.

property tree

Calculate NJ tree.

This property controls the addition of the -tree switch, treat this property as a boolean.

property type

PROTEIN or DNA sequences

This controls the addition of the -type parameter and its associated value. Set this property to the argument value required.

property usetree

File name of guide tree

This controls the addition of the -usetree parameter and its associated value. Set this property to the argument value required.

property usetree1

File name of guide tree for profile1

This controls the addition of the -usetree1 parameter and its associated value. Set this property to the argument value required.

property usetree2

File name of guide tree for profile2

This controls the addition of the -usetree2 parameter and its associated value. Set this property to the argument value required.

property window

Window around best diags.

This controls the addition of the -window parameter and its associated value. Set this property to the argument value required.

class Bio.Align.Applications.**ClustalOmegaCommandline**(*cmd='clustalo'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for clustal omega.

http://www.clustal.org/omega

Notes

Last checked against version: 1.2.0

References

Sievers F, Wilm A, Dineen DG, Gibson TJ, Karplus K, Li W, Lopez R, McWilliam H, Remmert M, Söding J, Thompson JD, Higgins DG (2011). Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega. Molecular Systems Biology 7:539 https://doi.org/10.1038/msb.2011.75

Examples

```
>>> from Bio.Align.Applications import ClustalOmegaCommandline
>>> in_file = "unaligned.fasta"
>>> out_file = "aligned.fasta"
>>> clustalomega_cline = ClustalOmegaCommandline(infile=in_file, outfile=out_file,␣
↪verbose=True, auto=True)
>>> print(clustalomega_cline)
clustalo -i unaligned.fasta -o aligned.fasta --auto -v
```

You would typically run the command line with clustalomega_cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='clustalo'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Set options automatically (might overwrite some of your options)

This property controls the addition of the –auto switch, treat this property as a boolean.

property clusteringout

Clustering output file

This controls the addition of the –clustering-out parameter and its associated value. Set this property to the argument value required.

property clustersize

soft maximum of sequences in sub-clusters

This controls the addition of the –cluster-size parameter and its associated value. Set this property to the argument value required.

property dealign

Dealign input sequences

This property controls the addition of the –dealign switch, treat this property as a boolean.

property distmat_full

Use full distance matrix for guide-tree calculation (slow; mBed is default)

This property controls the addition of the –full switch, treat this property as a boolean.

property distmat_full_iter

Use full distance matrix for guide-tree calculation during iteration (mBed is default)

This property controls the addition of the –full-iter switch, treat this property as a boolean.

property distmat_in

Pairwise distance matrix input file (skips distance computation).

This controls the addition of the –distmat-in parameter and its associated value. Set this property to the argument value required.

property distmat_out

Pairwise distance matrix output file.

This controls the addition of the –distmat-out parameter and its associated value. Set this property to the argument value required.

property force

Force file overwriting.

This property controls the addition of the –force switch, treat this property as a boolean.

property guidetree_in

Guide tree input file (skips distance computation and guide-tree clustering step).

This controls the addition of the –guidetree-in parameter and its associated value. Set this property to the argument value required.

property guidetree_out

Guide tree output file.

This controls the addition of the –guidetree-out parameter and its associated value. Set this property to the argument value required.

property help

Print help and exit.

This property controls the addition of the -h switch, treat this property as a boolean.

property hmm_input

HMM input files

This controls the addition of the –hmm-in parameter and its associated value. Set this property to the argument value required.

property infile

Multiple sequence input file

This controls the addition of the -i parameter and its associated value. Set this property to the argument value required.

property infmt

Forced sequence input file format (default: auto)

Allowed values: a2m, fa[sta], clu[stal], msf, phy[lip], selex, st[ockholm], vie[nna]

This controls the addition of the –infmt parameter and its associated value. Set this property to the argument value required.

property isprofile

disable check if profile, force profile (default no)

This property controls the addition of the –is-profile switch, treat this property as a boolean.

property iterations

Number of (combined guide-tree/HMM) iterations

This controls the addition of the –iterations parameter and its associated value. Set this property to the argument value required.

property log

Log all non-essential output to this file.

This controls the addition of the -l parameter and its associated value. Set this property to the argument value required.

property long_version

Print long version information and exit

This property controls the addition of the –long-version switch, treat this property as a boolean.

property max_guidetree_iterations

Maximum number of guidetree iterations

This controls the addition of the –max-guidetree-iterations parameter and its associated value. Set this property to the argument value required.

property max_hmm_iterations

Maximum number of HMM iterations

This controls the addition of the –max-hmm-iterations parameter and its associated value. Set this property to the argument value required.

property maxnumseq

Maximum allowed number of sequences

This controls the addition of the –maxnumseq parameter and its associated value. Set this property to the argument value required.

property maxseqlen

Maximum allowed sequence length

This controls the addition of the –maxseqlen parameter and its associated value. Set this property to the argument value required.

property outfile

Multiple sequence alignment output file (default: stdout).

This controls the addition of the -o parameter and its associated value. Set this property to the argument value required.

property outfmt

MSA output file format: a2m=fa[sta],clu[stal],msf,phy[lip],selex,st[ockholm],vie[nna] (default: fasta).

This controls the addition of the –outfmt parameter and its associated value. Set this property to the argument value required.

property outputorder

MSA output order like in input/guide-tree

This controls the addition of the –output-order parameter and its associated value. Set this property to the argument value required.

property percentid

convert distances into percent identities (default no)

This property controls the addition of the –percent-id switch, treat this property as a boolean.

property profile1

Pre-aligned multiple sequence file (aligned columns will be kept fix).

This controls the addition of the –profile1 parameter and its associated value. Set this property to the argument value required.

property profile2

Pre-aligned multiple sequence file (aligned columns will be kept fix).

This controls the addition of the –profile2 parameter and its associated value. Set this property to the argument value required.

property residuenumber

in Clustal format print residue numbers (default no)

This property controls the addition of the –residuenumber switch, treat this property as a boolean.

property seqtype

{Protein, RNA, DNA} Force a sequence type (default: auto).

This controls the addition of the -t parameter and its associated value. Set this property to the argument value required.

property threads

Number of processors to use

This controls the addition of the –threads parameter and its associated value. Set this property to the argument value required.

property usekimura

use Kimura distance correction for aligned sequences (default no)

This property controls the addition of the –use-kimura switch, treat this property as a boolean.

property verbose

Verbose output

This property controls the addition of the -v switch, treat this property as a boolean.

property version

Print version information and exit

This property controls the addition of the –version switch, treat this property as a boolean.

property wrap

number of residues before line-wrap in output

This controls the addition of the –wrap parameter and its associated value. Set this property to the argument value required.

class Bio.Align.Applications.**PrankCommandline**(*cmd='prank'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for the multiple alignment program PRANK.

http://www.ebi.ac.uk/goldman-srv/prank/prank/

Notes

Last checked against version: 081202

References

Loytynoja, A. and Goldman, N. 2005. An algorithm for progressive multiple alignment of sequences with insertions. Proceedings of the National Academy of Sciences, 102: 10557–10562.

Loytynoja, A. and Goldman, N. 2008. Phylogeny-aware gap placement prevents errors in sequence alignment and evolutionary analysis. Science, 320: 1632.

Examples

To align a FASTA file (unaligned.fasta) with the output in aligned FASTA format with the output filename starting with "aligned" (you can't pick the filename explicitly), no tree output and no XML output, use:

```
>>> from Bio.Align.Applications import PrankCommandline
>>> prank_cline = PrankCommandline(d="unaligned.fasta",
...                                o="aligned", # prefix only!
...                                f=8, # FASTA output
...                                notree=True, noxml=True)
>>> print(prank_cline)
prank -d=unaligned.fasta -o=aligned -f=8 -noxml -notree
```

You would typically run the command line with prank_cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='prank'*, ***kwargs*)

Initialize the class.

property F

Force insertions to be always skipped: same as +F

This property controls the addition of the -F switch, treat this property as a boolean.

__annotations__ = {}

property codon

Codon aware alignment or not

This property controls the addition of the -codon switch, treat this property as a boolean.

property convert

Convert input alignment to new format. Do not perform alignment

This property controls the addition of the -convert switch, treat this property as a boolean.

property d

Input filename

This controls the addition of the -d parameter and its associated value. Set this property to the argument value required.

property dnafreqs

DNA frequencies - 'A,C,G,T'. eg '25,25,25,25' as a quote surrounded string value. Default: empirical

This controls the addition of the -dnafreqs parameter and its associated value. Set this property to the argument value required.

property dots

Show insertion gaps as dots

This property controls the addition of the -dots switch, treat this property as a boolean.

property f

Output alignment format. Default: 8 FASTA Option are: 1. IG/Stanford 8. Pearson/Fasta 2. GenBank/GB 11. Phylip3.2 3. NBRF 12. Phylip 4. EMBL 14. PIR/CODATA 6. DNAStrider 15. MSF 7. Fitch 17. PAUP/NEXUS

This controls the addition of the -f parameter and its associated value. Set this property to the argument value required.

property fixedbranches

Use fixed branch lengths of input value

This controls the addition of the -fixedbranches parameter and its associated value. Set this property to the argument value required.

property gapext

Gap extension probability. Default: dna 0.5 / prot 0.5

This controls the addition of the -gapext parameter and its associated value. Set this property to the argument value required.

property gaprate

Gap opening rate. Default: dna 0.025 prot 0.0025

This controls the addition of the -gaprate parameter and its associated value. Set this property to the argument value required.

property kappa

Transition/transversion ratio. Default: 2

This controls the addition of the -kappa parameter and its associated value. Set this property to the argument value required.

property longseq

Save space in pairwise alignments

This property controls the addition of the -longseq switch, treat this property as a boolean.

property m

User-defined alignment model filename. Default: HKY2/WAG

This controls the addition of the -m parameter and its associated value. Set this property to the argument value required.

property matinitsize

Matrix initial size multiplier

This controls the addition of the -matinitsize parameter and its associated value. Set this property to the argument value required.

property matresize

Matrix resizing multiplier

This controls the addition of the -matresize parameter and its associated value. Set this property to the argument value required.

property maxbranches

Use maximum branch lengths of input value

This controls the addition of the -maxbranches parameter and its associated value. Set this property to the argument value required.

property mttranslate

Translate to protein using mt table

This property controls the addition of the -mttranslate switch, treat this property as a boolean.

property nopost

Do not compute posterior support. Default: compute

This property controls the addition of the -nopost switch, treat this property as a boolean.

property notree

Do not output dnd tree files (PRANK versions earlier than v.120626)

This property controls the addition of the -notree switch, treat this property as a boolean.

property noxml

Do not output XML files (PRANK versions earlier than v.120626)

This property controls the addition of the -noxml switch, treat this property as a boolean.

property o

Output filenames prefix. Default: 'output'
Will write: output.?.fas (depending on requested format), output.?.xml and output.?.dnd

This controls the addition of the -o parameter and its associated value. Set this property to the argument value required.

property once

Run only once. Default: twice if no guidetree given

This property controls the addition of the -once switch, treat this property as a boolean.

property printnodes

Output each node; mostly for debugging

This property controls the addition of the -printnodes switch, treat this property as a boolean.

property pwdist

Expected pairwise distance for computing guidetree. Default: dna 0.25 / prot 0.5

This controls the addition of the -pwdist parameter and its associated value. Set this property to the argument value required.

property pwgenomic

Do pairwise alignment, no guidetree

This property controls the addition of the -pwgenomic switch, treat this property as a boolean.

property pwgenomicdist

Distance for pairwise alignment. Default: 0.3

This controls the addition of the -pwgenomicdist parameter and its associated value. Set this property to the argument value required.

property quiet

Reduce verbosity

This property controls the addition of the -quiet switch, treat this property as a boolean.

property realbranches

Disable branch length truncation

This property controls the addition of the -realbranches switch, treat this property as a boolean.

property rho

Purine/pyrimidine ratio. Default: 1

This controls the addition of the -rho parameter and its associated value. Set this property to the argument value required.

property scalebranches

Scale branch lengths. Default: dna 1 / prot 2

This controls the addition of the -scalebranches parameter and its associated value. Set this property to the argument value required.

property shortnames

Truncate names at first space

This property controls the addition of the -shortnames switch, treat this property as a boolean.

property showtree

Output dnd tree files (PRANK v.120626 and later)

This property controls the addition of the -showtree switch, treat this property as a boolean.

property showxml

Output XML files (PRANK v.120626 and later)

This property controls the addition of the -showxml switch, treat this property as a boolean.

property skipins

Skip insertions in posterior support

This property controls the addition of the -skipins switch, treat this property as a boolean.

property t

Input guide tree filename

This controls the addition of the -t parameter and its associated value. Set this property to the argument value required.

property termgap

Penalise terminal gaps normally

This property controls the addition of the -termgap switch, treat this property as a boolean.

property translate

Translate to protein

This property controls the addition of the -translate switch, treat this property as a boolean.

property tree

Input guide tree as Newick string

This controls the addition of the -tree parameter and its associated value. Set this property to the argument value required.

property twice

Always run twice

This property controls the addition of the -twice switch, treat this property as a boolean.

property uselogs

Slower but should work for a greater number of sequences

This property controls the addition of the -uselogs switch, treat this property as a boolean.

property writeanc

Output ancestral sequences

This property controls the addition of the -writeanc switch, treat this property as a boolean.

class Bio.Align.Applications.**MafftCommandline**(*cmd='mafft'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for the multiple alignment program MAFFT.

http://align.bmr.kyushu-u.ac.jp/mafft/software/

Notes

Last checked against version: MAFFT v6.717b (2009/12/03)

References

Katoh, Toh (BMC Bioinformatics 9:212, 2008) Improved accuracy of multiple ncRNA alignment by incorporating structural information into a MAFFT-based framework (describes RNA structural alignment methods)

Katoh, Toh (Briefings in Bioinformatics 9:286-298, 2008) Recent developments in the MAFFT multiple sequence alignment program (outlines version 6)

Katoh, Toh (Bioinformatics 23:372-374, 2007) Errata PartTree: an algorithm to build an approximate tree from a large number of unaligned sequences (describes the PartTree algorithm)

Katoh, Kuma, Toh, Miyata (Nucleic Acids Res. 33:511-518, 2005) MAFFT version 5: improvement in accuracy of multiple sequence alignment (describes [ancestral versions of] the G-INS-i, L-INS-i and E-INS-i strategies)

Katoh, Misawa, Kuma, Miyata (Nucleic Acids Res. 30:3059-3066, 2002)

Examples

```
>>> from Bio.Align.Applications import MafftCommandline
>>> mafft_exe = "/opt/local/mafft"
>>> in_file = "../Doc/examples/opuntia.fasta"
>>> mafft_cline = MafftCommandline(mafft_exe, input=in_file)
>>> print(mafft_cline)
/opt/local/mafft ../Doc/examples/opuntia.fasta
```

If the mafft binary is on the path (typically the case on a Unix style operating system) then you don't need to supply the executable location:

```
>>> from Bio.Align.Applications import MafftCommandline
>>> in_file = "../Doc/examples/opuntia.fasta"
>>> mafft_cline = MafftCommandline(input=in_file)
>>> print(mafft_cline)
mafft ../Doc/examples/opuntia.fasta
```

You would typically run the command line with mafft_cline() or via the Python subprocess module, as described in the Biopython tutorial.

Note that MAFFT will write the alignment to stdout, which you may want to save to a file and then parse, e.g.:

```
stdout, stderr = mafft_cline()
with open("aligned.fasta", "w") as handle:
    handle.write(stdout)
from Bio import AlignIO
align = AlignIO.read("aligned.fasta", "fasta")
```

Alternatively, to parse the output with AlignIO directly you can use StringIO to turn the string into a handle:

```
stdout, stderr = mafft_cline()
from io import StringIO
from Bio import AlignIO
align = AlignIO.read(StringIO(stdout), "fasta")
```

__init__(*cmd='mafft'*, ***kwargs*)

Initialize the class.

property LEXP

Gap extension penalty to skip the alignment. Default: 0.00

This controls the addition of the –LEXP parameter and its associated value. Set this property to the argument value required.

property LOP

Gap opening penalty to skip the alignment. Default: -6.00

This controls the addition of the –LOP parameter and its associated value. Set this property to the argument value required.

__annotations__ = {}

property aamatrix

Use a user-defined AA scoring matrix. Default: BLOSUM62

This controls the addition of the –aamatrix parameter and its associated value. Set this property to the argument value required.

property adjustdirection

Adjust direction according to the first sequence. Default off.

This property controls the addition of the –adjustdirection switch, treat this property as a boolean.

property adjustdirectionaccurately

Adjust direction according to the first sequence,for highly diverged data; very slowDefault off.

This property controls the addition of the –adjustdirectionaccurately switch, treat this property as a boolean.

property amino

Assume the sequences are amino acid (True/False). Default: auto

This property controls the addition of the –amino switch, treat this property as a boolean.

property auto

Automatically select strategy. Default off.

This property controls the addition of the –auto switch, treat this property as a boolean.

property bl

BLOSUM number matrix is used. Default: 62

This controls the addition of the –bl parameter and its associated value. Set this property to the argument value required.

property clustalout

Output format: clustal (True) or fasta (False, default)

This property controls the addition of the –clustalout switch, treat this property as a boolean.

property dpparttree

The PartTree algorithm is used with distances based on DP. Default: off

This property controls the addition of the –dpparttree switch, treat this property as a boolean.

property ep

Offset value, which works like gap extension penalty, for group-to- group alignment. Default: 0.123

This controls the addition of the –ep parameter and its associated value. Set this property to the argument value required.

property fastapair

All pairwise alignments are computed with FASTA (Pearson and Lipman 1988). Default: off

This property controls the addition of the –fastapair switch, treat this property as a boolean.

property fastaparttree

The PartTree algorithm is used with distances based on FASTA. Default: off

This property controls the addition of the –fastaparttree switch, treat this property as a boolean.

property fft

Use FFT approximation in group-to-group alignment. Default: on

This property controls the addition of the –fft switch, treat this property as a boolean.

property fmodel

Incorporate the AA/nuc composition information into the scoring matrix (True) or not (False, default)

This property controls the addition of the –fmodel switch, treat this property as a boolean.

property genafpair

All pairwise alignments are computed with a local algorithm with the generalized affine gap cost (Altschul 1998). Default: off

This property controls the addition of the –genafpair switch, treat this property as a boolean.

property globalpair

All pairwise alignments are computed with the Needleman-Wunsch algorithm. Default: off

This property controls the addition of the –globalpair switch, treat this property as a boolean.

property groupsize

Do not make alignment larger than number sequences. Default: the number of input sequences

This property controls the addition of the –groupsize switch, treat this property as a boolean.

property input

Input file name

This controls the addition of the input parameter and its associated value. Set this property to the argument value required.

property input1

Second input file name for the mafft-profile command

This controls the addition of the input1 parameter and its associated value. Set this property to the argument value required.

property inputorder

Output order: same as input (True, default) or alignment based (False)

This property controls the addition of the –inputorder switch, treat this property as a boolean.

property jtt

JTT PAM number (Jones et al. 1992) matrix is used. number>0. Default: BLOSUM62

This controls the addition of the –jtt parameter and its associated value. Set this property to the argument value required.

property lep

Offset value at local pairwise alignment. Default: 0.1

This controls the addition of the –lep parameter and its associated value. Set this property to the argument value required.

property lexp

Gap extension penalty at local pairwise alignment. Default: -0.1

This controls the addition of the –lexp parameter and its associated value. Set this property to the argument value required.

property localpair

All pairwise alignments are computed with the Smith-Waterman algorithm. Default: off

This property controls the addition of the –localpair switch, treat this property as a boolean.

property lop

Gap opening penalty at local pairwise alignment. Default: 0.123

This controls the addition of the –lop parameter and its associated value. Set this property to the argument value required.

property maxiterate

Number cycles of iterative refinement are performed. Default: 0

This controls the addition of the –maxiterate parameter and its associated value. Set this property to the argument value required.

property memsave

Use the Myers-Miller (1988) algorithm. Default: automatically turned on when the alignment length exceeds 10,000 (aa/nt).

This property controls the addition of the –memsave switch, treat this property as a boolean.

property namelength

Name length in CLUSTAL and PHYLIP output.

MAFFT v6.847 (2011) added –namelength for use with the –clustalout option for CLUSTAL output.

MAFFT v7.024 (2013) added support for this with the –phylipout option for PHYLIP output (default 10).

This controls the addition of the –namelength parameter and its associated value. Set this property to the argument value required.

property nofft

Do not use FFT approximation in group-to-group alignment. Default: off

This property controls the addition of the –nofft switch, treat this property as a boolean.

property noscore

Alignment score is not checked in the iterative refinement stage. Default: off (score is checked)

This property controls the addition of the –noscore switch, treat this property as a boolean.

property nuc

Assume the sequences are nucleotide (True/False). Default: auto

This property controls the addition of the –nuc switch, treat this property as a boolean.

property op

Gap opening penalty at group-to-group alignment. Default: 1.53

This controls the addition of the –op parameter and its associated value. Set this property to the argument value required.

property partsize

The number of partitions in the PartTree algorithm. Default: 50

This controls the addition of the –partsize parameter and its associated value. Set this property to the argument value required.

property parttree

Use a fast tree-building method with the 6mer distance. Default: off

This property controls the addition of the –parttree switch, treat this property as a boolean.

property phylipout

Output format: phylip (True), or fasta (False, default)

This property controls the addition of the –phylipout switch, treat this property as a boolean.

property quiet

Do not report progress (True) or not (False, default).

This property controls the addition of the –quiet switch, treat this property as a boolean.

property reorder

Output order: aligned (True) or in input order (False, default)

This property controls the addition of the –reorder switch, treat this property as a boolean.

property retree

Guide tree is built number times in the progressive stage. Valid with 6mer distance. Default: 2

This controls the addition of the –retree parameter and its associated value. Set this property to the argument value required.

property seed

Seed alignments given in alignment_n (fasta format) are aligned with sequences in input.

This controls the addition of the –seed parameter and its associated value. Set this property to the argument value required.

property sixmerpair

Distance is calculated based on the number of shared 6mers. Default: on

This property controls the addition of the –6merpair switch, treat this property as a boolean.

property thread

Number of threads to use. Default: 1

This controls the addition of the –thread parameter and its associated value. Set this property to the argument value required.

property tm

Transmembrane PAM number (Jones et al. 1994) matrix is used. number>0. Default: BLOSUM62

This controls the addition of the –tm parameter and its associated value. Set this property to the argument value required.

property treeout

Guide tree is output to the input.tree file (True) or not (False, default)

This property controls the addition of the –treeout switch, treat this property as a boolean.

property weighti

Weighting factor for the consistency term calculated from pairwise alignments. Default: 2.7

This controls the addition of the –weighti parameter and its associated value. Set this property to the argument value required.

class Bio.Align.Applications.**DialignCommandline**(*cmd='dialign2-2'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for the multiple alignment program DIALIGN2-2.

http://bibiserv.techfak.uni-bielefeld.de/dialign/welcome.html

Notes

Last checked against version: 2.2

References

B. Morgenstern (2004). DIALIGN: Multiple DNA and Protein Sequence Alignment at BiBiServ. Nucleic Acids Research 32, W33-W36.

Examples

To align a FASTA file (unaligned.fasta) with the output files names aligned.* including a FASTA output file (aligned.fa), use:

```
>>> from Bio.Align.Applications import DialignCommandline
>>> dialign_cline = DialignCommandline(input="unaligned.fasta",
...                                    fn="aligned", fa=True)
>>> print(dialign_cline)
dialign2-2 -fa -fn aligned unaligned.fasta
```

You would typically run the command line with dialign_cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='dialign2-2'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property afc

Creates additional output file '*.afc' containing data of all fragments considered for alignment WARNING: this file can be HUGE !

This property controls the addition of the -afc switch, treat this property as a boolean.

property afc_v

Like '-afc' but verbose: fragments are explicitly printed. WARNING: this file can be EVEN BIGGER !

This property controls the addition of the -afc_v switch, treat this property as a boolean.

property anc

Anchored alignment. Requires a file <seq_file>.anc containing anchor points.

This property controls the addition of the -anc switch, treat this property as a boolean.

property cs

If segments are translated, not only the 'Watson strand' but also the 'Crick strand' is looked at.

This property controls the addition of the -cs switch, treat this property as a boolean.

property cw

Additional output file in CLUSTAL W format.

This property controls the addition of the -cw switch, treat this property as a boolean.

property ds

'dna alignment speed up' - non-translated nucleic acid fragments are taken into account only if they start with at least two matches. Speeds up DNA alignment at the expense of sensitivity.

This property controls the addition of the -ds switch, treat this property as a boolean.

property fa

Additional output file in FASTA format.

This property controls the addition of the -fa switch, treat this property as a boolean.

property ff

Creates file *.frg containing information about all fragments that are part of the respective optimal pairwise alignmnets plus information about consistency in the multiple alignment

This property controls the addition of the -ff switch, treat this property as a boolean.

property fn

Output files are named <out_file>.<extension>.

This controls the addition of the -fn parameter and its associated value. Set this property to the argument value required.

property fop

Creates file *.fop containing coordinates of all fragments that are part of the respective pairwise alignments.

This property controls the addition of the -fop switch, treat this property as a boolean.

property fsm

Creates file *.fsm containing coordinates of all fragments that are part of the final alignment

This property controls the addition of the -fsm switch, treat this property as a boolean.

property input

Input file name. Must be FASTA format

This controls the addition of the input parameter and its associated value. Set this property to the argument value required.

property iw

Overlap weights switched off (by default, overlap weights are used if up to 35 sequences are aligned). This option speeds up the alignment but may lead to reduced alignment quality.

This property controls the addition of the -iw switch, treat this property as a boolean.

property lgs

'long genomic sequences' - combines the following options: -ma, -thr 2, -lmax 30, -smin 8, -nta, -ff, -fop, -ff, -cs, -ds, -pst

This property controls the addition of the -lgs switch, treat this property as a boolean.

property lgs_t

Like '-lgs' but with all segment pairs assessed at the peptide level (rather than 'mixed alignments' as with the '-lgs' option). Therefore faster than -lgs but not very sensitive for non-coding regions.

This property controls the addition of the -lgs_t switch, treat this property as a boolean.

property lmax

Maximum fragment length = x (default: x = 40 or x = 120 for 'translated' fragments). Shorter x speeds up the program but may affect alignment quality.

This controls the addition of the -lmax parameter and its associated value. Set this property to the argument value required.

property lo

(Long Output) Additional file *.log with information about fragments selected for pairwise alignment and about consistency in multi-alignment procedure.

This property controls the addition of the -lo switch, treat this property as a boolean.

property ma

'mixed alignments' consisting of P-fragments and N-fragments if nucleic acid sequences are aligned.

This property controls the addition of the -ma switch, treat this property as a boolean.

property mask

Residues not belonging to selected fragments are replaced by '*' characters in output alignment (rather than being printed in lower-case characters)

This property controls the addition of the -mask switch, treat this property as a boolean.

property mat

Creates file *mat with substitution counts derived from the fragments that have been selected for alignment.

This property controls the addition of the -mat switch, treat this property as a boolean.

property mat_thr

Like '-mat' but only fragments with weight score > t are considered

This property controls the addition of the -mat_thr switch, treat this property as a boolean.

property max_link

'maximum linkage' clustering used to construct sequence tree (instead of UPGMA).

This property controls the addition of the -max_link switch, treat this property as a boolean.

property min_link

'minimum linkage' clustering used.

This property controls the addition of the -min_link switch, treat this property as a boolean.

property mot

'motif' option.

This controls the addition of the -mot parameter and its associated value. Set this property to the argument value required.

property msf

Separate output file in MSF format.

This property controls the addition of the -msf switch, treat this property as a boolean.

property n

Input sequences are nucleic acid sequences. No translation of fragments.

This property controls the addition of the -n switch, treat this property as a boolean.

property nt

Input sequences are nucleic acid sequences and 'nucleic acid segments' are translated to 'peptide segments'.

This property controls the addition of the -nt switch, treat this property as a boolean.

property nta

'no textual alignment' - textual alignment suppressed. This option makes sense if other output files are of interest – e.g. the fragment files created with -ff, -fop, -fsm or -lo.

This property controls the addition of the -nta switch, treat this property as a boolean.

property o

Fast version, resulting alignments may be slightly different.

This property controls the addition of the -o switch, treat this property as a boolean.

property ow

Overlap weights enforced (By default, overlap weights are used only if up to 35 sequences are aligned since calculating overlap weights is time consuming).

This property controls the addition of the -ow switch, treat this property as a boolean.

property pst

'print status'. Creates and updates a file *.sta with information about the current status of the program run. This option is recommended if large data sets are aligned since it allows the user to estimate the remaining running time.

This property controls the addition of the -pst switch, treat this property as a boolean.

property smin

Minimum similarity value for first residue pair (or codon pair) in fragments. Speeds up protein alignment or alignment of translated DNA fragments at the expense of sensitivity.

This property controls the addition of the -smin switch, treat this property as a boolean.

property stars

Maximum number of '*' characters indicating degree of local similarity among sequences. By default, no stars are used but numbers between 0 and 9, instead.

This controls the addition of the -stars parameter and its associated value. Set this property to the argument value required.

property stdo

Results written to standard output.

This property controls the addition of the -stdo switch, treat this property as a boolean.

property ta

Standard textual alignment printed (overrides suppression of textual alignments in special options, e.g. -lgs)

This property controls the addition of the -ta switch, treat this property as a boolean.

property thr

Threshold T = x.

This controls the addition of the -thr parameter and its associated value. Set this property to the argument value required.

property xfr

'exclude fragments' - list of fragments can be specified that are NOT considered for pairwise alignment

This property controls the addition of the -xfr switch, treat this property as a boolean.

class Bio.Align.Applications.**ProbconsCommandline**(*cmd='probcons'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for the multiple alignment program PROBCONS.

http://probcons.stanford.edu/

Notes

Last checked against version: 1.12

References

Do, C.B., Mahabhashyam, M.S.P., Brudno, M., and Batzoglou, S. 2005. PROBCONS: Probabilistic Consistency-based Multiple Sequence Alignment. Genome Research 15: 330-340.

Examples

To align a FASTA file (unaligned.fasta) with the output in ClustalW format, and otherwise default settings, use:

```
>>> from Bio.Align.Applications import ProbconsCommandline
>>> probcons_cline = ProbconsCommandline(input="unaligned.fasta",
...                                      clustalw=True)
>>> print(probcons_cline)
probcons -clustalw unaligned.fasta
```

You would typically run the command line with probcons_cline() or via the Python subprocess module, as described in the Biopython tutorial.

Note that PROBCONS will write the alignment to stdout, which you may want to save to a file and then parse, e.g.:

```
stdout, stderr = probcons_cline()
with open("aligned.aln", "w") as handle:
    handle.write(stdout)
from Bio import AlignIO
align = AlignIO.read("aligned.fasta", "clustalw")
```

Alternatively, to parse the output with AlignIO directly you can use StringIO to turn the string into a handle:

```
stdout, stderr = probcons_cline()
from io import StringIO
from Bio import AlignIO
align = AlignIO.read(StringIO(stdout), "clustalw")
```

__init__(*cmd='probcons'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property a

Print sequences in alignment order rather than input order (default: off)

This property controls the addition of the -a switch, treat this property as a boolean.

property annot

Write annotation for multiple alignment to FILENAME

This controls the addition of the -annot parameter and its associated value. Set this property to the argument value required.

property clustalw

Use CLUSTALW output format instead of MFA

This property controls the addition of the -clustalw switch, treat this property as a boolean.

property consistency

Use 0 <= REPS <= 5 (default: 2) passes of consistency transformation

This controls the addition of the -c parameter and its associated value. Set this property to the argument value required.

property emissions

Also reestimate emission probabilities (default: off)

This property controls the addition of the -e switch, treat this property as a boolean.

property input

Input file name. Must be multiple FASTA alignment (MFA) format

This controls the addition of the input parameter and its associated value. Set this property to the argument value required.

property ir

Use 0 <= REPS <= 1000 (default: 100) passes of iterative-refinement

This controls the addition of the -ir parameter and its associated value. Set this property to the argument value required.

property pairs

Generate all-pairs pairwise alignments

This property controls the addition of the -pairs switch, treat this property as a boolean.

property paramfile

Read parameters from FILENAME

This controls the addition of the -p parameter and its associated value. Set this property to the argument value required.

property pre

Use 0 <= REPS <= 20 (default: 0) rounds of pretraining

This controls the addition of the -pre parameter and its associated value. Set this property to the argument value required.

property train

Compute EM transition probabilities, store in FILENAME (default: no training)

This controls the addition of the -t parameter and its associated value. Set this property to the argument value required.

property verbose

Report progress while aligning (default: off)

This property controls the addition of the -verbose switch, treat this property as a boolean.

property viterbi

Use Viterbi algorithm to generate all pairs (automatically enables -pairs)

This property controls the addition of the -viterbi switch, treat this property as a boolean.

class Bio.Align.Applications.**TCoffeeCommandline**(*cmd='t_coffee'*, ***kwargs*)

Bases: *AbstractCommandline*

Commandline object for the TCoffee alignment program.

http://www.tcoffee.org/Projects_home_page/t_coffee_home_page.html

The T-Coffee command line tool has a lot of switches and options. This wrapper implements a VERY limited number of options - if you would like to help improve it please get in touch.

Notes

Last checked against: Version_6.92

References

T-Coffee: A novel method for multiple sequence alignments. Notredame, Higgins, Heringa, JMB,302(205-217) 2000

Examples

To align a FASTA file (unaligned.fasta) with the output in ClustalW format (file aligned.aln), and otherwise default settings, use:

```
>>> from Bio.Align.Applications import TCoffeeCommandline
>>> tcoffee_cline = TCoffeeCommandline(infile="unaligned.fasta",
...                                    output="clustalw",
...                                    outfile="aligned.aln")
>>> print(tcoffee_cline)
t_coffee -output clustalw -infile unaligned.fasta -outfile aligned.aln
```

You would typically run the command line with tcoffee_cline() or via the Python subprocess module, as described in the Biopython tutorial.

SEQ_TYPES = ['dna', 'protein', 'dna_protein']

__init__(*cmd='t_coffee'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property convert

Specify you want to perform a file conversion

This property controls the addition of the -convert switch, treat this property as a boolean.

property gapext

Indicates the penalty applied for extending a gap (negative integer)

This controls the addition of the -gapext parameter and its associated value. Set this property to the argument value required.

property gapopen

Indicates the penalty applied for opening a gap (negative integer)

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property infile

Specify the input file.

This controls the addition of the -infile parameter and its associated value. Set this property to the argument value required.

property matrix

Specify the filename of the substitution matrix to use. Default: blosum62mt

This controls the addition of the -matrix parameter and its associated value. Set this property to the argument value required.

property mode

Specifies a special mode: genome, quickaln, dali, 3dcoffee

This controls the addition of the -mode parameter and its associated value. Set this property to the argument value required.

property outfile

Specify the output file. Default: <your sequences>.aln

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property outorder

Specify the order of sequence to outputEither 'input', 'aligned' or <filename> of Fasta file with sequence order

This controls the addition of the -outorder parameter and its associated value. Set this property to the argument value required.

property output

Specify the output type.

One (or more separated by a comma) of: 'clustalw_aln', 'clustalw', 'gcg', 'msf_aln', 'pir_aln', 'fasta_aln', 'phylip', 'pir_seq', 'fasta_seq'

This controls the addition of the -output parameter and its associated value. Set this property to the argument value required.

property quiet

Turn off log output

This property controls the addition of the -quiet switch, treat this property as a boolean.

property type

Specify the type of sequence being aligned

This controls the addition of the -type parameter and its associated value. Set this property to the argument value required.

class Bio.Align.Applications.**MSAProbsCommandline**(*cmd='msaprobs'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for MSAProbs.

http://msaprobs.sourceforge.net

Notes

Last checked against version: 0.9.7

References

Yongchao Liu, Bertil Schmidt, Douglas L. Maskell: "MSAProbs: multiple sequence alignment based on pair hidden Markov models and partition function posterior probabilities". Bioinformatics, 2010, 26(16): 1958 -1964

Examples

```
>>> from Bio.Align.Applications import MSAProbsCommandline
>>> in_file = "unaligned.fasta"
>>> out_file = "aligned.cla"
>>> cline = MSAProbsCommandline(infile=in_file, outfile=out_file, clustalw=True)
>>> print(cline)
msaprobs -o aligned.cla -clustalw unaligned.fasta
```

You would typically run the command line with cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='msaprobs'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property alignment_order

print sequences in alignment order rather than input order (default: off)

This property controls the addition of the -a switch, treat this property as a boolean.

property annot

write annotation for multiple alignment to FILENAME

This controls the addition of the -annot parameter and its associated value. Set this property to the argument value required.

property clustalw

use CLUSTALW output format instead of FASTA format

This property controls the addition of the -clustalw switch, treat this property as a boolean.

property consistency

use 0 <= REPS <= 5 (default: 2) passes of consistency transformation

This controls the addition of the -c parameter and its associated value. Set this property to the argument value required.

property infile

Multiple sequence input file

This controls the addition of the infile parameter and its associated value. Set this property to the argument value required.

property iterative_refinement

use 0 <= REPS <= 1000 (default: 10) passes of iterative-refinement

This controls the addition of the -ir parameter and its associated value. Set this property to the argument value required.

property numthreads

specify the number of threads used, and otherwise detect automatically

This controls the addition of the -num_threads parameter and its associated value. Set this property to the argument value required.

property outfile

specify the output file name (STDOUT by default)

This controls the addition of the -o parameter and its associated value. Set this property to the argument value required.

property verbose

report progress while aligning (default: off)

This property controls the addition of the -v switch, treat this property as a boolean.

property version

print out version of MSAPROBS

This controls the addition of the -version parameter and its associated value. Set this property to the argument value required.

Bio.Align.substitution_matrices package

Module contents

Substitution matrices.

class Bio.Align.substitution_matrices.**Array**(*alphabet=None*, *dims=None*, *data=None*, *dtype=float*)

Bases: ndarray

numpy array subclass indexed by integers and by letters.

static __new__(*cls*, *alphabet=None*, *dims=None*, *data=None*, *dtype=float*)

Create a new Array instance.

__array_finalize__(*obj*, /)

Present so subclasses can call super. Does nothing.

__getitem__(*key*)

Return self[key].

__setitem__(*key*, *value*)

Set self[key] to value.

__contains__(*key*)

Return key in self.

__array_prepare__(*array*, [*context*,]/)

Returns a view of *array* with the same type as self.

__array_wrap__(*array*, [*context*,]/)

Returns a view of *array* with the same type as self.

__array_ufunc__(*ufunc*, *method*, **inputs*, ***kwargs*)

__reduce__()

For pickling.

__setstate__(*state*, /)

For unpickling.

The *state* argument must be a sequence that contains the following elements:

Parameters

version
[int] optional pickle version. If omitted defaults to 0.

shape
[tuple]

dtype
[data-type]

isFortran
[bool]

rawdata
[string or list] a binary string with the data (or a list if 'a' is an object array)

`transpose`(*axes=None*)

Transpose the array.

`property alphabet`

Return the alphabet property.

`copy`()

Create and return a copy of the array.

`get`(*key*, *value=None*)

Return the value of the key if found; return value otherwise.

`items`()

Return an iterator of (key, value) pairs in the array.

`keys`()

Return a tuple with the keys associated with the array.

`values`()

Return a tuple with the values stored in the array.

`update`(*E=None*, ***F*)

Update the array from dict/iterable E and F.

`select`(*alphabet*)

Subset the array by selecting the letters from the specified alphabet.

`__format__`(*fmt*)

Default object formatter.

`format`(*fmt=''*)

Return a string representation of the array.

The argument `fmt` specifies the number format to be used. By default, the number format is "%i" if the array contains integer numbers, and "%.1f" otherwise.

`__str__`()

Return str(self).

`__repr__`()

Return repr(self).

Bio.Align.substitution_matrices.**read**(*handle*, *dtype=float*)

Parse the file and return an Array object.

Bio.Align.substitution_matrices.**load**(*name=None*)

Load and return a precalculated substitution matrix.

```
>>> from Bio.Align import substitution_matrices
>>> names = substitution_matrices.load()
```

Submodules

Bio.Align.AlignInfo module

Extract information from alignment objects.

In order to try and avoid huge alignment objects with tons of functions, functions which return summary type information about alignments should be put into classes in this module.

class Bio.Align.AlignInfo.**SummaryInfo**(*alignment*)

Bases: object

Calculate summary info about the alignment.

This class should be used to calculate information summarizing the results of an alignment. This may either be straight consensus info or more complicated things.

__init__(*alignment*)

Initialize with the alignment to calculate information on.

ic_vector attribute. A list of ic content for each column number.

dumb_consensus(*threshold=0.7*, *ambiguous='X'*, *require_multiple=False*)

Output a fast consensus sequence of the alignment.

This doesn't do anything fancy at all. It will just go through the sequence residue by residue and count up the number of each type of residue (ie. A or G or T or C for DNA) in all sequences in the alignment. If the percentage of the most common residue type is greater then the passed threshold, then we will add that residue type, otherwise an ambiguous character will be added.

This could be made a lot fancier (ie. to take a substitution matrix into account), but it just meant for a quick and dirty consensus.

Arguments:

- threshold - The threshold value that is required to add a particular atom.
- ambiguous - The ambiguous character to be added when the threshold is not reached.
- require_multiple - If set as True, this will require that more than 1 sequence be part of an alignment to put it in the consensus (ie. not just 1 sequence and gaps).

gap_consensus(*threshold=0.7*, *ambiguous='X'*, *require_multiple=False*)

Output a fast consensus sequence of the alignment, allowing gaps.

Same as dumb_consensus(), but allows gap on the output.

Things to do:

- Let the user define that with only one gap, the result character in consensus is gap.
- Let the user select gap character, now it takes the same as input.

replacement_dictionary(*skip_chars=None*, *letters=None*)

Generate a replacement dictionary to plug into a substitution matrix.

This should look at an alignment, and be able to generate the number of substitutions of different residues for each other in the aligned object.

Will then return a dictionary with this information:

```
{('A', 'C') : 10, ('C', 'A') : 12, ('G', 'C') : 15 ....}
```

This also treats weighted sequences. The following example shows how we calculate the replacement dictionary. Given the following multiple sequence alignment:

```
GTATC  0.5
AT--C  0.8
CTGTC  1.0
```

For the first column we have:

```
('A', 'G') : 0.5 * 0.8 = 0.4
('C', 'G') : 0.5 * 1.0 = 0.5
('A', 'C') : 0.8 * 1.0 = 0.8
```

We then continue this for all of the columns in the alignment, summing the information for each substitution in each column, until we end up with the replacement dictionary.

Arguments:

- skip_chars - Not used; setting it to anything other than None will raise a ValueError
- letters - An iterable (e.g. a string or list of characters to include.

pos_specific_score_matrix(*axis_seq=None*, *chars_to_ignore=None*)

Create a position specific score matrix object for the alignment.

This creates a position specific score matrix (pssm) which is an alternative method to look at a consensus sequence.

Arguments:

- chars_to_ignore - A list of all characters not to include in the pssm.
- axis_seq - An optional argument specifying the sequence to put on the axis of the PSSM. This should be a Seq object. If nothing is specified, the consensus sequence, calculated with default parameters, will be used.

Returns:

- A PSSM (position specific score matrix) object.

information_content(*start=0*, *end=None*, *e_freq_table=None*, *log_base=2*, *chars_to_ignore=None*, *pseudo_count=0*)

Calculate the information content for each residue along an alignment.

Arguments:

- start, end - The starting an ending points to calculate the information content. These points should be relative to the first sequence in the alignment, starting at zero (ie. even if the 'real' first position in the seq is 203 in the initial sequence, for the info content, we need to use zero). This defaults to the entire length of the first sequence.

- e_freq_table - A dictionary specifying the expected frequencies for each letter (e.g. {'G' : 0.4, 'C' : 0.4, 'T' : 0.1, 'A' : 0.1}). Gap characters should not be included, since these should not have expected frequencies.
- log_base - The base of the logarithm to use in calculating the information content. This defaults to 2 so the info is in bits.
- chars_to_ignore - A listing of characters which should be ignored in calculating the info content. Defaults to none.

Returns:

- A number representing the info content for the specified region.

Please see the Biopython manual for more information on how information content is calculated.

get_column(*col*)

Return column of alignment.

class Bio.Align.AlignInfo.**PSSM**(*pssm*)

Bases: object

Represent a position specific score matrix.

This class is meant to make it easy to access the info within a PSSM and also make it easy to print out the information in a nice table.

Let's say you had an alignment like this:

```
GTATC
AT--C
CTGTC
```

The position specific score matrix (when printed) looks like:

```
  G A T C
G 1 1 0 1
T 0 0 3 0
A 1 1 0 0
T 0 0 2 0
C 0 0 0 3
```

You can access a single element of the PSSM using the following:

```
your_pssm[sequence_number][residue_count_name]
```

For instance, to get the 'T' residue for the second element in the above alignment you would need to do:

your_pssm[1]['T']

__init__(*pssm*)

Initialize with pssm data to represent.

The pssm passed should be a list with the following structure:

list[0] - The letter of the residue being represented (for instance, from the example above, the first few list[0]s would be GTAT... list[1] - A dictionary with the letter substitutions and counts.

__getitem__(*pos*)

__str__()

Return str(self).

get_residue(*pos*)

Return the residue letter at the specified position.

Bio.Align.AlignInfo.**print_info_content**(*summary_info*, *fout=None*, *rep_record=0*)

3 column output: position, aa in representative sequence, ic_vector value.

Bio.Align.a2m module

Bio.Align support for A2M files.

A2M files are alignment files created by align2model or hmmscore in the SAM Sequence Alignment and Modeling Software System.

class Bio.Align.a2m.**AlignmentWriter**(*target*)

Bases: *AlignmentWriter*

Alignment file writer for the A2M file format.

fmt: str | None = 'A2M'

format_alignment(*alignment*)

Return a string with the alignment in the A2M file format.

write_alignments(*stream*, *alignments*)

Write a single alignment to the output file, and return 1.

alignments - A list or iterator returning Alignment objects stream - Output file stream.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.a2m.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Alignment iterator for files in the A2M file format.

An A2M file contains one multiple alignment. Matches are represented by upper case letters and deletions by dashes in alignment columns containing matches or deletions only. Insertions are represented by lower case letters, with gaps aligned to the insertion shown as periods. Header lines start with '>' followed by the name of the sequence, and optionally a description.

fmt: str | None = 'A2M'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.analysis module

Code for performing calculations on codon alignments.

`Bio.Align.analysis.`**`calculate_dn_ds`**(*alignment*, *method='NG86'*, *codon_table=None*, *k=1*, *cfreq=None*)

Calculate dN and dS of the given two sequences.

Available methods:

- NG86 - Nei and Gojobori (1986) (PMID 3444411).
- LWL85 - Li et al. (1985) (PMID 3916709).
- ML - Goldman and Yang (1994) (PMID 7968486).
- YN00 - Yang and Nielsen (2000) (PMID 10666704).

Arguments:

- k - transition/transversion rate ratio
- cfreq - Current codon frequency vector can only be specified when you are using ML method. Possible ways of getting cfreq are: F1x4, F3x4 and F61.

`Bio.Align.analysis.`**`calculate_dn_ds_matrix`**(*alignment*, *method='NG86'*, *codon_table=None*)

Calculate dN and dS pairwise for the multiple alignment, and return as matrices.

Argument:

- method - Available methods include NG86, LWL85, YN00 and ML.
- codon_table - Codon table to use for forward translation.

`Bio.Align.analysis.`**`mktest`**(*alignment*, *species=None*, *codon_table=None*)

McDonald-Kreitman test for neutrality.

Implement the McDonald-Kreitman test for neutrality (PMID: 1904993) This method counts changes rather than sites (http://mkt.uab.es/mkt/help_mkt.asp).

Arguments:

- alignment - Alignment of gene nucleotide sequences to compare.
- species - List of the species ID for each sequence in the alignment. Typically, the species ID is the species name as a string, or an integer.
- codon_table - Codon table to use for forward translation.

Return the p-value of test result.

Bio.Align.bed module

Bio.Align support for BED (Browser Extensible Data) files.

The Browser Extensible Data (BED) format, stores a series of pairwise alignments in a single file. Typically they are used for transcript to genome alignments. BED files store the alignment positions and alignment scores, but not the aligned sequences.

See http://genome.ucsc.edu/FAQ/FAQformat.html#format1

You are expected to use this module via the Bio.Align functions.

Coordinates in the BED format are defined in terms of zero-based start positions (like Python) and aligning region sizes.

A minimal aligned region of length one and starting at first position in the source sequence would have `start == 0` and `size == 1`.

As we can see in this example, `start + size` will give one more than the zero-based end position. We can therefore manipulate `start` and `start + size` as python list slice boundaries.

class Bio.Align.bed.**AlignmentWriter**(*target*, *bedN=12*)

Bases: *AlignmentWriter*

Alignment file writer for the Browser Extensible Data (BED) file format.

__init__(*target*, *bedN=12*)

Create an AlignmentWriter object.

Arguments:

- target - output stream or file name
- **bedN - number of columns in the BED file.**
  This must be between 3 and 12; default value is 12.

format_alignment(*alignment*)

Return a string with one alignment formatted as a BED line.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.bed.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Alignment iterator for Browser Extensible Data (BED) files.

Each line in the file contains one pairwise alignment, which are loaded and returned incrementally. Additional alignment information is stored as attributes of each alignment.

fmt: str | None = 'BED'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.bigbed module

Bio.Align support for alignment files in the bigBed format.

The bigBed format stores a series of pairwise alignments in a single indexed binary file. Typically they are used for transcript to genome alignments. As in the BED format, the alignment positions and alignment scores are stored, but the aligned sequences are not.

See http://genome.ucsc.edu/goldenPath/help/bigBed.html for more information.

You are expected to use this module via the Bio.Align functions.

class Bio.Align.bigbed.**Field**(*as_type*, *name*, *comment*)

Bases: `tuple`

__getnewargs__()

Return self as a plain tuple. Used by copy and pickle.

__match_args__ = ('as_type', 'name', 'comment')

static __new__(*_cls*, *as_type*, *name*, *comment*)

Create new instance of Field(as_type, name, comment)

__repr__()

Return a nicely formatted representation string

__slots__ = ()

as_type

Alias for field number 0

comment

Alias for field number 2

name

Alias for field number 1

class Bio.Align.bigbed.AutoSQLTable(*name*, *comment*, *fields*)

Bases: list

AutoSQL table describing the columns of an (possibly extended) BED format.

default: *AutoSQLTable* = [('string', 'chrom', 'Reference sequence chromosome or scaffold'), ('uint', 'chromStart', 'Start position in chromosome'), ('uint', 'chromEnd', 'End position in chromosome'), ('string', 'name', 'Name of item.'), ('uint', 'score', 'Score (0-1000)'), ('char[1]', 'strand', '+ or - for strand'), ('uint', 'thickStart', 'Start of where display should be thick (start codon)'), ('uint', 'thickEnd', 'End of where display should be thick (stop codon)'), ('uint', 'reserved', 'Used as itemRgb as of 2004-11-22'), ('int', 'blockCount', 'Number of blocks'), ('int[blockCount]', 'blockSizes', 'Comma separated list of block sizes'), ('int[blockCount]', 'chromStarts', 'Start positions relative to chromStart')]

__init__(*name*, *comment*, *fields*)

Create an AutoSQL table describing the columns of an (extended) BED format.

classmethod from_bytes(*data*)

Return an AutoSQLTable initialized using the bytes object data.

classmethod from_string(*data*)

Return an AutoSQLTable initialized using the string object data.

__str__()

Return str(self).

__bytes__()

__getitem__(*i*)

x.__getitem__(y) <==> x[y]

__annotations__ = {'default': 'AutoSQLTable'}

class Bio.Align.bigbed.**AlignmentWriter**(*target*, *bedN=12*, *declaration=None*, *targets=None*, *compress=True*, *itemsPerSlot=512*, *blockSize=256*, *extraIndex=()*)

Bases: *AlignmentWriter*

Alignment file writer for the bigBed file format.

fmt: str | None = 'bigBed'

mode = 'wb'

__init__(*target*, *bedN=12*, *declaration=None*, *targets=None*, *compress=True*, *itemsPerSlot=512*, *blockSize=256*, *extraIndex=()*)

Create an AlignmentWriter object.

Arguments:

- target - output stream or file name.
- **bedN - number of columns in the BED file.**
  This must be between 3 and 12; default value is 12.
- **declaration - an AutoSQLTable object declaring the fields in the**
  BED file. Required only if the BED file contains extra (custom) fields. Default value is None.
- **targets - A list of SeqRecord objects with the chromosomes in**
  the order as they appear in the alignments. The sequence contents in each SeqRecord may be undefined, but the sequence length must be defined, as in this example:

  SeqRecord(Seq(None, length=248956422), id="chr1")

  If targets is None (the default value), the alignments must have an attribute .targets providing the list of SeqRecord objects.
- **compress - If True (default), compress data using zlib.**
  If False, do not compress data. Use compress=False for faster searching.
- **blockSize - Number of items to bundle in r-tree.**
  See UCSC's bedToBigBed program for more information. Default value is 256.
- **itemsPerSlot - Number of data points bundled at lowest level.**
  See UCSC's bedToBigBed program for more information. Use itemsPerSlot=1 for faster searching. Default value is 512.
- **extraIndex - List of strings with the names of extra columns to be**
  indexed. Default value is an empty list.

write_file(*stream*, *alignments*)

Write the alignments to the file stream, and return the number of alignments.

alignments - A list or iterator returning Alignment objects stream - Output file stream.

write_alignments(*alignments*, *output*, *reductions*, *extra_indices*)

Write alignments to the output file, and return the number of alignments.

alignments - A list or iterator returning Alignment objects stream - Output file stream.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.bigbed.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Alignment iterator for bigBed files.

The pairwise alignments stored in the bigBed file are loaded and returned incrementally. Additional alignment information is stored as attributes of each alignment.

fmt: str | None = 'bigBed'

mode = 'b'

__len__()

Return the number of alignments.

The number of alignments is cached. If not yet calculated, the iterator is rewound to the beginning, and the number of alignments is calculated by iterating over the alignments. The iterator is then returned to its original position in the file.

search(*chromosome=None*, *start=None*, *end=None*)

Iterate over alignments overlapping the specified chromosome region..

This method searches the index to find alignments to the specified chromosome that fully or partially overlap the chromosome region between start and end.

Arguments:

- chromosome - chromosome name. If None (default value), include all alignments.
- start - starting position on the chromosome. If None (default value), use 0 as the starting position.
- end - end position on the chromosome. If None (default value), use the length of the chromosome as the end position.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.bigmaf module

Bio.Align support for the "bigmaf" multiple alignment format.

The bigMaf format stores multiple alignments in a format compatible with the MAF (Multiple Alignment Format) format. BigMaf files are binary and are indexed as a bigBed file.

See https://genome.ucsc.edu/goldenPath/help/bigMaf.html

class Bio.Align.bigmaf.**AlignmentWriter**(*target*, *targets=None*, *compress=True*, *blockSize=256*, *itemsPerSlot=512*)

Bases: *AlignmentWriter*

Alignment file writer for the bigMaf file format.

fmt: str | None = 'bigMaf'

__init__(*target*, *targets=None*, *compress=True*, *blockSize=256*, *itemsPerSlot=512*)

Create an AlignmentWriter object.

Arguments:

- target - output stream or file name.

- **targets - A list of SeqRecord objects with the chromosomes in**
  the order as they appear in the alignments. The sequence contents in each SeqRecord may be undefined, but the sequence length must be defined, as in this example:

  SeqRecord(Seq(None, length=248956422), id="chr1")

  If targets is None (the default value), the alignments must have an attribute .targets providing the list of SeqRecord objects.
- **compress - If True (default), compress data using zlib.**
  If False, do not compress data. Use compress=False for faster searching.
- **blockSize - Number of items to bundle in r-tree.**
  See UCSC's bedToBigBed program for more information. Default value is 256.
- **itemsPerSlot - Number of data points bundled at lowest level.**
  See UCSC's bedToBigBed program for more information. Use itemsPerSlot=1 for faster searching. Default value is 512.

`write_file`(*stream*, *alignments*)

Write the file.

`__abstractmethods__ = frozenset({})`

`__annotations__ = {'fmt': 'Optional[str]'}`

class Bio.Align.bigmaf.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*, *AlignmentIterator*

Alignment iterator for bigMaf files.

The file may contain multiple alignments, which are loaded and returned incrementally.

Alignment annotations are stored in the `.annotations` attribute of the `Alignment` object, except for the alignment score, which is stored as an attribute. Sequence information of empty parts in the alignment block (sequences that connect the previous alignment block to the next alignment block, but do not align to the current alignment block) is stored in the alignment annotations under the `"empty"` key. Annotations specific to each line in the alignment are stored in the `.annotations` attribute of the corresponding sequence record.

`fmt: str | None = 'bigMaf'`

`mode = 'b'`

`__init__`(*source*)

Create an AlignmentIterator object.

Arguments: - source - input file stream, or path to input file

`__abstractmethods__ = frozenset({})`

`__annotations__ = {'fmt': 'Optional[str]'}`

Bio.Align.bigpsl module

Bio.Align support for alignment files in the bigPsl format.

A bigPsl file is a bigBed file with a BED12+13 format consisting of the 12 predefined BED fields and 13 custom fields defined in the autoSql file bigPsl.as. This module uses the Bio.Align.bigbed module to parse the file, but stores the data in a PSL-consistent manner as defined in bigPsl.as. As the bigPsl format is a special case of the bigBed format, bigPsl files are binary and are indexed as bigBed files.

See http://genome.ucsc.edu/goldenPath/help/bigPsl.html for more information.

You are expected to use this module via the Bio.Align functions.

class Bio.Align.bigpsl.**AlignmentWriter**(*target*, *targets=None*, *compress=True*, *extraIndex=()*, *cds=False*, *fa=False*, *mask=None*, *wildcard='N'*)

Bases: *AlignmentWriter*

Alignment file writer for the bigPsl file format.

fmt: str | None = 'bigPsl'

__init__(*target*, *targets=None*, *compress=True*, *extraIndex=()*, *cds=False*, *fa=False*, *mask=None*, *wildcard='N'*)

Create an AlignmentWriter object.

Arguments:

- target - output stream or file name.
- **targets - A list of SeqRecord objects with the chromosomes in the** order as they appear in the alignments. The sequence contents in each SeqRecord may be undefined, but the sequence length must be defined, as in this example:

  SeqRecord(Seq(None, length=248956422), id="chr1")

  If targets is None (the default value), the alignments must have an attribute .targets providing the list of SeqRecord objects.
- **compress - If True (default), compress data using zlib.** If False, do not compress data.
- **extraIndex - List of strings with the names of extra columns to be** indexed. Default value is an empty list.
- **cds - If True, look for a query feature of type CDS and write** it in NCBI style in the PSL file (default: False).
- **fa - If True, include the query sequence in the PSL file** (default: False).
- **mask - Specify if repeat regions in the target sequence are** masked and should be reported in the *repMatches* field instead of in the *matches* field. Acceptable values are None : no masking (default); "lower": masking by lower-case characters; "upper": masking by upper-case characters.
- **wildcard - Report alignments to the wildcard character in the** target or query sequence in the *nCount* field instead of in the *matches*, *misMatches*, or *repMatches* fields. Default value is 'N'.

write_file(*stream*, *alignments*)

Write the file.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.bigpsl.AlignmentIterator(*source*)

Bases: *AlignmentIterator*

Alignment iterator for bigPsl files.

The pairwise alignments stored in the bigPsl file are loaded and returned incrementally. Additional alignment information is stored as attributes of each alignment.

fmt: str | None = 'bigPsl'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.chain module

Bio.Align support for the "chain" pairwise alignment format.

As described by UCSC, a chain file stores a series of pairwise alignments in a single file. Typically they are used for genome to genome alignments. Chain files store the lengths of the aligned segments, alignment gaps, and alignment scores, but do not store the aligned sequences.

See https://genome.ucsc.edu/goldenPath/help/chain.html.

You are expected to use this module via the Bio.Align functions.

Coordinates in the chain file format are defined in terms of zero-based start positions (like Python) and aligning region sizes.

class Bio.Align.chain.AlignmentWriter(*target*)

Bases: *AlignmentWriter*

Alignment file writer for the UCSC chain file format.

fmt: str | None = 'chain'

format_alignment(*alignment*)

Return a string with one alignment formatted as a chain block.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.chain.AlignmentIterator(*source*)

Bases: *AlignmentIterator*

Alignment iterator for UCSC chain files.

Each chain block in the file contains one pairwise alignment, which are loaded and returned incrementally. The alignment score is scored as an attribute of the alignment; the ID is stored under the key "id" in the dictionary referred to by the annotations attribute of the alignment.

fmt: str | None = 'chain'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.clustal module

Bio.Align support for "clustal" output from CLUSTAL W and other tools.

You are expected to use this module via the Bio.Align functions (or the Bio.SeqIO functions if you are interested in the sequences only).

class Bio.Align.clustal.**AlignmentWriter**(*target*)

Bases: *AlignmentWriter*

Clustalw alignment writer.

fmt: str | None = 'Clustal'

write_header(*stream*, *alignments*)

Use this to write the file header.

format_alignment(*alignment*)

Return a string with a single alignment in the Clustal format.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.clustal.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Clustalw alignment iterator.

fmt: str | None = 'Clustal'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.emboss module

Bio.Align support for "emboss" alignment output from EMBOSS tools.

This module contains a parser for the EMBOSS srspair/pair/simple file format, for example from the needle, water, and stretcher tools.

class Bio.Align.emboss.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Emboss alignment iterator.

For reading the (pairwise) alignments from EMBOSS tools in what they call the "pairs" and "simple" formats.

fmt: str | None = 'EMBOSS'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.exonerate module

Bio.Align support for Exonerate output format.

This module provides support for Exonerate outputs. Exonerate is a generic tool for pairwise sequence comparison that allows you to align sequences using several different models.

Bio.Align.exonerate was tested on the following Exonerate versions and models:

- version: 2.2
- models: - affine:local - cdna2genome - coding2coding - est2genome - genome2genome - ner - protein2dna - protein2genome - ungapped - ungapped:translated

Although model testing were not exhaustive, the parser should be able to cope with all Exonerate models. Please file a bug report if you stumble upon an unparsable file.

You are expected to use this module via the Bio.Align functions.

class Bio.Align.exonerate.**AlignmentWriter**(*target*, *fmt='vulgar'*)

Bases: *AlignmentWriter*

Alignment file writer for the Exonerate cigar and vulgar file format.

fmt: str | None = 'Exonerate'

__init__(*target*, *fmt='vulgar'*)

Create an AlignmentWriter object.

Arguments:

- target - output stream or file name
- **fmt - write alignments in the vulgar (Verbose Useful Labelled** Gapped Alignment Report) format (fmt="vulgar") or in the cigar (Compact Idiosyncratic Gapped Alignment Report) format (fmt="cigar"). Default value is 'vulgar'.

write_header(*stream*, *alignments*)

Write the header.

write_footer(*stream*)

Write the footer.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.exonerate.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Alignment iterator for the Exonerate text, cigar, and vulgar formats.

Each line in the file contains one pairwise alignment, which are loaded and returned incrementally. Alignment score information such as the number of matches and mismatches are stored as attributes of each alignment.

fmt: str | None = 'Exonerate'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.fasta module

Bio.Align support for aligned FASTA files.

Aligned FASTA files are FASTA files in which alignment gaps in a sequence are represented by dashes. Each sequence line in an aligned FASTA should have the same length.

class Bio.Align.fasta.**AlignmentWriter**(*target*)

Bases: *AlignmentWriter*

Alignment file writer for the aligned FASTA file format.

fmt: str | None = 'FASTA'

format_alignment(*alignment*)

Return a string with the alignment in aligned FASTA format.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.fasta.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Alignment iterator for aligned FASTA files.

An aligned FASTA file contains one multiple alignment. Alignment gaps are represented by dashes in the sequence lines. Header lines start with '>' followed by the name of the sequence, and optionally a description.

fmt: str | None = 'FASTA'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.hhr module

Bio.Align support for hhr files generated by HHsearch or HHblits in HH-suite.

You are expected to use this module via the Bio.Align functions.

class Bio.Align.hhr.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Alignment iterator for hhr output files generated by HHsearch or HHblits.

HHsearch and HHblits are part of the HH-suite of programs for Hidden Markov Models. An output files in the hhr format contains multiple pairwise alignments for a single query sequence.

fmt: str | None = 'hhr'

__len__()

Return the number of alignments.

The number of alignments is cached. If not yet calculated, the iterator is rewound to the beginning, and the number of alignments is calculated by iterating over the alignments. The iterator is then returned to its original position in the file.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.interfaces module

Bio.Align support module (not for general use).

Unless you are writing a new parser or writer for Bio.Align, you should not use this module. It provides base classes to try and simplify things.

class Bio.Align.interfaces.**AlignmentIterator**(*source*)

Bases: *AlignmentsAbstractBaseClass*

Base class for building Alignment iterators.

You should write a parse method that returns an Alignment generator. You may wish to redefine the __init__ method as well.

Subclasses may define the following class attributes: - mode - 't' or 'b' for text or binary files, respectively - fmt - a human-readable name for the file format.

mode = 't'

fmt: str | None = None

__init__(*source*)

Create an AlignmentIterator object.

Arguments: - source - input file stream, or path to input file

This method MAY be overridden by any subclass.

Note when subclassing: - there should be a single non-optional argument, the source. - you can add additional optional arguments.

__next__()

Return the next alignment.

__len__()

Return the number of alignments.

The number of alignments is cached. If not yet calculated, the iterator is rewound to the beginning, and the number of alignments is calculated by iterating over the alignments. The iterator is then returned to its original position in the file.

__enter__()

__exit__(*exc_type*, *exc_value*, *exc_traceback*)

__getitem__(*index*)

Return the alignments as an Alignments object (which inherits from list).

Only an index of the form [:] (i.e., a full slice) is supported. The file stream is returned to its zero position, and the file header is read and stored in an Alignments object. Next, we iterate over the alignments and store them in the Alignments object. The iterator is then returned to its original position in the file, and the Alignments object is returned. The Alignments object contains the exact same information as the Alignment iterator self, but stores the alignments in a list instead of as an iterator, allowing indexing.

Typical usage is

```
>>> from Bio import Align
>>> alignments = Align.parse("Blat/dna_rna.psl", "psl")
>>> alignments.metadata
{'psLayout version': '3'}
```

As *alignments* is an iterator and not a list, we cannot retrieve an alignment by its index:

```
>>> alignment = alignments[2]
Traceback (most recent call last):
  ...
KeyError: 'only [:] (a full slice) can be used as the index'
```

So we use the iterator to create a list-like Alignments object:

```
>>> alignments = alignments[:]
```

While *alignments* is a list-like object, it has the same *metadata* attribute representing the information stored in the file header:

```
>>> alignments.metadata
{'psLayout version': '3'}
```

Now we can index individual alignments:

```
>>> len(alignments)
4
>>> alignment = alignments[2]
>>> alignment.target.id, alignment.query.id
('chr3', 'NR_111921.1')
```

rewind()

Rewind the iterator to let it loop over the alignments from the beginning.

__abstractmethods__ = frozenset({'_read_next_alignment'})

__annotations__ = {'fmt': typing.Optional[str]}

class Bio.Align.interfaces.**AlignmentWriter**(*target*)

Bases: ABC

Base class for alignment writers. This class should be subclassed.

It is intended for alignment file formats with an (optional) header, one or more alignments, and an (optional) footer.

The user may call the write_file() method to write a complete file containing the alignments.

Alternatively, users may call the write_header(), followed by multiple calls to format_alignment() and/or write_alignments(), followed finally by write_footer().

Subclasses may define the following class attributes: - mode - 'w' or 'wb' for text or binary files, respectively - fmt - a human-readable name for the file format.

mode = 'w'

fmt: str | None = None

__init__(*target*)

Create the writer object.

Arguments: - target - output file stream, or path to output file

This method MAY be overridden by any subclass.

Note when subclassing: - there should be a single non-optional argument, the target. - you can add additional optional arguments.

write_header(*stream*, *alignments*)

Write the file header to the output file.

write_footer(*stream*)

Write the file footer to the output file.

format_alignment(*alignment*)

Format a single alignment as a string.

alignment - an Alignment object

write_single_alignment(*stream*, *alignments*)

Write a single alignment to the output file, and return 1.

alignments - A list or iterator returning Alignment objects stream - Output file stream.

write_multiple_alignments(*stream*, *alignments*)

Write alignments to the output file, and return the number of alignments.

alignments - A list or iterator returning Alignment objects stream - Output file stream.

write_alignments(*stream*, *alignments*)

Write alignments to the output file, and return the number of alignments.

alignments - A list or iterator returning Alignment objects stream - Output file stream.

write_file(*stream*, *alignments*)

Write the alignments to the file strenm, and return the number of alignments.

alignments - A list or iterator returning Alignment objects stream - Output file stream.

write(*alignments*)

Write a file with the alignments, and return the number of alignments.

alignments - A list or iterator returning Alignment objects

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': typing.Optional[str]}

Bio.Align.maf module

Bio.Align support for the "maf" multiple alignment format.

The Multiple Alignment Format, described by UCSC, stores a series of multiple alignments in a single file. It is suitable for whole-genome to whole-genome alignments, metadata such as source chromosome, start position, size, and strand can be stored.

See http://genome.ucsc.edu/FAQ/FAQformat.html#format5

You are expected to use this module via the Bio.Align functions.

Coordinates in the MAF format are defined in terms of zero-based start positions (like Python) and aligning region sizes.

A minimal aligned region of length one and starting at first position in the source sequence would have `start == 0` and `size == 1`.

As we can see on this example, `start + size` will give one more than the zero-based end position. We can therefore manipulate `start` and `start + size` as python list slice boundaries.

class Bio.Align.maf.**AlignmentWriter**(*target*)

Bases: *AlignmentWriter*

Accepts Alignment objects, writes a MAF file.

fmt: str | None = 'MAF'

write_header(*stream*, *alignments*)

Write the MAF header.

format_alignment(*alignment*)

Return a string with a single alignment formatted as a MAF block.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.maf.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Alignment iterator for Multiple Alignment Format files.

The file may contain multiple concatenated alignments, which are loaded and returned incrementally.

File meta-data are stored in the `.metadata` attribute of the returned iterator. Alignment annotations are stored in the `.annotations` attribute of the `Alignment` object, except for the alignment score, which is stored as an attribute. Sequence information of empty parts in the alignment block (sequences that connect the previous alignment block to the next alignment block, but do not align to the current alignment block) is stored in the alignment annotations under the `"empty"` key. Annotations specific to each line in the alignment are stored in the `.annotations` attribute of the corresponding sequence record.

fmt: str | None = 'MAF'

status_characters = ('C', 'I', 'N', 'n', 'M', 'T')

empty_status_characters = ('C', 'I', 'M', 'n')

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.mauve module

Bio.Align support for "xmfa" output from Mauve/ProgressiveMauve.

You are expected to use this module via the Bio.Align functions.

class Bio.Align.mauve.**AlignmentWriter**(*target*, *metadata=None*, *identifiers=None*)

Bases: *AlignmentWriter*

Mauve xmfa alignment writer.

fmt: str | None = 'Mauve'

__init__(*target*, *metadata=None*, *identifiers=None*)

Create an AlignmentWriter object.

Arguments:

- target - output stream or file name
- **metadata - metadata to be included in the output. If metadata** is None, then the alignments object to be written must have an attribute *metadata*.
- **identifiers - list of the IDs of the sequences included in the** alignment. Sequences will be numbered according to their index in this list. If identifiers is None, then the alignments object to be written must have an attribute *identifiers*.

write_header(*stream*, *alignments*)

Write the file header to the output file.

write_file(*stream*, *alignments*)

Write a file with the alignments, and return the number of alignments.

alignments - A Bio.Align.mauve.AlignmentIterator object.

format_alignment(*alignment*)

Return a string with a single alignment in the Mauve format.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.mauve.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Mauve xmfa alignment iterator.

fmt: str | None = 'Mauve'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.msf module

Bio.Align support for GCG MSF format.

The file format was produced by the GCG PileUp and LocalPileUp tools, and later tools such as T-COFFEE and MUSCLE support it as an optional output format.

You are expected to use this module via the Bio.Align functions.

class Bio.Align.msf.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

GCG MSF alignment iterator.

fmt: str | None = 'MSF'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.nexus module

Bio.Align support for the "nexus" file format.

You are expected to use this module via the Bio.Align functions.

See also the Bio.Nexus module (which this code calls internally), as this offers more than just accessing the alignment or its sequences as SeqRecord objects.

class Bio.Align.nexus.**AlignmentWriter**(*target*, *interleave=None*)

Bases: *AlignmentWriter*

Nexus alignment writer.

Note that Nexus files are only expected to hold ONE alignment matrix.

You are expected to call this class via Bio.Align.write().

fmt: str | None = 'Nexus'

__init__(*target*, *interleave=None*)

Create an AlignmentWriter object.

Arguments:

- target - output stream or file name
- **interleave - if None (default): interleave if columns > 1000**
  if True: use interleaved format if False: do not use interleaved format

write_file(*stream*, *alignments*)

Write a file with the alignments, and return the number of alignments.

alignments - A list or iterator returning Alignment objects

format_alignment(*alignment*, *interleave=None*)

Return a string with a single alignment in the Nexus format.

Creates an empty Nexus object, adds the sequences and then gets Nexus to prepare the output.

- alignment - An Alignment object
- **interleave - if None (default): interleave if columns > 1000**
  if True: use interleaved format if False: do not use interleaved format

write_alignment(*alignment*, *stream*, *interleave=None*)

Write a single alignment to the output file.

- alignment - An Alignment object
- stream - output stream
- **interleave - if None (default): interleave if columns > 1000**
  if True: use interleaved format if False: do not use interleaved format

write_alignments(*stream*, *alignments*)

Write alignments to the output file, and return the number of alignments.

alignments - A list or iterator returning Alignment objects

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.nexus.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Nexus alignment iterator.

fmt: str | None = 'Nexus'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.phylip module

Bio.Align support for the alignment format for input files for PHYLIP tools.

You are expected to use this module via the Bio.Align functions.

class Bio.Align.phylip.**AlignmentWriter**(*target*)

Bases: *AlignmentWriter*

Clustalw alignment writer.

fmt: str | None = 'PHYLIP'

format_alignment(*alignment*)

Return a string with a single alignment in the Phylip format.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.phylip.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Reads a Phylip alignment file and returns an Alignment iterator.

Record names are limited to at most 10 characters.

The parser determines from the file contents if the file format is sequential or interleaved, and parses the file accordingly.

For more information on the file format, please see: http://evolution.genetics.washington.edu/phylip/doc/sequence.html http://evolution.genetics.washington.edu/phylip/doc/main.html#inputfiles

fmt: str | None = 'PHYLIP'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.psl module

Bio.Align support for the "psl" pairwise alignment format.

The Pattern Space Layout (PSL) format, described by UCSC, stores a series of pairwise alignments in a single file. Typically they are used for transcript to genome alignments. PSL files store the alignment positions and alignment scores, but do not store the aligned sequences.

See http://genome.ucsc.edu/FAQ/FAQformat.html#format2

You are expected to use this module via the Bio.Align functions.

Coordinates in the PSL format are defined in terms of zero-based start positions (like Python) and aligning region sizes.

A minimal aligned region of length one and starting at first position in the source sequence would have `start == 0` and `size == 1`.

As we can see in this example, `start + size` will give one more than the zero-based end position. We can therefore manipulate `start` and `start + size` as python list slice boundaries.

class Bio.Align.psl.**AlignmentWriter**(*target*, *header=True*, *mask=None*, *wildcard='N'*)

Bases: *AlignmentWriter*

Alignment file writer for the Pattern Space Layout (PSL) file format.

fmt: str | None = 'PSL'

__init__(*target*, *header=True*, *mask=None*, *wildcard='N'*)

Create an AlignmentWriter object.

Arguments:

- target - output stream or file name
- **header - If True (default), write the PSL header consisting of** five lines containing the PSL format version and a header for each column. If False, suppress the PSL header, resulting in a simple tab-delimited file.
- **mask - Specify if repeat regions in the target sequence are** masked and should be reported in the *repMatches* field of the PSL file instead of in the *matches* field. Acceptable values are None : no masking (default); "lower": masking by lower-case characters; "upper": masking by upper-case characters.
- **wildcard - Report alignments to the wildcard character in the** target or query sequence in the *nCount* field of the PSL file instead of in the *matches*, *mismatches*, or *repMatches* fields. Default value is 'N'.

write_header(*stream*, *alignments*)

Write the PSL header.

format_alignment(*alignment*)

Return a string with a single alignment formatted as one PSL line.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.psl.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Alignment iterator for Pattern Space Layout (PSL) files.

Each line in the file contains one pairwise alignment, which are loaded and returned incrementally. Alignment score information such as the number of matches and mismatches are stored as attributes of each alignment.

fmt: str | None = 'PSL'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.sam module

Bio.Align support for the "sam" pairwise alignment format.

The Sequence Alignment/Map (SAM) format, created by Heng Li and Richard Durbin at the Wellcome Trust Sanger Institute, stores a series of alignments to the genome in a single file. Typically they are used for next-generation sequencing data. SAM files store the alignment positions for mapped sequences, and may also store the aligned sequences and other information associated with the sequence.

See http://www.htslib.org/ for more information.

You are expected to use this module via the Bio.Align functions.

Coordinates in the SAM format are defined in terms of one-based start positions; the parser converts these to zero-based coordinates to be consistent with Python and other alignment formats.

class Bio.Align.sam.**AlignmentWriter**(*target*, *md=False*)

Bases: *AlignmentWriter*

Alignment file writer for the Sequence Alignment/Map (SAM) file format.

fmt: str | None = 'SAM'

__init__(*target*, *md=False*)

Create an AlignmentWriter object.

Arguments:

- **md - If True, calculate the MD tag from the alignment and include it** in the output. If False (default), do not include the MD tag in the output.

write_header(*stream*, *alignments*)

Write the SAM header.

format_alignment(*alignment*, *md=None*)

Return a string with a single alignment formatted as one SAM line.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

class Bio.Align.sam.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Alignment iterator for Sequence Alignment/Map (SAM) files.

Each line in the file contains one genomic alignment, which are loaded and returned incrementally. The following columns are stored as attributes of the alignment:

- flag: The FLAG combination of bitwise flags;
- mapq: Mapping Quality (only stored if available)

- **rnext: Reference sequence name of the primary alignment of the next read** in the alignment (only stored if available)
- **pnext: Zero-based position of the primary alignment of the next read in** the template (only stored if available)
- tlen: signed observed template length (only stored if available)

Other information associated with the alignment by its tags are stored in the annotations attribute of each alignment.

Any hard clipping (clipped sequences not present in the query sequence) are stored as 'hard_clip_left' and 'hard_clip_right' in the annotations dictionary attribute of the query sequence record.

The sequence quality, if available, is stored as 'phred_quality' in the letter_annotations dictionary attribute of the query sequence record.

fmt: str | None = 'SAM'

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.stockholm module

Bio.Align support for alignment files in the Stockholm file format.

You are expected to use this module via the Bio.Align functions.

For example, consider this alignment from PFAM for the HAT helix motif:

```
# STOCKHOLM 1.0
#=GF ID   HAT
#=GF AC   PF02184.18
#=GF DE   HAT (Half-A-TPR) repeat
#=GF AU   SMART;
#=GF SE   Alignment kindly provided by SMART
#=GF GA   21.00 21.00;
#=GF TC   21.00 21.00;
#=GF NC   20.90 20.90;
#=GF BM   hmmbuild HMM.ann SEED.ann
#=GF SM   hmmsearch -Z 57096847 -E 1000 --cpu 4 HMM pfamseq
#=GF TP   Repeat
#=GF CL   CL0020
#=GF RN   [1]
#=GF RM   9478129
#=GF RT   The HAT helix, a repetitive motif implicated in RNA processing.
#=GF RA   Preker PJ, Keller W;
#=GF RL   Trends Biochem Sci 1998;23:15-16.
#=GF DR   INTERPRO; IPR003107;
#=GF DR   SMART; HAT;
#=GF DR   SO; 0001068; polypeptide_repeat;
#=GF CC   The HAT (Half A TPR) repeat is found in several RNA processing
#=GF CC   proteins [1].
#=GF SQ   3
#=GS CRN_DROME/191-222     AC P17886.2
#=GS CLF1_SCHPO/185-216    AC P87312.1
```

(continues on next page)

(continued from previous page)

```
#=GS CLF1_SCHPO/185-216    DR PDB; 3JB9 R; 185-216;
#=GS O16376_CAEEL/201-233  AC O16376.2
CRN_DROME/191-222                KEIDRAREIYERFVYVH.PDVKNWIKFARFEES
CLF1_SCHPO/185-216               HENERARGIYERFVVVH.PEVTNWLRWARFEEE
#=GR CLF1_SCHPO/185-216    SS    --HHHHHHHHHHHHHHS.--HHHHHHHHHHHHH
O16376_CAEEL/201-233             KEIDRARSVYQRFLHVHGINVQNWIKYAKFEER
#=GC SS_cons                     --HHHHHHHHHHHHHHS.--HHHHHHHHHHHHH
#=GC seq_cons                    KEIDRARuIYERFVaVH.P-VpNWIKaARFEEc
//
```

Parsing this file using Bio.Align stores the alignment, its annotations, as well as the sequences and their annotations:

```
>>> from Bio.Align import stockholm
>>> alignments = stockholm.AlignmentIterator("Stockholm/example.sth")
>>> alignment = next(alignments)
>>> alignment.shape
(3, 33)
>>> alignment[0]
'KEIDRAREIYERFVYVH-PDVKNWIKFARFEES'
```

Alignment meta-data are stored in alignment.annotations:

```
>>> alignment.annotations["accession"]
'PF02184.18'
>>> alignment.annotations["references"][0]["title"]
'The HAT helix, a repetitive motif implicated in RNA processing.'
```

Annotations of alignment columns are stored in alignment.column_annotations:

```
>>> alignment.column_annotations["consensus secondary structure"]
'--HHHHHHHHHHHHHHS.--HHHHHHHHHHHHH'
```

Sequences and their annotations are stored in alignment.sequences:

```
>>> alignment.sequences[0].id
'CRN_DROME/191-222'
>>> alignment.sequences[0].seq
Seq('KEIDRAREIYERFVYVHPDVKNWIKFARFEES')
>>> alignment.sequences[1].letter_annotations["secondary structure"]
'--HHHHHHHHHHHHHHS--HHHHHHHHHHHHH'
```

Slicing specific columns of an alignment will slice any per-column-annotations:

```
>>> alignment.column_annotations["consensus secondary structure"]
'--HHHHHHHHHHHHHHS.--HHHHHHHHHHHHH'
>>> part_alignment = alignment[:,10:20]
>>> part_alignment.column_annotations["consensus secondary structure"]
'HHHHHHS.--'
```

class Bio.Align.stockholm.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Alignment iterator for alignment files in the Stockholm format.

The file may contain multiple concatenated alignments, which are loaded and returned incrementally.

Alignment meta-data (lines starting with #=GF) are stored in the dictionary alignment.annotations. Column annotations (lines starting with #=GC) are stored in the dictionary alignment.column_annotations. Sequence names are stored in record.id. Sequence record meta-data (lines starting with #=GS) are stored in the dictionary record.annotations. Sequence letter annotations (lines starting with #=GR) are stored in the dictionary record.letter_annotations.

Wrap-around alignments are not supported - each sequence must be on a single line.

For more information on the file format, please see: http://sonnhammer.sbc.su.se/Stockholm.html https://en.wikipedia.org/wiki/Stockholm_format

```
fmt:  str | None = 'Stockholm'
```

```
gf_mapping = {'**':  '**', 'AC': 'accession', 'AU': 'author', 'BM': 'build method',
'CB': 'calibration method', 'CC': 'comment', 'CL': 'clan', 'DE': 'definition', 'GA':
'gathering method', 'ID': 'identifier', 'NC': 'noise cutoff', 'PI': 'previous
identifier', 'SE': 'source of seed', 'SM': 'search method', 'SS': 'source of
structure', 'TC': 'trusted cutoff', 'TP': 'type', 'WK': 'wikipedia'}
```

```
gr_mapping = {'AS': 'active site', 'CSA': 'Catalytic Site Atlas', 'IN': 'intron',
'LI': 'ligand binding', 'PP': 'posterior probability', 'SA': 'surface
accessibility', 'SS': 'secondary structure', 'TM': 'transmembrane', 'pAS': 'active
site - Pfam predicted', 'sAS': 'active site - from SwissProt'}
```

```
gc_mapping = {'2L3J_B_SS': '2L3J B SS', 'AS_cons':  'consensus active site', 'CORE':
'CORE', 'CSA_cons':  'consensus Catalytic Site Atlas', 'IN_cons':  'consensus
intron', 'LI_cons':  'consensus ligand binding', 'MM': 'model mask', 'PK': 'PK',
'PK_SS': 'PK SS', 'PP_cons':  'consensus posterior probability', 'RF': 'reference
coordinate annotation', 'RNA_elements':  'RNA elements', 'RNA_ligand_AdoCbl':  'RNA
ligand AdoCbl', 'RNA_ligand_AqCbl':  'RNA ligand AqCbl', 'RNA_ligand_FMN': 'RNA
ligand FMN', 'RNA_ligand_Guanidinium':  'RNA ligand Guanidinium', 'RNA_ligand_SAM':
'RNA ligand SAM', 'RNA_ligand_THF_1':  'RNA ligand THF 1', 'RNA_ligand_THF_2':  'RNA
ligand THF 2', 'RNA_ligand_TPP': 'RNA ligand TPP', 'RNA_ligand_preQ1':  'RNA ligand
preQ1', 'RNA_motif_k_turn':  'RNA motif k turn', 'RNA_structural_element':  'RNA
structural element', 'RNA_structural_elements':  'RNA structural elements',
'Repeat_unit':  'Repeat unit', 'SA_cons':  'consensus surface accessibility',
'SS_cons':  'consensus secondary structure', 'TM_cons':  'consensus transmembrane',
'cons':  'cons', 'pAS_cons':  'consensus active site - Pfam predicted', 'sAS_cons':
'consensus active site - from SwissProt', 'scorecons':  'consensus score',
'scorecons_70':  'consensus score 70', 'scorecons_80':  'consensus score 80',
'scorecons_90':  'consensus score 90', 'seq_cons':  'consensus sequence'}
```

```
gs_mapping = {'AC': 'accession', 'LO': 'look', 'OC': 'organism classification',
'OS': 'organism'}
```

```
__abstractmethods__ = frozenset({})
```

```
__annotations__ = {'fmt':  'Optional[str]'}
```

```
key = 'IN'
```

```
keyword = 'cons'
```

```
value = 'intron'
```

class Bio.Align.stockholm.**AlignmentWriter**(*target*)

Bases: *AlignmentWriter*

Alignment file writer for the Stockholm file format.

gf_mapping = {'': '**', 'accession': 'AC', 'author': 'AU', 'build method': 'BM', 'calibration method': 'CB', 'clan': 'CL', 'comment': 'CC', 'definition': 'DE', 'gathering method': 'GA', 'identifier': 'ID', 'noise cutoff': 'NC', 'previous identifier': 'PI', 'search method': 'SM', 'source of seed': 'SE', 'source of structure': 'SS', 'trusted cutoff': 'TC', 'type': 'TP', 'wikipedia': 'WK'}**

gs_mapping = {'accession': 'AC', 'look': 'LO', 'organism': 'OS', 'organism classification': 'OC'}

gr_mapping = {'Catalytic Site Atlas': 'CSA', 'active site': 'AS', 'active site - Pfam predicted': 'pAS', 'active site - from SwissProt': 'sAS', 'intron': 'IN', 'ligand binding': 'LI', 'posterior probability': 'PP', 'secondary structure': 'SS', 'surface accessibility': 'SA', 'transmembrane': 'TM'}

gc_mapping = {'2L3J B SS': '2L3J_B_SS', 'CORE': 'CORE', 'PK': 'PK', 'PK SS': 'PK_SS', 'RNA elements': 'RNA_elements', 'RNA ligand AdoCbl': 'RNA_ligand_AdoCbl', 'RNA ligand AqCbl': 'RNA_ligand_AqCbl', 'RNA ligand FMN': 'RNA_ligand_FMN', 'RNA ligand Guanidinium': 'RNA_ligand_Guanidinium', 'RNA ligand SAM': 'RNA_ligand_SAM', 'RNA ligand THF 1': 'RNA_ligand_THF_1', 'RNA ligand THF 2': 'RNA_ligand_THF_2', 'RNA ligand TPP': 'RNA_ligand_TPP', 'RNA ligand preQ1': 'RNA_ligand_preQ1', 'RNA motif k turn': 'RNA_motif_k_turn', 'RNA structural element': 'RNA_structural_element', 'RNA structural elements': 'RNA_structural_elements', 'Repeat unit': 'Repeat_unit', 'cons': 'cons', 'consensus Catalytic Site Atlas': 'CSA_cons', 'consensus active site': 'AS_cons', 'consensus active site - Pfam predicted': 'pAS_cons', 'consensus active site - from SwissProt': 'sAS_cons', 'consensus intron': 'IN_cons', 'consensus ligand binding': 'LI_cons', 'consensus posterior probability': 'PP_cons', 'consensus score': 'scorecons', 'consensus score 70': 'scorecons_70', 'consensus score 80': 'scorecons_80', 'consensus score 90': 'scorecons_90', 'consensus secondary structure': 'SS_cons', 'consensus sequence': 'seq_cons', 'consensus surface accessibility': 'SA_cons', 'consensus transmembrane': 'TM_cons', 'model mask': 'MM', 'reference coordinate annotation': 'RF'}

fmt: str | None = 'Stockholm'

format_alignment(*alignment*)

Return a string with a single alignment in the Stockholm format.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Bio.Align.tabular module

Bio.Align support for tabular output from BLAST or FASTA.

This module contains a parser for tabular output from BLAST run with the '-outfmt 7' argument, as well as tabular output from William Pearson's FASTA alignment tools using the '-m 8CB' or '-m 8CC' arguments.

class Bio.Align.tabular.**State**(*value*)

Bases: Enum

Enumerate alignment states needed when parsing a BTOP string.

MATCH = 1

QUERY_GAP = 2

TARGET_GAP = 3

NONE = 4

class Bio.Align.tabular.**AlignmentIterator**(*source*)

Bases: *AlignmentIterator*

Alignment iterator for tabular output from BLAST or FASTA.

For reading (pairwise) alignments from tabular output generated by BLAST run with the '-outfmt 7' argument, as well as tabular output generated by William Pearson's FASTA alignment programs with the '-m 8CB' or '-m 8CC' output formats.

fmt: str | None = 'Tabular'

parse_btop(*btop*)

Parse a BTOP string and return alignment coordinates.

A BTOP (Blast trace-back operations) string is used by BLAST to describe a sequence alignment.

parse_cigar(*cigar*)

Parse a CIGAR string and return alignment coordinates.

A CIGAR string, as defined by the SAM Sequence Alignment/Map format, describes a sequence alignment as a series of lengths and operation (alignment/insertion/deletion) codes.

__abstractmethods__ = frozenset({})

__annotations__ = {'fmt': 'Optional[str]'}

Module contents

Code for dealing with sequence alignments.

One of the most important things in this module is the MultipleSeqAlignment class, used in the Bio.AlignIO module.

class Bio.Align.**AlignmentCounts**(*gaps*, *identities*, *mismatches*)

Bases: tuple

__getnewargs__()

Return self as a plain tuple. Used by copy and pickle.

__match_args__ = ('gaps', 'identities', 'mismatches')

static __new__(*_cls*, *gaps*, *identities*, *mismatches*)

Create new instance of AlignmentCounts(gaps, identities, mismatches)

__repr__()

Return a nicely formatted representation string

__slots__ = ()

gaps

Alias for field number 0

identities

Alias for field number 1

mismatches

Alias for field number 2

class Bio.Align.**MultipleSeqAlignment**(*records*, *alphabet=None*, *annotations=None*, *column_annotations=None*)

Bases: object

Represents a classical multiple sequence alignment (MSA).

By this we mean a collection of sequences (usually shown as rows) which are all the same length (usually with gap characters for insertions or padding). The data can then be regarded as a matrix of letters, with well defined columns.

You would typically create an MSA by loading an alignment file with the AlignIO module:

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("Clustalw/opuntia.aln", "clustal")
>>> print(align)
Alignment with 7 rows and 156 columns
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.1|AF191
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273291|gb|AF191665.1|AF191
```

In some respects you can treat these objects as lists of SeqRecord objects, each representing a row of the alignment. Iterating over an alignment gives the SeqRecord object for each row:

```
>>> len(align)
7
>>> for record in align:
...     print("%s %i" % (record.id, len(record)))
...
gi|6273285|gb|AF191659.1|AF191 156
gi|6273284|gb|AF191658.1|AF191 156
gi|6273287|gb|AF191661.1|AF191 156
gi|6273286|gb|AF191660.1|AF191 156
gi|6273290|gb|AF191664.1|AF191 156
gi|6273289|gb|AF191663.1|AF191 156
gi|6273291|gb|AF191665.1|AF191 156
```

You can also access individual rows as SeqRecord objects via their index:

```
>>> print(align[0].id)
gi|6273285|gb|AF191659.1|AF191
>>> print(align[-1].id)
gi|6273291|gb|AF191665.1|AF191
```

And extract columns as strings:

```
>>> print(align[:, 1])
AAAAAAA
```

Or, take just the first ten columns as a sub-alignment:

```
>>> print(align[:, :10])
Alignment with 7 rows and 10 columns
TATACATTAA gi|6273285|gb|AF191659.1|AF191
TATACATTAA gi|6273284|gb|AF191658.1|AF191
TATACATTAA gi|6273287|gb|AF191661.1|AF191
TATACATAAA gi|6273286|gb|AF191660.1|AF191
TATACATTAA gi|6273290|gb|AF191664.1|AF191
TATACATTAA gi|6273289|gb|AF191663.1|AF191
TATACATTAA gi|6273291|gb|AF191665.1|AF191
```

Combining this alignment slicing with alignment addition allows you to remove a section of the alignment. For example, taking just the first and last ten columns:

```
>>> print(align[:, :10] + align[:, -10:])
Alignment with 7 rows and 20 columns
TATACATTAAGTGTACCAGA gi|6273285|gb|AF191659.1|AF191
TATACATTAAGTGTACCAGA gi|6273284|gb|AF191658.1|AF191
TATACATTAAGTGTACCAGA gi|6273287|gb|AF191661.1|AF191
TATACATAAAGTGTACCAGA gi|6273286|gb|AF191660.1|AF191
TATACATTAAGTGTACCAGA gi|6273290|gb|AF191664.1|AF191
TATACATTAAGTATACCAGA gi|6273289|gb|AF191663.1|AF191
TATACATTAAGTGTACCAGA gi|6273291|gb|AF191665.1|AF191
```

Note - This object does NOT attempt to model the kind of alignments used in next generation sequencing with multiple sequencing reads which are much shorter than the alignment, and where there is usually a consensus or reference sequence with special status.

__init__(*records*, *alphabet=None*, *annotations=None*, *column_annotations=None*)

Initialize a new MultipleSeqAlignment object.

Arguments:

- **records - A list (or iterator) of SeqRecord objects, whose**
  sequences are all the same length. This may be an empty list.
- **alphabet - For backward compatibility only; its value should always**
  be None.
- annotations - Information about the whole alignment (dictionary).
- **column_annotations - Per column annotation (restricted dictionary).**
  This holds Python sequences (lists, strings, tuples) whose length matches the number of columns. A typical use would be a secondary structure consensus string.

You would normally load a MSA from a file using Bio.AlignIO, but you can do this from a list of SeqRecord objects too:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> a = SeqRecord(Seq("AAAACGT"), id="Alpha")
>>> b = SeqRecord(Seq("AAA-CGT"), id="Beta")
>>> c = SeqRecord(Seq("AAAAGGT"), id="Gamma")
>>> align = MultipleSeqAlignment([a, b, c],
...                              annotations={"tool": "demo"},
...                              column_annotations={"stats": "CCCXCCC"})
>>> print(align)
Alignment with 3 rows and 7 columns
AAAACGT Alpha
AAA-CGT Beta
AAAAGGT Gamma
>>> align.annotations
{'tool': 'demo'}
>>> align.column_annotations
{'stats': 'CCCXCCC'}
```

property column_annotations

Dictionary of per-letter-annotation for the sequence.

__str__()

Return a multi-line string summary of the alignment.

This output is intended to be readable, but large alignments are shown truncated. A maximum of 20 rows (sequences) and 50 columns are shown, with the record identifiers. This should fit nicely on a single screen. e.g.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> a = SeqRecord(Seq("ACTGCTAGCTAG"), id="Alpha")
>>> b = SeqRecord(Seq("ACT-CTAGCTAG"), id="Beta")
>>> c = SeqRecord(Seq("ACTGCTAGATAG"), id="Gamma")
>>> align = MultipleSeqAlignment([a, b, c])
>>> print(align)
Alignment with 3 rows and 12 columns
ACTGCTAGCTAG Alpha
ACT-CTAGCTAG Beta
ACTGCTAGATAG Gamma
```

See also the alignment's format method.

__repr__()

Return a representation of the object for debugging.

The representation cannot be used with eval() to recreate the object, which is usually possible with simple python objects. For example:

<Bio.Align.MultipleSeqAlignment instance (2 records of length 14) at a3c184c>

The hex string is the memory address of the object, see help(id). This provides a simple way to visually distinguish alignments of the same size.

__format__(*format_spec*)

Return the alignment as a string in the specified file format.

The format should be a lower case string supported as an output format by Bio.AlignIO (such as "fasta", "clustal", "phylip", "stockholm", etc), which is used to turn the alignment into a string.

e.g.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> a = SeqRecord(Seq("ACTGCTAGCTAG"), id="Alpha", description="")
>>> b = SeqRecord(Seq("ACT-CTAGCTAG"), id="Beta", description="")
>>> c = SeqRecord(Seq("ACTGCTAGATAG"), id="Gamma", description="")
>>> align = MultipleSeqAlignment([a, b, c])
>>> print(format(align, "fasta"))
>Alpha
ACTGCTAGCTAG
>Beta
ACT-CTAGCTAG
>Gamma
ACTGCTAGATAG

>>> print(format(align, "phylip"))
 3 12
Alpha      ACTGCTAGCT AG
Beta       ACT-CTAGCT AG
Gamma      ACTGCTAGAT AG
```

__iter__()

Iterate over alignment rows as SeqRecord objects.

e.g.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> a = SeqRecord(Seq("ACTGCTAGCTAG"), id="Alpha")
>>> b = SeqRecord(Seq("ACT-CTAGCTAG"), id="Beta")
>>> c = SeqRecord(Seq("ACTGCTAGATAG"), id="Gamma")
>>> align = MultipleSeqAlignment([a, b, c])
>>> for record in align:
...     print(record.id)
...     print(record.seq)
...
Alpha
ACTGCTAGCTAG
Beta
ACT-CTAGCTAG
Gamma
ACTGCTAGATAG
```

__len__()

Return the number of sequences in the alignment.

Use len(alignment) to get the number of sequences (i.e. the number of rows), and alignment.get_alignment_length() to get the length of the longest sequence (i.e. the number of columns).

This is easy to remember if you think of the alignment as being like a list of SeqRecord objects.

get_alignment_length()

Return the maximum length of the alignment.

All objects in the alignment should (hopefully) have the same length. This function will go through and find this length by finding the maximum length of sequences in the alignment.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> a = SeqRecord(Seq("ACTGCTAGCTAG"), id="Alpha")
>>> b = SeqRecord(Seq("ACT-CTAGCTAG"), id="Beta")
>>> c = SeqRecord(Seq("ACTGCTAGATAG"), id="Gamma")
>>> align = MultipleSeqAlignment([a, b, c])
>>> align.get_alignment_length()
12
```

If you want to know the number of sequences in the alignment, use len(align) instead:

```
>>> len(align)
3
```

extend(*records*)

Add more SeqRecord objects to the alignment as rows.

They must all have the same length as the original alignment. For example,

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> a = SeqRecord(Seq("AAAACGT"), id="Alpha")
>>> b = SeqRecord(Seq("AAA-CGT"), id="Beta")
>>> c = SeqRecord(Seq("AAAAGGT"), id="Gamma")
>>> d = SeqRecord(Seq("AAAACGT"), id="Delta")
>>> e = SeqRecord(Seq("AAA-GGT"), id="Epsilon")
```

First we create a small alignment (three rows):

```
>>> align = MultipleSeqAlignment([a, b, c])
>>> print(align)
Alignment with 3 rows and 7 columns
AAAACGT Alpha
AAA-CGT Beta
AAAAGGT Gamma
```

Now we can extend this alignment with another two rows:

```
>>> align.extend([d, e])
>>> print(align)
Alignment with 5 rows and 7 columns
AAAACGT Alpha
AAA-CGT Beta
```

(continues on next page)

(continued from previous page)

```
AAAAGGT Gamma
AAAACGT Delta
AAA-GGT Epsilon
```

Because the alignment object allows iteration over the rows as SeqRecords, you can use the extend method with a second alignment (provided its sequences have the same length as the original alignment).

append(*record*)

Add one more SeqRecord object to the alignment as a new row.

This must have the same length as the original alignment (unless this is the first record).

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("Clustalw/opuntia.aln", "clustal")
>>> print(align)
Alignment with 7 rows and 156 columns
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.
↪1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.
↪1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.
↪1|AF191
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.
↪1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.
↪1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.
↪1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273291|gb|AF191665.
↪1|AF191
>>> len(align)
7
```

We'll now construct a dummy record to append as an example:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> dummy = SeqRecord(Seq("N"*156), id="dummy")
```

Now append this to the alignment,

```
>>> align.append(dummy)
>>> print(align)
Alignment with 8 rows and 156 columns
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.
↪1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.
↪1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.
↪1|AF191
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.
↪1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.
```

(continues on next page)

(continued from previous page)

```
↪1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.
↪1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273291|gb|AF191665.
↪1|AF191
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN...NNN dummy
>>> len(align)
8
```

__add__(*other*)

Combine two alignments with the same number of rows by adding them.

If you have two multiple sequence alignments (MSAs), there are two ways to think about adding them - by row or by column. Using the extend method adds by row. Using the addition operator adds by column. For example,

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> a1 = SeqRecord(Seq("AAAAC"), id="Alpha")
>>> b1 = SeqRecord(Seq("AAA-C"), id="Beta")
>>> c1 = SeqRecord(Seq("AAAAG"), id="Gamma")
>>> a2 = SeqRecord(Seq("GT"), id="Alpha")
>>> b2 = SeqRecord(Seq("GT"), id="Beta")
>>> c2 = SeqRecord(Seq("GT"), id="Gamma")
>>> left = MultipleSeqAlignment([a1, b1, c1],
...                             annotations={"tool": "demo", "name": "start"},
...                             column_annotations={"stats": "CCCXC"})
>>> right = MultipleSeqAlignment([a2, b2, c2],
...                              annotations={"tool": "demo", "name": "end"},
...                              column_annotations={"stats": "CC"})
```

Now, let's look at these two alignments:

```
>>> print(left)
Alignment with 3 rows and 5 columns
AAAAC Alpha
AAA-C Beta
AAAAG Gamma
>>> print(right)
Alignment with 3 rows and 2 columns
GT Alpha
GT Beta
GT Gamma
```

And add them:

```
>>> combined = left + right
>>> print(combined)
Alignment with 3 rows and 7 columns
AAAACGT Alpha
AAA-CGT Beta
AAAAGGT Gamma
```

For this to work, both alignments must have the same number of records (here they both have 3 rows):

```
>>> len(left)
3
>>> len(right)
3
>>> len(combined)
3
```

The individual rows are SeqRecord objects, and these can be added together. Refer to the SeqRecord documentation for details of how the annotation is handled. This example is a special case in that both original alignments shared the same names, meaning when the rows are added they also get the same name.

Any common annotations are preserved, but differing annotation is lost. This is the same behaviour used in the SeqRecord annotations and is designed to prevent accidental propagation of inappropriate values:

```
>>> combined.annotations
{'tool': 'demo'}
```

Similarly any common per-column-annotations are combined:

```
>>> combined.column_annotations
{'stats': 'CCCXCCC'}
```

__getitem__(*index*)

Access part of the alignment.

Depending on the indices, you can get a SeqRecord object (representing a single row), a Seq object (for a single column), a string (for a single character) or another alignment (representing some part or all of the alignment).

align[r,c] gives a single character as a string align[r] gives a row as a SeqRecord align[r,:] gives a row as a SeqRecord align[:,c] gives a column as a Seq

align[:] and align[:,:] give a copy of the alignment

Anything else gives a sub alignment, e.g. align[0:2] or align[0:2,:] uses only row 0 and 1 align[:,1:3] uses only columns 1 and 2 align[0:2,1:3] uses only rows 0 & 1 and only cols 1 & 2

We'll use the following example alignment here for illustration:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> a = SeqRecord(Seq("AAAACGT"), id="Alpha")
>>> b = SeqRecord(Seq("AAA-CGT"), id="Beta")
>>> c = SeqRecord(Seq("AAAAGGT"), id="Gamma")
>>> d = SeqRecord(Seq("AAAACGT"), id="Delta")
>>> e = SeqRecord(Seq("AAA-GGT"), id="Epsilon")
>>> align = MultipleSeqAlignment([a, b, c, d, e])
```

You can access a row of the alignment as a SeqRecord using an integer index (think of the alignment as a list of SeqRecord objects here):

```
>>> first_record = align[0]
>>> print("%s %s" % (first_record.id, first_record.seq))
```

(continues on next page)

(continued from previous page)

```
Alpha AAAACGT
>>> last_record = align[-1]
>>> print("%s %s" % (last_record.id, last_record.seq))
Epsilon AAA-GGT
```

You can also access use python's slice notation to create a sub-alignment containing only some of the SeqRecord objects:

```
>>> sub_alignment = align[2:5]
>>> print(sub_alignment)
Alignment with 3 rows and 7 columns
AAAAGGT Gamma
AAAACGT Delta
AAA-GGT Epsilon
```

This includes support for a step, i.e. align[start:end:step], which can be used to select every second sequence:

```
>>> sub_alignment = align[::2]
>>> print(sub_alignment)
Alignment with 3 rows and 7 columns
AAAACGT Alpha
AAAAGGT Gamma
AAA-GGT Epsilon
```

Or to get a copy of the alignment with the rows in reverse order:

```
>>> rev_alignment = align[::-1]
>>> print(rev_alignment)
Alignment with 5 rows and 7 columns
AAA-GGT Epsilon
AAAACGT Delta
AAAAGGT Gamma
AAA-CGT Beta
AAAACGT Alpha
```

You can also use two indices to specify both rows and columns. Using simple integers gives you the entry as a single character string. e.g.

```
>>> align[3, 4]
'C'
```

This is equivalent to:

```
>>> align[3][4]
'C'
```

or:

```
>>> align[3].seq[4]
'C'
```

To get a single column (as a string) use this syntax:

```
>>> align[:, 4]
'CCGCG'
```

Or, to get part of a column,

```
>>> align[1:3, 4]
'CG'
```

However, in general you get a sub-alignment,

```
>>> print(align[1:5, 3:6])
Alignment with 4 rows and 3 columns
-CG Beta
AGG Gamma
ACG Delta
-GG Epsilon
```

This should all seem familiar to anyone who has used the NumPy array or matrix objects.

__delitem__(*index*)

Delete SeqRecord by index or multiple SeqRecords by slice.

sort(*key=None*, *reverse=False*)

Sort the rows (SeqRecord objects) of the alignment in place.

This sorts the rows alphabetically using the SeqRecord object id by default. The sorting can be controlled by supplying a key function which must map each SeqRecord to a sort value.

This is useful if you want to add two alignments which use the same record identifiers, but in a different order. For example,

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> align1 = MultipleSeqAlignment([
...              SeqRecord(Seq("ACGT"), id="Human"),
...              SeqRecord(Seq("ACGG"), id="Mouse"),
...              SeqRecord(Seq("ACGC"), id="Chicken"),
...          ])
>>> align2 = MultipleSeqAlignment([
...              SeqRecord(Seq("CGGT"), id="Mouse"),
...              SeqRecord(Seq("CGTT"), id="Human"),
...              SeqRecord(Seq("CGCT"), id="Chicken"),
...          ])
```

If you simple try and add these without sorting, you get this:

```
>>> print(align1 + align2)
Alignment with 3 rows and 8 columns
ACGTCGGT <unknown id>
ACGGCGTT <unknown id>
ACGCCGCT Chicken
```

Consult the SeqRecord documentation which explains why you get a default value when annotation like the identifier doesn't match up. However, if we sort the alignments first, then add them we get the desired result:

```
>>> align1.sort()
>>> align2.sort()
>>> print(align1 + align2)
Alignment with 3 rows and 8 columns
ACGCCGCT Chicken
ACGTCGTT Human
ACGGCGGT Mouse
```

As an example using a different sort order, you could sort on the GC content of each sequence.

```
>>> from Bio.SeqUtils import gc_fraction
>>> print(align1)
Alignment with 3 rows and 4 columns
ACGC Chicken
ACGT Human
ACGG Mouse
>>> align1.sort(key = lambda record: gc_fraction(record.seq))
>>> print(align1)
Alignment with 3 rows and 4 columns
ACGT Human
ACGC Chicken
ACGG Mouse
```

There is also a reverse argument, so if you wanted to sort by ID but backwards:

```
>>> align1.sort(reverse=True)
>>> print(align1)
Alignment with 3 rows and 4 columns
ACGG Mouse
ACGT Human
ACGC Chicken
```

property substitutions

Return an Array with the number of substitutions of letters in the alignment.

As an example, consider a multiple sequence alignment of three DNA sequences:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> seq1 = SeqRecord(Seq("ACGT"), id="seq1")
>>> seq2 = SeqRecord(Seq("A--A"), id="seq2")
>>> seq3 = SeqRecord(Seq("ACGT"), id="seq3")
>>> seq4 = SeqRecord(Seq("TTTC"), id="seq4")
>>> alignment = MultipleSeqAlignment([seq1, seq2, seq3, seq4])
>>> print(alignment)
Alignment with 4 rows and 4 columns
ACGT seq1
A--A seq2
ACGT seq3
TTTC seq4
```

```
>>> m = alignment.substitutions
>>> print(m)
    A   C   G   T
A 3.0 0.5 0.0 2.5
C 0.5 1.0 0.0 2.0
G 0.0 0.0 1.0 1.0
T 2.5 2.0 1.0 1.0
```

Note that the matrix is symmetric, with counts divided equally on both sides of the diagonal. For example, the total number of substitutions between A and T in the alignment is 3.5 + 3.5 = 7.

Any weights associated with the sequences are taken into account when calculating the substitution matrix. For example, given the following multiple sequence alignment:

```
GTATC  0.5
AT--C  0.8
CTGTC  1.0
```

For the first column we have:

```
('A', 'G') : 0.5 * 0.8 = 0.4
('C', 'G') : 0.5 * 1.0 = 0.5
('A', 'C') : 0.8 * 1.0 = 0.8
```

property alignment

Return an Alignment object based on the MultipleSeqAlignment object.

This makes a copy of each SeqRecord with a gap-less sequence. Any future changes to the original records in the MultipleSeqAlignment will not affect the new records in the Alignment.

class Bio.Align.**Alignment**(*sequences*, *coordinates=None*)

Bases: object

Represents a sequence alignment.

An Alignment object has a *.sequences* attribute storing the sequences (Seq, MutableSeq, SeqRecord, or string objects) that were aligned, as well as a *.coordinates* attribute storing the sequence coordinates defining the alignment as a NumPy array.

Other commonly used attributes (which may or may not be present) are:

- **annotations - A dictionary with annotations describing the** alignment;
- **column_annotations - A dictionary with annotations describing each** column in the alignment;
- score - The alignment score.

classmethod infer_coordinates(*lines*)

Infer the coordinates from a printed alignment (DEPRECATED).

This method is primarily employed in Biopython's alignment parsers, though it may be useful for other purposes.

For an alignment consisting of N sequences, printed as N lines with the same number of columns, where gaps are represented by dashes, this method will calculate the sequence coordinates that define the alignment. The coordinates are returned as a NumPy array of integers, and can be used to create an Alignment object.

This is an example for the alignment of three sequences TAGGCATACGTG, AACGTACGT, and ACGCATACTTG, with gaps in the second and third sequence:

```
>>> from Bio.Align import Alignment
>>> lines = ["TAGGCATACGTG",
...          "AACG--TACGT-",
...          "-ACGCATACTTG",
...         ]
>>> sequences = [line.replace("-", "") for line in lines]
>>> sequences
['TAGGCATACGTG', 'AACGTACGT', 'ACGCATACTTG']
>>> coordinates = Alignment.infer_coordinates(lines)
>>> print(coordinates)
[[ 0  1  4  6 11 12]
 [ 0  1  4  4  9  9]
 [ 0  0  3  5 10 11]]
>>> alignment = Alignment(sequences, coordinates)
```

classmethod parse_printed_alignment(*lines*)

Infer the sequences and coordinates from a printed alignment.

This method is primarily employed in Biopython's alignment parsers, though it may be useful for other purposes.

For an alignment consisting of N sequences, printed as N lines with the same number of columns, where gaps are represented by dashes, this method will calculate the sequence coordinates that define the alignment. It returns the tuple (sequences, coordinates), where sequences is the list of N sequences after removing the gaps, and the coordinates is a 2D NumPy array of integers. Together, the sequences and coordinates can be used to create an Alignment object.

This is an example for the alignment of three sequences TAGGCATACGTG, AACGTACGT, and ACGCATACTTG, with gaps in the second and third sequence. Note that the input sequences are bytes objects.

```
>>> from Bio.Align import Alignment
>>> from Bio.Seq import Seq
>>> lines = [b"TAGGCATACGTG",
...          b"AACG--TACGT-",
...          b"-ACGCATACTTG",
...         ]
>>> sequences, coordinates = Alignment.parse_printed_alignment(lines)
>>> sequences
[b'TAGGCATACGTG', b'AACGTACGT', b'ACGCATACTTG']
>>> print(coordinates)
[[ 0  1  4  6 11 12]
 [ 0  1  4  4  9  9]
 [ 0  0  3  5 10 11]]
>>> sequences = [Seq(sequence) for sequence in sequences]
>>> sequences
[Seq('TAGGCATACGTG'), Seq('AACGTACGT'), Seq('ACGCATACTTG')]
>>> alignment = Alignment(sequences, coordinates)
>>> print(alignment)
                 0 TAGGCATACGTG 12
                 0 AACG--TACGT-  9
                 0 -ACGCATACTTG 11
```

__init__(*sequences*, *coordinates=None*)

Initialize a new Alignment object.

Arguments:

- **sequences - A list of the sequences (Seq, MutableSeq, SeqRecord,**
  or string objects) that were aligned.
- **coordinates - The sequence coordinates that define the alignment.**
  If None (the default value), assume that the sequences align to each other without any gaps.

__array__(*dtype=None*)

__add__(*other*)

Combine two alignments by adding them row-wise.

For example,

```
>>> import numpy as np
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import Alignment
>>> a1 = SeqRecord(Seq("AAAAC"), id="Alpha")
>>> b1 = SeqRecord(Seq("AAAC"), id="Beta")
>>> c1 = SeqRecord(Seq("AAAAG"), id="Gamma")
>>> a2 = SeqRecord(Seq("GTT"), id="Alpha")
>>> b2 = SeqRecord(Seq("TT"), id="Beta")
>>> c2 = SeqRecord(Seq("GT"), id="Gamma")
>>> left = Alignment([a1, b1, c1],
...                  coordinates=np.array([[0, 3, 4, 5],
...                                        [0, 3, 3, 4],
...                                        [0, 3, 4, 5]]))
>>> left.annotations = {"tool": "demo", "name": "start"}
>>> left.column_annotations = {"stats": "CCCXC"}
>>> right = Alignment([a2, b2, c2],
...                   coordinates=np.array([[0, 1, 2, 3],
...                                         [0, 0, 1, 2],
...                                         [0, 1, 1, 2]]))
>>> right.annotations = {"tool": "demo", "name": "end"}
>>> right.column_annotations = {"stats": "CXC"}
```

Now, let's look at these two alignments:

```
>>> print(left)
Alpha             0 AAAAC 5
Beta              0 AAA-C 4
Gamma             0 AAAAG 5

>>> print(right)
Alpha             0 GTT 3
Beta              0 -TT 2
Gamma             0 G-T 2
```

And add them:

```
>>> combined = left + right
>>> print(combined)
Alpha             0 AAAACGTT 8
Beta              0 AAA-C-TT 6
Gamma             0 AAAAGG-T 7
```

For this to work, both alignments must have the same number of sequences (here they both have 3 rows):

```
>>> len(left)
3
>>> len(right)
3
>>> len(combined)
3
```

The sequences are SeqRecord objects, and these can be added together. Refer to the SeqRecord documentation for details of how the annotation is handled. This example is a special case in that both original alignments shared the same names, meaning when the rows are added they also get the same name.

Any common annotations are preserved, but differing annotation is lost. This is the same behavior used in the SeqRecord annotations and is designed to prevent accidental propagation of inappropriate values:

```
>>> combined.annotations
{'tool': 'demo'}
```

Similarly any common per-column-annotations are combined:

```
>>> combined.column_annotations
{'stats': 'CCCXCCXC'}
```

property frequencies

Return the frequency of each letter in each column of the alignment.

Gaps are represented by a dash ("-") character. For example,

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> aligner.mode = "global"
>>> alignments = aligner.align("GACCTG", "CGATCG")
>>> alignment = alignments[0]
>>> print(alignment)
target            0 -GACCT-G 6
                  0 -||--|-| 8
query             0 CGA--TCG 6

>>> alignment.frequencies
{'-': array([1., 0., 0., 1., 1., 0., 1., 0.]), 'G': array([0., 2., 0., 0., 0.,␣
↪0., 0., 2.]), 'A': array([0., 0., 2., 0., 0., 0., 0., 0.]), 'C': array([1., 0.
↪, 0., 1., 1., 0., 1., 0.]), 'T': array([0., 0., 0., 0., 0., 2., 0., 0.])}
>>> aligner.mode = "local"
>>> alignments = aligner.align("GACCTG", "CGATCG")
>>> alignment = alignments[0]
>>> print(alignment)
target            0 GACCT-G 6
```

(continues on next page)

(continued from previous page)

```
                  0 ||--|-| 7
query             1 GA--TCG 6

>>> alignment.frequencies
{'G': array([2., 0., 0., 0., 0., 0., 2.]), 'A': array([0., 2., 0., 0., 0., 0.,␣
↪0.]), 'C': array([0., 0., 1., 1., 0., 1., 0.]), 'T': array([0., 0., 0., 0., 2.
↪, 0., 0.]), '-': array([0., 0., 1., 1., 0., 1., 0.])}
```

property target

Return self.sequences[0] for a pairwise alignment.

property query

Return self.sequences[1] for a pairwise alignment.

__eq__(*other*)

Check if two Alignment objects specify the same alignment.

__ne__(*other*)

Check if two Alignment objects have different alignments.

__lt__(*other*)

Check if self should come before other.

__le__(*other*)

Check if self should come before or is equal to other.

__gt__(*other*)

Check if self should come after other.

__ge__(*other*)

Check if self should come after or is equal to other.

__getitem__(*key*)

Return self[key].

Indices of the form

self[:, :]

return a copy of the Alignment object;

self[:, i:] self[:, :j] self[:, i:j] self[:, iterable] (where iterable returns integers)

return a new Alignment object spanning the selected columns;

self[k, i] self[k, i:] self[k, :j] self[k, i:j] self[k, iterable] (where iterable returns integers) self[k] (equivalent to self[k, :])

return a string with the aligned sequence (including gaps) for the selected columns, where k = 0 represents the target and k = 1 represents the query sequence; and

self[:, i]

returns a string with the selected column in the alignment.

```
>>> from Bio.Align import PairwiseAligner
>>> aligner = PairwiseAligner()
>>> alignments = aligner.align("ACCGGTTT", "ACGGGTT")
```

(continues on next page)

(continued from previous page)

```
>>> alignment = alignments[0]
>>> print(alignment)
target            0 ACCGG-TTT 8
                  0 ||-||-||- 9
query             0 AC-GGGTT- 7

>>> alignment[0, :]
'ACCGG-TTT'
>>> alignment[1, :]
'AC-GGGTT-'
>>> alignment[0]
'ACCGG-TTT'
>>> alignment[1]
'AC-GGGTT-'
>>> alignment[0, 1:-2]
'CCGG-T'
>>> alignment[1, 1:-2]
'C-GGGT'
>>> alignment[0, (1, 5, 2)]
'C-C'
>>> alignment[1, ::2]
'A-GT-'
>>> alignment[1, range(0, 9, 2)]
'A-GT-'
>>> alignment[:, 0]
'AA'
>>> alignment[:, 5]
'-G'
>>> alignment[:, 1:]
<Alignment object (2 rows x 8 columns) at 0x...>
>>> print(alignment[:, 1:])
target            1 CCGG-TTT 8
                  0 |-||-||- 8
query             1 C-GGGTT- 7

>>> print(alignment[:, 2:])
target            2 CGG-TTT 8
                  0 -||-||- 7
query             2 -GGGTT- 7

>>> print(alignment[:, 3:])
target            3 GG-TTT 8
                  0 ||-||- 6
query             2 GGGTT- 7

>>> print(alignment[:, 3:-1])
target            3 GG-TT 7
                  0 ||-|| 5
query             2 GGGTT 7

>>> print(alignment[:, ::2])
target            0 ACGTT 5
```

(continues on next page)

(continued from previous page)

```
                   0 |-||- 5
query              0 A-GT- 3

>>> print(alignment[:, range(1, 9, 2)])
target             0 CG-T 3
                   0 ||-| 4
query              0 CGGT 4

>>> print(alignment[:, (2, 7, 3)])
target             0 CTG 3
                   0 -|| 3
query              0 -TG 2
```

__format__(*format_spec*)

Return the alignment as a string in the specified file format.

Wrapper for self.format().

format(*fmt=''*, **args*, ***kwargs*)

Return the alignment as a string in the specified file format.

Arguments:

- **fmt - File format. Acceptable values are an empty string to**
  create a human-readable representation of the alignment, or any of the alignment file formats supported by *Bio.Align* (some have not yet been implemented).

All other arguments are passed to the format-specific writer functions:

- **mask - PSL format only. Specify if repeat regions in the target**
  sequence are masked and should be reported in the *repMatches* field of the PSL file instead of in the *matches* field. Acceptable values are None : no masking (default); "lower": masking by lower-case characters; "upper": masking by upper-case characters.
- **wildcard - PSL format only. Report alignments to the wildcard**
  character in the target or query sequence in the *nCount* field of the PSL file instead of in the *matches*, *misMatches*, or *repMatches* fields. Default value is 'N'.
- **md - SAM format only. If True, calculate the MD tag from**
  the alignment and include it in the output. If False (default), do not include the MD tag in the output.

__str__()

Return a human-readable string representation of the alignment.

For sequence alignments, each line has at most 80 columns. The first 10 columns show the (possibly truncated) sequence name, which may be the id attribute of a SeqRecord, or otherwise 'target' or 'query' for pairwise alignments. The next 10 columns show the sequence coordinate, using zero-based counting as usual in Python. The remaining 60 columns shown the sequence, using dashes to represent gaps. At the end of the alignment, the end coordinates are shown on the right of the sequence, again in zero-based coordinates.

Pairwise alignments have an additional line between the two sequences showing whether the sequences match ('|') or mismatch ('.'), or if there is a gap ('-'). The coordinates shown for this line are the column indices, which can be useful when extracting a subalignment.

For example,

```
>>> from Bio.Align import PairwiseAligner
>>> aligner = PairwiseAligner()
```

```
>>> seqA = "TTAACCCCATTTG"
>>> seqB = "AAGCCCCTTT"
>>> seqC = "AAAGGGGCTT"
```

```
>>> alignments = aligner.align(seqA, seqB)
>>> len(alignments)
1
>>> alignment = alignments[0]
>>> print(alignment)
target            0 TTAA-CCCCATTTG 13
                  0 --||-||||-|||- 14
query             0 --AAGCCCC-TTT- 10
```

Note that seqC is the reverse complement of seqB. Aligning it to the reverse strand gives the same alignment, but the query coordinates are switched:

```
>>> alignments = aligner.align(seqA, seqC, strand="-")
>>> len(alignments)
1
>>> alignment = alignments[0]
>>> print(alignment)
target            0 TTAA-CCCCATTTG 13
                  0 --||-||||-|||- 14
query            10 --AAGCCCC-TTT-  0
```

__repr__()

Return a representation of the alignment, including its shape.

The representation cannot be used with eval() to recreate the object, which is usually possible with simple python objects. For example:

<Alignment object (2 rows x 14 columns) at 0x10403d850>

The hex string is the memory address of the object and can be used to distinguish different Alignment objects. See help(id) for more information.

```
>>> import numpy as np
>>> from Bio.Align import Alignment
>>> alignment = Alignment(("ACCGT", "ACGT"),
...                       coordinates = np.array([[0, 2, 3, 5],
...                                               [0, 2, 2, 4],
...                                              ]))
>>> print(alignment)
target            0 ACCGT 5
                  0 ||-|| 5
query             0 AC-GT 4

>>> alignment
<Alignment object (2 rows x 5 columns) at 0x...>
```

__len__()

Return the number of sequences in the alignment.

property length

Return the alignment length, i.e. the number of columns when printed..

The alignment length is the number of columns in the alignment when it is printed, and is equal to the sum of the number of matches, number of mismatches, and the total length of gaps in the target and query. Sequence sections beyond the aligned segment are not included in the number of columns.

For example,

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> aligner.mode = "global"
>>> alignments = aligner.align("GACCTG", "CGATCG")
>>> alignment = alignments[0]
>>> print(alignment)
target            0 -GACCT-G 6
                  0 -||--|-| 8
query             0 CGA--TCG 6

>>> alignment.length
8
>>> aligner.mode = "local"
>>> alignments = aligner.align("GACCTG", "CGATCG")
>>> alignment = alignments[0]
>>> print(alignment)
target            0 GACCT-G 6
                  0 ||--|-| 7
query             1 GA--TCG 6

>>> len(alignment)
2
>>> alignment.length
7
```

property shape

Return the shape of the alignment as a tuple of two integer values.

The first integer value is the number of sequences in the alignment as returned by len(alignment), which is always 2 for pairwise alignments.

The second integer value is the number of columns in the alignment when it is printed, and is equal to the sum of the number of matches, number of mismatches, and the total length of gaps in the target and query. Sequence sections beyond the aligned segment are not included in the number of columns.

For example,

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> aligner.mode = "global"
>>> alignments = aligner.align("GACCTG", "CGATCG")
>>> alignment = alignments[0]
>>> print(alignment)
target            0 -GACCT-G 6
                  0 -||--|-| 8
query             0 CGA--TCG 6

```

(continues on next page)

(continued from previous page)

```
>>> len(alignment)
2
>>> alignment.shape
(2, 8)
>>> aligner.mode = "local"
>>> alignments = aligner.align("GACCTG", "CGATCG")
>>> alignment = alignments[0]
>>> print(alignment)
target            0 GACCT-G 6
                  0 ||--|-| 7
query             1 GA--TCG 6

>>> len(alignment)
2
>>> alignment.shape
(2, 7)
```

property aligned

Return the indices of subsequences aligned to each other.

This property returns the start and end indices of subsequences in the target and query sequence that were aligned to each other. If the alignment between target (t) and query (q) consists of N chunks, you get two tuples of length N:

(((t_start1, t_end1), (t_start2, t_end2), . . . , (t_startN, t_endN)),
((q_start1, q_end1), (q_start2, q_end2), . . . , (q_startN, q_endN)))

For example,

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> alignments = aligner.align("GAACT", "GAT")
>>> alignment = alignments[0]
>>> print(alignment)
target            0 GAACT 5
                  0 ||--| 5
query             0 GA--T 3

>>> alignment.aligned
array([[[0, 2],
        [4, 5]],

       [[0, 2],
        [2, 3]]])
>>> alignment = alignments[1]
>>> print(alignment)
target            0 GAACT 5
                  0 |-|-| 5
query             0 G-A-T 3

>>> alignment.aligned
array([[[0, 1],
        [2, 3],
```

(continues on next page)

(continued from previous page)

```
        [4, 5]],

       [[0, 1],
        [1, 2],
        [2, 3]]])
```

Note that different alignments may have the same subsequences aligned to each other. In particular, this may occur if alignments differ from each other in terms of their gap placement only:

```
>>> aligner.mismatch_score = -10
>>> alignments = aligner.align("AAACAAA", "AAAGAAA")
>>> len(alignments)
2
>>> print(alignments[0])
target            0 AAAC-AAA 7
                  0 |||--||| 8
query             0 AAA-GAAA 7

>>> alignments[0].aligned
array([[[0, 3],
        [4, 7]],

       [[0, 3],
        [4, 7]]])
>>> print(alignments[1])
target            0 AAA-CAAA 7
                  0 |||--||| 8
query             0 AAAG-AAA 7

>>> alignments[1].aligned
array([[[0, 3],
        [4, 7]],

       [[0, 3],
        [4, 7]]])
```

The property can be used to identify alignments that are identical to each other in terms of their aligned sequences.

property indices

Return the sequence index of each lettter in the alignment.

This property returns a 2D NumPy array with the sequence index of each letter in the alignment. Gaps are indicated by -1. The array has the same number of rows and columns as the alignment, as given by *self.shape*.

For example,

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> aligner.mode = "local"
```

```
>>> alignments = aligner.align("GAACTGG", "AATG")
>>> alignment = alignments[0]
>>> print(alignment)
target            1 AACTG 6
                  0 ||-|| 5
query             0 AA-TG 4

>>> alignment.indices
array([[ 1,  2,  3,  4,  5],
       [ 0,  1, -1,  2,  3]])
>>> alignment = alignments[1]
>>> print(alignment)
target            1 AACTGG 7
                  0 ||-|-| 6
query             0 AA-T-G 4

>>> alignment.indices
array([[ 1,  2,  3,  4,  5,  6],
       [ 0,  1, -1,  2, -1,  3]])
```

```
>>> alignments = aligner.align("GAACTGG", "CATT", strand="-")
>>> alignment = alignments[0]
>>> print(alignment)
target            1 AACTG 6
                  0 ||-|| 5
query             4 AA-TG 0

>>> alignment.indices
array([[ 1,  2,  3,  4,  5],
       [ 3,  2, -1,  1,  0]])
>>> alignment = alignments[1]
>>> print(alignment)
target            1 AACTGG 7
                  0 ||-|-| 6
query             4 AA-T-G 0

>>> alignment.indices
array([[ 1,  2,  3,  4,  5,  6],
       [ 3,  2, -1,  1, -1,  0]])
```

property inverse_indices

Return the alignment column index for each letter in each sequence.

This property returns a list of 1D NumPy arrays; the number of arrays is equal to the number of aligned sequences, and the length of each array is equal to the length of the corresponding sequence. For each letter in each sequence, the array contains the corresponding column index in the alignment. Letters not included in the alignment are indicated by -1.

For example,

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> aligner.mode = "local"
```

```
>>> alignments = aligner.align("GAACTGG", "AATG")
>>> alignment = alignments[0]
>>> print(alignment)
target            1 AACTG 6
                  0 ||-|| 5
query             0 AA-TG 4

>>> alignment.inverse_indices
[array([-1,  0,  1,  2,  3,  4, -1]), array([0, 1, 3, 4])]
>>> alignment = alignments[1]
>>> print(alignment)
target            1 AACTGG 7
                  0 ||-|-| 6
query             0 AA-T-G 4

>>> alignment.inverse_indices
[array([-1,  0,  1,  2,  3,  4,  5]), array([0, 1, 3, 5])]
>>> alignments = aligner.align("GAACTGG", "CATT", strand="-")
>>> alignment = alignments[0]
>>> print(alignment)
target            1 AACTG 6
                  0 ||-|| 5
query             4 AA-TG 0

>>> alignment.inverse_indices
[array([-1,  0,  1,  2,  3,  4, -1]), array([4, 3, 1, 0])]
>>> alignment = alignments[1]
>>> print(alignment)
target            1 AACTGG 7
                  0 ||-|-| 6
query             4 AA-T-G 0

>>> alignment.inverse_indices
[array([-1,  0,  1,  2,  3,  4,  5]), array([5, 3, 1, 0])]
```

sort(*key=None*, *reverse=False*)

Sort the sequences of the alignment in place.

By default, this sorts the sequences alphabetically using their id attribute if available, or by their sequence contents otherwise. For example,

```
>>> from Bio.Align import PairwiseAligner
>>> aligner = PairwiseAligner()
>>> aligner.gap_score = -1
>>> alignments = aligner.align("AATAA", "AAGAA")
>>> len(alignments)
1
>>> alignment = alignments[0]
>>> print(alignment)
target            0 AATAA 5
                  0 ||.|| 5
query             0 AAGAA 5
```

(continues on next page)

(continued from previous page)

```
>>> alignment.sort()
>>> print(alignment)
target            0 AAGAA 5
                  0 ||.|| 5
query             0 AATAA 5
```

Alternatively, a key function can be supplied that maps each sequence to a sort value. For example, you could sort on the GC content of each sequence.

```
>>> from Bio.SeqUtils import gc_fraction
>>> alignment.sort(key=gc_fraction)
>>> print(alignment)
target            0 AATAA 5
                  0 ||.|| 5
query             0 AAGAA 5
```

You can reverse the sort order by passing *reverse=True*:

```
>>> alignment.sort(key=gc_fraction, reverse=True)
>>> print(alignment)
target            0 AAGAA 5
                  0 ||.|| 5
query             0 AATAA 5
```

The sequences are now sorted by decreasing GC content value.

map(*alignment*)

Map the alignment to self.target and return the resulting alignment.

Here, self.query and alignment.target are the same sequence.

A typical example is where self is the pairwise alignment between a chromosome and a transcript, the argument is the pairwise alignment between the transcript and a sequence (e.g., as obtained by RNA-seq), and we want to find the alignment of the sequence to the chromosome:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> aligner.mode = 'local'
>>> aligner.open_gap_score = -1
>>> aligner.extend_gap_score = 0
>>> chromosome = "AAAAAAAACCCCCCCAAAAAAAAAAAGGGGGGAAAAAAAA"
>>> transcript = "CCCCCCCGGGGGG"
>>> alignments1 = aligner.align(chromosome, transcript)
>>> len(alignments1)
1
>>> alignment1 = alignments1[0]
>>> print(alignment1)
target            8 CCCCCCCAAAAAAAAAAAGGGGGG 32
                  0 |||||||-----------|||||| 24
query             0 CCCCCCC-----------GGGGGG 13

>>> sequence = "CCCCGGGG"
>>> alignments2 = aligner.align(transcript, sequence)
>>> len(alignments2)
```

(continues on next page)

(continued from previous page)

```
1
>>> alignment2 = alignments2[0]
>>> print(alignment2)
target            3 CCCCGGGG 11
                  0 ||||||||  8
query             0 CCCCGGGG  8

>>> alignment = alignment1.map(alignment2)
>>> print(alignment)
target           11 CCCCAAAAAAAAAAGGGG 30
                  0 ||||----------|||| 19
query             0 CCCC----------GGGG  8

>>> format(alignment, "psl")
'8\t0\t0\t0\t0\t1\t11\t+\tquery\t8\t0\t8\ttarget\t40\t11\t30\t2\t4,4,\t0,4,\
↪t11,26,\n'
```

Mapping the alignment does not depend on the sequence contents. If we delete the sequence contents, the same alignment is found in PSL format (though we obviously lose the ability to print the sequence alignment):

```
>>> alignment1.target = Seq(None, len(alignment1.target))
>>> alignment1.query = Seq(None, len(alignment1.query))
>>> alignment2.target = Seq(None, len(alignment2.target))
>>> alignment2.query = Seq(None, len(alignment2.query))
>>> alignment = alignment1.map(alignment2)
>>> format(alignment, "psl")
'8\t0\t0\t0\t0\t1\t11\t+\tquery\t8\t0\t8\ttarget\t40\t11\t30\t2\t4,4,\t0,4,\
↪t11,26,\n'
```

The map method can also be used to lift over an alignment between different genome assemblies. In this case, self is a DNA alignment between two genome assemblies, and the argument is an alignment of a transcript against one of the genome assemblies:

```
>>> np.set_printoptions(threshold=5)  # print 5 array elements per row
>>> chain = Align.read("Blat/panTro5ToPanTro6.over.chain", "chain")
>>> chain.sequences[0].id
'chr1'
>>> len(chain.sequences[0].seq)
228573443
>>> chain.sequences[1].id
'chr1'
>>> len(chain.sequences[1].seq)
224244399
>>> print(chain.coordinates)
[[122250000 122250400 122250400 ... 122909818 122909819 122909835]
 [111776384 111776784 111776785 ... 112019962 112019962 112019978]]
```

showing that the range 122250000:122909835 of chr1 on chimpanzee genome assembly panTro5 aligns to range 111776384:112019978 of chr1 of chimpanzee genome assembly panTro6.

```
>>> alignment = Align.read("Blat/est.panTro5.psl", "psl")
>>> alignment.sequences[0].id
```

(continues on next page)

(continued from previous page)

```
'chr1'
>>> len(alignment.sequences[0].seq)
228573443
>>> alignment.sequences[1].id
'DC525629'
>>> len(alignment.sequences[1].seq)
407
>>> print(alignment.coordinates)
[[122835789 122835847 122840993 122841145 122907212 122907314]
 [       32        90        90       242       242       344]]
```

This shows that nucleotide range 32:344 of expressed sequence tag DC525629 aligns to range 122835789:122907314 of chr1 of chimpanzee genome assembly panTro5.

Note that the target sequence chain.sequences[0].seq and the target sequence alignment.sequences[0] have the same length:

```
>>> len(chain.sequences[0].seq) == len(alignment.sequences[0].seq)
True
```

We swap the target and query of the chain such that the query of the chain corresponds to the target of alignment:

```
>>> chain = chain[::-1]
>>> chain.sequences[0].id
'chr1'
>>> len(chain.sequences[0].seq)
224244399
>>> chain.sequences[1].id
'chr1'
>>> len(chain.sequences[1].seq)
228573443
>>> print(chain.coordinates)
[[111776384 111776784 111776785 ... 112019962 112019962 112019978]
 [122250000 122250400 122250400 ... 122909818 122909819 122909835]]
```

Now we can get the coordinates of DC525629 against chimpanzee genome assembly panTro6 by calling map on the chain, with alignment as the argument:

```
>>> lifted_alignment = chain.map(alignment)
>>> lifted_alignment.sequences[0].id
'chr1'
>>> len(lifted_alignment.sequences[0].seq)
224244399
>>> lifted_alignment.sequences[1].id
'DC525629'
>>> len(lifted_alignment.sequences[1].seq)
407
>>> print(lifted_alignment.coordinates)
[[111982717 111982775 111987921 111988073 112009200 112009302]
 [       32        90        90       242       242       344]]
```

This shows that nucleotide range 32:344 of expressed sequence tag DC525629 aligns to range 111982717:112009302 of chr1 of chimpanzee genome assembly panTro6. Note that the genome span of

DC525629 on chimpanzee genome assembly panTro5 is 122907314 - 122835789 = 71525 bp, while on panTro6 the genome span is 112009302 - 111982717 = 26585 bp.

mapall(*alignments*)

Map each of the alignments to self, and return the mapped alignment.

property substitutions

Return an Array with the number of substitutions of letters in the alignment.

As an example, consider a sequence alignment of two RNA sequences:

```
>>> from Bio.Align import PairwiseAligner
>>> target =
↪"ATACTTACCTGGCAGGGGAGATACCATGATCACGAAGGTGGTTTTCCCAGGGCGAGGCTTATCCATTGCACTCCGGATGTGCTGACCCCTG
↪"  # human spliceosomal small nuclear RNA U1
>>> query =
↪"ATACTTACCTGACAGGGGAGGCACCATGATCACACAGGTGGTCCTCCCAGGGCGAGGCTCTTCCATTGCACTGCGGGAGGGTTGACCCCTG
↪"  # sea lamprey spliceosomal small RNA U1
>>> aligner = PairwiseAligner()
>>> aligner.gap_score = -10
>>> alignments = aligner.align(target, query)
>>> len(alignments)
1
>>> alignment = alignments[0]
>>> print(alignment)
target            0 ATACTTACCTGGCAGGGGAGATACCATGATCACGAAGGTGGTTTTCCCAGGGCGAGGCTT
                  0 ||||||||||.||||||||..||||||||||..|||||||..|||||||||||||||.
query             0 ATACTTACCTGACAGGGGAGGCACCATGATCACACAGGTGGTCCTCCCAGGGCGAGGCTC

target           60 ATCCATTGCACTCCGGATGTGCTGACCCCTGCGATTTCCCCAAATGTGGGAAACTCGACT
                  60 .||||||||||.|||..|.|.||||||||||||||||||||||||||||||||||||||
query             60 TTCCATTGCACTGCGGGAGGGTTGACCCCTGCGATTTCCCCAAATGTGGGAAACTCGACT

target          120 GCATAATTTGTGGTAGTGGGGGACTGCGTTCGCGCTTTCCCCTG 164
                120 |.|||||||||||||||||||||||||||||||.|||||.| 164
query           120 GTATAATTTGTGGTAGTGGGGGACTGCGTTCGCGCTATCCCCCG 164

>>> m = alignment.substitutions
>>> print(m)
     A    C    G    T
A 28.0  1.0  2.0  1.0
C  0.0 39.0  1.0  2.0
G  2.0  0.0 45.0  0.0
T  2.0  5.0  1.0 35.0
```

Note that the matrix is not symmetric: rows correspond to the target sequence, and columns to the query sequence. For example, the number of T's in the target sequence that are aligned to a C in the query sequence is

```
>>> m['T', 'C']
5.0
```

and the number of C's in the query sequence tat are aligned to a T in the query sequence is

```
>>> m['C', 'T']
2.0
```

For some applications (for example, to define a scoring matrix from the substitution matrix), a symmetric matrix may be preferred, which can be calculated as follows:

```
>>> m += m.transpose()
>>> m /= 2.0
>>> print(m)
     A    C    G    T
A 28.0  0.5  2.0  1.5
C  0.5 39.0  0.5  3.5
G  2.0  0.5 45.0  0.5
T  1.5  3.5  0.5 35.0
```

The matrix is now symmetric, with counts divided equally on both sides of the diagonal:

```
>>> m['C', 'T']
3.5
>>> m['T', 'C']
3.5
```

The total number of substitutions between T's and C's in the alignment is 3.5 + 3.5 = 7.

counts()

Return number of identities, mismatches, and gaps of a pairwise alignment.

```
>>> aligner = PairwiseAligner(mode='global', match_score=2, mismatch_score=-1)
>>> for alignment in aligner.align("TACCG", "ACG"):
...     print("Score = %.1f:" % alignment.score)
...     c = alignment.counts()  # namedtuple
...     print(f"{c.gaps} gaps, {c.identities} identities, {c.mismatches}␣
↪mismatches")
...     print(alignment)
...
Score = 6.0:
2 gaps, 3 identities, 0 mismatches
target            0 TACCG 5
                  0 -||-| 5
query             0 -AC-G 3

Score = 6.0:
2 gaps, 3 identities, 0 mismatches
target            0 TACCG 5
                  0 -|-|| 5
query             0 -A-CG 3
```

This classifies each pair of letters in a pairwise alignment into gaps, perfect matches, or mismatches. It has been defined as a method (not a property) so that it may in future take optional argument(s) allowing the behavior to be customized. These three values are returned as a namedtuple. This is calculated for all the pairs of sequences in the alignment.

reverse_complement()

Reverse-complement the alignment and return it.

```
>>> sequences = ["ATCG", "AAG", "ATC"]
>>> coordinates = np.array([[0, 2, 3, 4], [0, 2, 2, 3], [0, 2, 3, 3]])
>>> alignment = Alignment(sequences, coordinates)
>>> print(alignment)
                   0 ATCG 4
                   0 AA-G 3
                   0 ATC- 3

>>> rc_alignment = alignment.reverse_complement()
>>> print(rc_alignment)
                   0 CGAT 4
                   0 C-TT 3
                   0 -GAT 3
```

The attribute *column_annotations*, if present, is associated with the reverse-complemented alignment, with its values in reverse order.

```
>>> alignment.column_annotations = {"score": [3, 2, 2, 2]}
>>> rc_alignment = alignment.reverse_complement()
>>> print(rc_alignment.column_annotations)
{'score': [2, 2, 2, 3]}
```

__hash__ = None

class Bio.Align.**AlignmentsAbstractBaseClass**

Bases: ABC

Abstract base class for sequence alignments.

Most users will not need to use this class. It is used internally as a base class for the list-like Alignments class, and for the AlignmentIterator class in Bio.Align.interfaces, which itself is the abstract base class for the alignment parsers in Bio/Align/.

__iter__()

Iterate over the alignments as Alignment objects.

This method SHOULD NOT be overridden by any subclass.

abstract __next__()

Return the next alignment.

abstract rewind()

Rewind the iterator to let it loop over the alignments from the beginning.

abstract __len__()

Return the number of alignments.

__abstractmethods__ = frozenset({'__len__', '__next__', 'rewind'})

class Bio.Align.**Alignments**(*alignments=()*)

Bases: *AlignmentsAbstractBaseClass*, list

__init__(*alignments=()*)

__next__()

Return the next alignment.

rewind()

Rewind the iterator to let it loop over the alignments from the beginning.

__len__()

Return the number of alignments.

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.Align.**PairwiseAlignments**(*seqA*, *seqB*, *score*, *paths*)

Bases: *AlignmentsAbstractBaseClass*

Implements an iterator over pairwise alignments returned by the aligner.

This class also supports indexing, which is fast for increasing indices, but may be slow for random access of a large number of alignments.

Note that pairwise aligners can return an astronomical number of alignments, even for relatively short sequences, if they align poorly to each other. We therefore recommend to first check the number of alignments, accessible as len(alignments), which can be calculated quickly even if the number of alignments is very large.

__init__(*seqA*, *seqB*, *score*, *paths*)

Initialize a new PairwiseAlignments object.

Arguments:

- seqA - The first sequence, as a plain string, without gaps.
- seqB - The second sequence, as a plain string, without gaps.
- score - The alignment score.
- **paths - An iterator over the paths in the traceback matrix;**
  each path defines one alignment.

You would normally obtain a PairwiseAlignments object by calling aligner.align(seqA, seqB), where aligner is a PairwiseAligner object or a CodonAligner object.

__len__()

Return the number of alignments.

__getitem__(*index*)

__next__()

Return the next alignment.

rewind()

Rewind the iterator to let it loop over the alignments from the beginning.

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.Align.**PairwiseAligner**(*scoring=None*, ***kwargs*)

Bases: PairwiseAligner

Performs pairwise sequence alignment using dynamic programming.

This provides functions to get global and local alignments between two sequences. A global alignment finds the best concordance between all characters in two sequences. A local alignment finds just the subsequences that align the best.

To perform a pairwise sequence alignment, first create a PairwiseAligner object. This object stores the match and mismatch scores, as well as the gap scores. Typically, match scores are positive, while mismatch scores and gap scores are negative or zero. By default, the match score is 1, and the mismatch and gap scores are zero. Based on the values of the gap scores, a PairwiseAligner object automatically chooses the appropriate alignment algorithm (the Needleman-Wunsch, Smith-Waterman, Gotoh, or Waterman-Smith-Beyer global or local alignment algorithm).

Calling the "score" method on the aligner with two sequences as arguments will calculate the alignment score between the two sequences. Calling the "align" method on the aligner with two sequences as arguments will return a generator yielding the alignments between the two sequences.

Some examples:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> alignments = aligner.align("TACCG", "ACG")
>>> for alignment in sorted(alignments):
...     print("Score = %.1f:" % alignment.score)
...     print(alignment)
...
Score = 3.0:
target            0 TACCG 5
                  0 -|-|| 5
query             0 -A-CG 3

Score = 3.0:
target            0 TACCG 5
                  0 -||-| 5
query             0 -AC-G 3
```

Specify the aligner mode as local to generate local alignments:

```
>>> aligner.mode = 'local'
>>> alignments = aligner.align("TACCG", "ACG")
>>> for alignment in sorted(alignments):
...     print("Score = %.1f:" % alignment.score)
...     print(alignment)
...
Score = 3.0:
target            1 ACCG 5
                  0 |-|| 4
query             0 A-CG 3

Score = 3.0:
target            1 ACCG 5
                  0 ||-| 4
query             0 AC-G 3
```

Do a global alignment. Identical characters are given 2 points, 1 point is deducted for each non-identical character.

```
>>> aligner.mode = 'global'
>>> aligner.match_score = 2
>>> aligner.mismatch_score = -1
>>> for alignment in aligner.align("TACCG", "ACG"):
...     print("Score = %.1f:" % alignment.score)
```

(continues on next page)

(continued from previous page)

```
...     print(alignment)
...
Score = 6.0:
target            0 TACCG 5
                  0 -||-| 5
query             0 -AC-G 3

Score = 6.0:
target            0 TACCG 5
                  0 -|-|| 5
query             0 -A-CG 3
```

Same as above, except now 0.5 points are deducted when opening a gap, and 0.1 points are deducted when extending it.

```
>>> aligner.open_gap_score = -0.5
>>> aligner.extend_gap_score = -0.1
>>> aligner.target_end_gap_score = 0.0
>>> aligner.query_end_gap_score = 0.0
>>> for alignment in aligner.align("TACCG", "ACG"):
...     print("Score = %.1f:" % alignment.score)
...     print(alignment)
...
Score = 5.5:
target            0 TACCG 5
                  0 -|-|| 5
query             0 -A-CG 3

Score = 5.5:
target            0 TACCG 5
                  0 -||-| 5
query             0 -AC-G 3
```

The alignment function can also use known matrices already included in Biopython:

```
>>> from Bio.Align import substitution_matrices
>>> aligner = Align.PairwiseAligner()
>>> aligner.substitution_matrix = substitution_matrices.load("BLOSUM62")
>>> alignments = aligner.align("KEVLA", "EVL")
>>> alignments = list(alignments)
>>> print("Number of alignments: %d" % len(alignments))
Number of alignments: 1
>>> alignment = alignments[0]
>>> print("Score = %.1f" % alignment.score)
Score = 13.0
>>> print(alignment)
target            0 KEVLA 5
                  0 -|||- 5
query             0 -EVL- 3
```

You can also set the value of attributes directly during construction of the PairwiseAligner object by providing them as keyword arguments:

```
>>> aligner = Align.PairwiseAligner(mode='global', match_score=2, mismatch_score=-1)
>>> for alignment in aligner.align("TACCG", "ACG"):
...     print("Score = %.1f:" % alignment.score)
...     print(alignment)
...
Score = 6.0:
target            0 TACCG 5
                  0 -||-| 5
query             0 -AC-G 3

Score = 6.0:
target            0 TACCG 5
                  0 -|-|| 5
query             0 -A-CG 3
```

__init__(*scoring=None*, ***kwargs*)

Initialize a PairwiseAligner as specified by the keyword arguments.

If scoring is None, use the default scoring scheme match = 1.0, mismatch = 0.0, gap score = 0.0 If scoring is "blastn", "megablast", or "blastp", use the default substitution matrix and gap scores for BLASTN, MEGABLAST, or BLASTP, respectively.

Loops over the remaining keyword arguments and sets them as attributes on the object.

__setattr__(*key*, *value*)

Implement setattr(self, name, value).

align(*seqA*, *seqB*, *strand='+'*)

Return the alignments of two sequences using PairwiseAligner.

score(*seqA*, *seqB*, *strand='+'*)

Return the alignment score of two sequences using PairwiseAligner.

__getstate__()

__setstate__(*state*)

class Bio.Align.**CodonAligner**(*codon_table=None*, *anchor_len=10*)

Bases: CodonAligner

Aligns a nucleotide sequence to an amino acid sequence.

This class implements a dynamic programming algorithm to align a nucleotide sequence to an amino acid sequence.

__init__(*codon_table=None*, *anchor_len=10*)

Initialize a CodonAligner for a specific genetic code.

Arguments:

- codon_table - a CodonTable object representing the genetic code. If codon_table is None, the standard genetic code is used.

score(*seqA*, *seqB*)

Return the alignment score of a protein sequence and nucleotide sequence.

Arguments:

- seqA - the protein sequence of amino acids (plain string, Seq, MutableSeq, or SeqRecord).

- seqB - the nucleotide sequence (plain string, Seq, MutableSeq, or SeqRecord); both DNA and RNA sequences are accepted.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> aligner = CodonAligner()
>>> dna = SeqRecord(Seq('ATGTCTCGT'), id='dna')
>>> pro = SeqRecord(Seq('MSR'), id='pro')
>>> score = aligner.score(pro, dna)
>>> print(score)
3.0
>>> rna = SeqRecord(Seq('AUGUCUCGU'), id='rna')
>>> score = aligner.score(pro, rna)
>>> print(score)
3.0
```

This is an example with a frame shift in the DNA sequence:

```
>>> dna = "ATGCTGGGCTCGAACGAGTCCGTGTATGCCCTAAGCTGAGCCCGTCG"
>>> pro = "MLGSNESRVCPKLSPS"
>>> len(pro)
16
>>> aligner.frameshift_score = -3.0
>>> score = aligner.score(pro, dna)
>>> print(score)
13.0
```

In the following example, the position of the frame shift is ambiguous:

```
>>> dna = 'TTTAAAAAAAAAAATTT'
>>> pro = 'FKKKKF'
>>> len(pro)
6
>>> aligner.frameshift_score = -1.0
>>> alignments = aligner.align(pro, dna)
>>> print(alignments.score)
5.0
>>> len(alignments)
3
>>> print(next(alignments))
target            0 F  K  K  K   4
query             0 TTTAAAAAAAAA 12

target            4 K  F    6
query            11 AAATTT 17

>>> print(next(alignments))
target            0 F  K  K   3
query             0 TTTAAAAAA 9

target            3 K  K  F    6
query             8 AAAAAATTT 17

>>> print(next(alignments))
```

(continues on next page)

(continued from previous page)

```
target            0 F  K   2
query             0 TTTAAA 6

target            2 K  K  K  F    6
query             5 AAAAAAAAATTT 17

>>> print(next(alignments))
Traceback (most recent call last):
...
StopIteration
```

align(*seqA*, *seqB*)

Align a nucleotide sequence to its corresponding protein sequence.

Arguments:

- seqA - the protein sequence of amino acids (plain string, Seq, MutableSeq, or SeqRecord).
- seqB - the nucleotide sequence (plain string, Seq, MutableSeq, or SeqRecord); both DNA and RNA sequences are accepted.

Returns an iterator of Alignment objects.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> aligner = CodonAligner()
>>> dna = SeqRecord(Seq('ATGTCTCGT'), id='dna')
>>> pro = SeqRecord(Seq('MSR'), id='pro')
>>> alignments = aligner.align(pro, dna)
>>> alignment = alignments[0]
>>> print(alignment)
pro               0 M  S  R   3
dna               0 ATGTCTCGT 9

>>> rna = SeqRecord(Seq('AUGUCUCGU'), id='rna')
>>> alignments = aligner.align(pro, rna)
>>> alignment = alignments[0]
>>> print(alignment)
pro               0 M  S  R   3
rna               0 AUGUCUCGU 9
```

This is an example with a frame shift in the DNA sequence:

```
>>> dna = "ATGCTGGGCTCGAACGAGTCCGTGTATGCCCTAAGCTGAGCCCGTCG"
>>> pro = "MLGSNESRVCPKLSPS"
>>> alignments = aligner.align(pro, dna)
>>> alignment = alignments[0]
>>> print(alignment)
target            0 M  L  G  S  N  E  S   7
query             0 ATGCTGGGCTCGAACGAGTCC 21

target            7 R  V  C  P  K  L  S  P  S   16
query            20 CGTGTATGCCCTAAGCTGAGCCCGTCG 47
```

Bio.Align.**write**(*alignments*, *target*, *fmt*, **args*, ***kwargs*)

Write alignments to a file.

Arguments:

- alignments - An Alignments object, an iterator of Alignment objects, or a single Alignment.
- target - File or file-like object to write to, or filename as string.
- fmt - String describing the file format (case-insensitive).

Note if providing a file or file-like object, your code should close the target after calling this function, or call .flush(), to ensure the data gets flushed to disk.

Returns the number of alignments written (as an integer).

Bio.Align.**parse**(*source*, *fmt*)

Parse an alignment file and return an iterator over alignments.

Arguments:

- source - File or file-like object to read from, or filename as string.
- fmt - String describing the file format (case-insensitive).

Typical usage, opening a file to read in, and looping over the alignments:

```
>>> from Bio import Align
>>> filename = "Exonerate/exn_22_m_ner_cigar.exn"
>>> for alignment in Align.parse(filename, "exonerate"):
...     print("Number of sequences in alignment", len(alignment))
...     print("Alignment score:", alignment.score)
Number of sequences in alignment 2
Alignment score: 6150.0
Number of sequences in alignment 2
Alignment score: 502.0
Number of sequences in alignment 2
Alignment score: 440.0
```

For lazy-loading file formats such as bigMaf, for which the file contents is read on demand only, ensure that the file remains open while extracting alignment data.

You can use the Bio.Align.read(...) function when the file contains only one alignment.

Bio.Align.**read**(*handle*, *fmt*)

Parse a file containing one alignment, and return it.

Arguments:

- source - File or file-like object to read from, or filename as string.
- fmt - String describing the file format (case-insensitive).

This function is for use parsing alignment files containing exactly one alignment. For example, reading a Clustal file:

```
>>> from Bio import Align
>>> alignment = Align.read("Clustalw/opuntia.aln", "clustal")
>>> print("Alignment shape:", alignment.shape)
Alignment shape: (7, 156)
>>> for sequence in alignment.sequences:
```

(continues on next page)

(continued from previous page)

```
...     print(sequence.id, len(sequence))
gi|6273285|gb|AF191659.1|AF191 146
gi|6273284|gb|AF191658.1|AF191 148
gi|6273287|gb|AF191661.1|AF191 146
gi|6273286|gb|AF191660.1|AF191 146
gi|6273290|gb|AF191664.1|AF191 150
gi|6273289|gb|AF191663.1|AF191 150
gi|6273291|gb|AF191665.1|AF191 156
```

If the file contains no records, or more than one record, an exception is raised. For example:

```
>>> from Bio import Align
>>> filename = "Exonerate/exn_22_m_ner_cigar.exn"
>>> alignment = Align.read(filename, "exonerate")
Traceback (most recent call last):
    ...
ValueError: More than one alignment found in file
```

Use the Bio.Align.parse function if you want to read a file containing more than one alignment.

28.1.3 Bio.AlignIO package

Submodules

Bio.AlignIO.ClustalIO module

Bio.AlignIO support for "clustal" output from CLUSTAL W and other tools.

You are expected to use this module via the Bio.AlignIO functions (or the Bio.SeqIO functions if you want to work directly with the gapped sequences).

class Bio.AlignIO.ClustalIO.**ClustalWriter**(*handle*)

Bases: *SequentialAlignmentWriter*

Clustalw alignment writer.

write_alignment(*alignment*)

Use this to write (another) single alignment to an open file.

class Bio.AlignIO.ClustalIO.**ClustalIterator**(*handle*, *seq_count=None*)

Bases: *AlignmentIterator*

Clustalw alignment iterator.

__next__()

Parse the next alignment from the handle.

Bio.AlignIO.EmbossIO module

Bio.AlignIO support for "emboss" alignment output from EMBOSS tools.

You are expected to use this module via the Bio.AlignIO functions (or the Bio.SeqIO functions if you want to work directly with the gapped sequences).

This module contains a parser for the EMBOSS pairs/simple file format, for example from the alignret, water and needle tools.

class Bio.AlignIO.EmbossIO.**EmbossIterator**(*handle*, *seq_count=None*)

Bases: *AlignmentIterator*

Emboss alignment iterator.

For reading the (pairwise) alignments from EMBOSS tools in what they call the "pairs" and "simple" formats.

__next__()

Parse the next alignment from the handle.

__annotations__ = {}

Bio.AlignIO.FastaIO module

Bio.AlignIO support for "fasta-m10" output from Bill Pearson's FASTA tools.

You are expected to use this module via the Bio.AlignIO functions (or the Bio.SeqIO functions if you want to work directly with the gapped sequences).

This module contains a parser for the pairwise alignments produced by Bill Pearson's FASTA tools, for use from the Bio.AlignIO interface where it is referred to as the "fasta-m10" file format (as we only support the machine readable output format selected with the -m 10 command line option).

This module does NOT cover the generic "fasta" file format originally developed as an input format to the FASTA tools. The Bio.AlignIO and Bio.SeqIO both use the Bio.SeqIO.FastaIO module to deal with these files, which can also be used to store a multiple sequence alignments.

Bio.AlignIO.FastaIO.**FastaM10Iterator**(*handle*, *seq_count=None*)

Alignment iterator for the FASTA tool's pairwise alignment output.

This is for reading the pairwise alignments output by Bill Pearson's FASTA program when called with the -m 10 command line option for machine readable output. For more details about the FASTA tools, see the website http://fasta.bioch.virginia.edu/ and the paper:

W.R. Pearson & D.J. Lipman PNAS (1988) 85:2444-2448

This class is intended to be used via the Bio.AlignIO.parse() function by specifying the format as "fasta-m10" as shown in the following code:

```
from Bio import AlignIO
handle = ...
for a in AlignIO.parse(handle, "fasta-m10"):
    assert len(a) == 2, "Should be pairwise!"
    print("Alignment length %i" % a.get_alignment_length())
    for record in a:
        print("%s %s %s" % (record.seq, record.name, record.id))
```

Note that this is not a full blown parser for all the information in the FASTA output - for example, most of the header and all of the footer is ignored. Also, the alignments are not batched according to the input queries.

Also note that there can be up to about 30 letters of flanking region included in the raw FASTA output as contextual information. This is NOT part of the alignment itself, and is not included in the resulting MultipleSeqAlignment objects returned.

Bio.AlignIO.Interfaces module

AlignIO support module (not for general use).

Unless you are writing a new parser or writer for Bio.AlignIO, you should not use this module. It provides base classes to try and simplify things.

class Bio.AlignIO.Interfaces.**AlignmentIterator**(*handle*, *seq_count=None*)

Bases: object

Base class for building MultipleSeqAlignment iterators.

You should write a next() method to return Alignment objects. You may wish to redefine the __init__ method as well.

__init__(*handle*, *seq_count=None*)

Create an AlignmentIterator object.

Arguments:

- handle - input file
- count - optional, expected number of records per alignment Recommend for fasta file format.

Note when subclassing:

- there should be a single non-optional argument, the handle, and optional count IN THAT ORDER.
- you can add additional optional arguments.

__next__()

Return the next alignment in the file.

This method should be replaced by any derived class to do something useful.

__iter__()

Iterate over the entries as MultipleSeqAlignment objects.

Example usage for (concatenated) PHYLIP files:

```
with open("many.phy","r") as myFile:
    for alignment in PhylipIterator(myFile):
        print("New alignment:")
        for record in alignment:
            print(record.id)
            print(record.seq)
```

__annotations__ = {}

class Bio.AlignIO.Interfaces.**AlignmentWriter**(*handle*)

Bases: object

Base class for building MultipleSeqAlignment writers.

You should write a write_alignment() method. You may wish to redefine the __init__ method as well.

__init__(*handle*)

Initialize the class.

write_file(*alignments*)

Use this to write an entire file containing the given alignments.

Arguments:

- alignments - A list or iterator returning MultipleSeqAlignment objects

In general, this method can only be called once per file.

This method should be replaced by any derived class to do something useful. It should return the number of alignments..

clean(*text*)

Use this to avoid getting newlines in the output.

__annotations__ = {}

class Bio.AlignIO.Interfaces.**SequentialAlignmentWriter**(*handle*)

Bases: *AlignmentWriter*

Base class for building MultipleSeqAlignment writers.

This assumes each alignment can be simply appended to the file. You should write a write_alignment() method. You may wish to redefine the __init__ method as well.

__init__(*handle*)

Initialize the class.

write_file(*alignments*)

Use this to write an entire file containing the given alignments.

Arguments:

- alignments - A list or iterator returning MultipleSeqAlignment objects

In general, this method can only be called once per file.

write_header()

Use this to write any header.

This method should be replaced by any derived class to do something useful.

write_footer()

Use this to write any footer.

This method should be replaced by any derived class to do something useful.

write_alignment(*alignment*)

Use this to write a single alignment.

This method should be replaced by any derived class to do something useful.

__annotations__ = {}

Bio.AlignIO.MafIO module

Bio.AlignIO support for the "maf" multiple alignment format.

The Multiple Alignment Format, described by UCSC, stores a series of multiple alignments in a single file. It is suitable for whole-genome to whole-genome alignments, metadata such as source chromosome, start position, size, and strand can be stored.

See http://genome.ucsc.edu/FAQ/FAQformat.html#format5

You are expected to use this module via the Bio.AlignIO functions(or the Bio.SeqIO functions if you want to work directly with the gapped sequences).

Coordinates in the MAF format are defined in terms of zero-based start positions (like Python) and aligning region sizes.

A minimal aligned region of length one and starting at first position in the source sequence would have `start == 0` and `size == 1`.

As we can see on this example, `start + size` will give one more than the zero-based end position. We can therefore manipulate `start` and `start + size` as python list slice boundaries.

For an inclusive end coordinate, we need to use `end = start + size - 1`. A 1-column wide alignment would have `start == end`.

class Bio.AlignIO.MafIO.**MafWriter**(*handle*)

Bases: *SequentialAlignmentWriter*

Accepts a MultipleSeqAlignment object, writes a MAF file.

write_header()

Write the MAF header.

write_alignment(*alignment*)

Write a complete alignment to a MAF block.

Writes every SeqRecord in a MultipleSeqAlignment object to its own MAF block (beginning with an 'a' line, containing 's' lines).

__annotations__ = {}

Bio.AlignIO.MafIO.**MafIterator**(*handle*, *seq_count=None*)

Iterate over a MAF file handle as MultipleSeqAlignment objects.

Iterates over lines in a MAF file-like object (handle), yielding MultipleSeqAlignment objects. SeqRecord IDs generally correspond to species names.

class Bio.AlignIO.MafIO.**MafIndex**(*sqlite_file*, *maf_file*, *target_seqname*)

Bases: `object`

Index for a MAF file.

The index is a sqlite3 database that is built upon creation of the object if necessary, and queried when methods *search* or *get_spliced* are used.

__init__(*sqlite_file*, *maf_file*, *target_seqname*)

Indexes or loads the index of a MAF file.

close()

Close the file handle being used to read the data.

Once called, further use of the index won't work. The sole purpose of this method is to allow explicit handle closure - for example if you wish to delete the file, on Windows you must first close all open handles to that file.

search(*starts*, *ends*)

Search index database for MAF records overlapping ranges provided.

Returns *MultipleSeqAlignment* results in order by start, then end, then internal offset field.

starts should be a list of 0-based start coordinates of segments in the reference. *ends* should be the list of the corresponding segment ends (in the half-open UCSC convention: http://genome.ucsc.edu/blog/the-ucsc-genome-browser-coordinate-counting-systems/).

get_spliced(*starts*, *ends*, *strand=1*)

Return a multiple alignment of the exact sequence range provided.

Accepts two lists of start and end positions on target_seqname, representing exons to be spliced in silico. Returns a *MultipleSeqAlignment* of the desired sequences spliced together.

starts should be a list of 0-based start coordinates of segments in the reference. *ends* should be the list of the corresponding segment ends (in the half-open UCSC convention: http://genome.ucsc.edu/blog/the-ucsc-genome-browser-coordinate-counting-systems/).

To ask for the alignment portion corresponding to the first 100 nucleotides of the reference sequence, you would use `search([0], [100])`

__repr__()

Return a string representation of the index.

__len__()

Return the number of records in the index.

Bio.AlignIO.MauveIO module

Bio.AlignIO support for "xmfa" output from Mauve/ProgressiveMauve.

You are expected to use this module via the Bio.AlignIO functions (or the Bio.SeqIO functions if you want to work directly with the gapped sequences).

For example, consider a progressiveMauve alignment file containing the following:

```
#FormatVersion Mauve1
#Sequence1File      a.fa
#Sequence1Entry     1
#Sequence1Format    FastA
#Sequence2File      b.fa
#Sequence2Entry     2
#Sequence2Format    FastA
#Sequence3File      c.fa
#Sequence3Entry     3
#Sequence3Format    FastA
#BackboneFile       three.xmfa.bbcols
> 1:0-0 + a.fa
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
> 2:5417-5968 + b.fa
```

(continues on next page)

(continued from previous page)

```
TTTAAACATCCCTCGGCCCGTCGCCCTTTTATAATAGCAGTACGTGAGAGGAGCGCCCTAAGCTTTGGGAAATTCAAGC-
--------------------------------------------------------------------------------
CTGGAACGTACTTGCTGGTTTCGCTACTATTTCAAACAAGTTAGAGGCCGTTACCTCGGGCGAACGTATAAACCATTCTG
> 3:9476-10076 - c.fa
TTTAAACACCTTTTTGGATG--GCCCAGTTCGTTCAGTTGTG-GGGAGGAGATCGCCCCAAACGTATGGTGAGTCGGGCG
TTTCCTATAGCTATAGGACCAATCCACTTACCATACGCCCGGCGTCGCCCAGTCCGGTTCGGTACCCTCCATGACCCACG
------------------------------------------------------------AAATGAGGGCCCAGGGTATGCTT
=
> 2:5969-6015 + b.fa
-----------------------
GGGCGAACGTATAAACCATTCTG
> 3:9429-9476 - c.fa
TTCGGTACCCTCCATGACCCACG
AAATGAGGGCCCAGGGTATGCTT
```

This is a multiple sequence alignment with multiple aligned sections, so you would probably load this using the Bio.AlignIO.parse() function:

```
>>> from Bio import AlignIO
>>> align = AlignIO.parse("Mauve/simple_short.xmfa", "mauve")
>>> alignments = list(align)
>>> for aln in alignments:
...     print(aln)
...
Alignment with 3 rows and 240 columns
--------------------------------------------...--- a.fa
TTTAAACATCCCTCGGCCCGTCGCCCTTTTATAATAGCAGTACG...CTG b.fa/5416-5968
TTTAAACACCTTTTTGGATG--GCCCAGTTCGTTCAGTTGTG-G...CTT c.fa/9475-10076
Alignment with 2 rows and 46 columns
-----------------------GGGCGAACGTATAAACCATTCTG b.fa/5968-6015
TTCGGTACCCTCCATGACCCACGAAATGAGGGCCCAGGGTATGCTT c.fa/9428-9476
```

Additional information is extracted from the XMFA file and available through the annotation attribute of each record:

```
>>> for record in alignments[0]:
...     print(record.id, len(record))
...     print("  start: %d, end: %d, strand: %d" %(
...         record.annotations['start'], record.annotations['end'],
...         record.annotations['strand']))
...
a.fa 240
  start: 0, end: 0, strand: 1
b.fa/5416-5968 240
  start: 5416, end: 5968, strand: 1
c.fa/9475-10076 240
  start: 9475, end: 10076, strand: -1
```

class Bio.AlignIO.MauveIO.**MauveWriter**(**args*, ***kwargs*)

Bases: *SequentialAlignmentWriter*

Mauve/XMFA alignment writer.

__init__(**args*, ***kwargs*)

Initialize the class.

write_alignment(*alignment*)

Use this to write (another) single alignment to an open file.

Note that sequences and their annotation are recorded together (rather than having a block of annotation followed by a block of aligned sequences).

__annotations__ = {}

class Bio.AlignIO.MauveIO.**MauveIterator**(*handle*, *seq_count=None*)

Bases: *AlignmentIterator*

Mauve xmfa alignment iterator.

__next__()

Parse the next alignment from the handle.

__annotations__ = {'_ids': list[str]}

Bio.AlignIO.MsfIO module

Bio.AlignIO support for GCG MSF format.

The file format was produced by the GCG PileUp and and LocalPileUp tools, and later tools such as T-COFFEE and MUSCLE support it as an optional output format.

The original GCG tool would write gaps at ends of each sequence which could be missing data as tildes (~), whereas internal gaps were periods (.) instead. This parser replaces both with minus signs (-) for consistency with the rest of Bio.AlignIO.

You are expected to use this module via the Bio.AlignIO functions (or the Bio.SeqIO functions if you want to work directly with the gapped sequences).

class Bio.AlignIO.MsfIO.**MsfIterator**(*handle*, *seq_count=None*)

Bases: *AlignmentIterator*

GCG MSF alignment iterator.

__next__()

Parse the next alignment from the handle.

__annotations__ = {}

Bio.AlignIO.NexusIO module

Bio.AlignIO support for the "nexus" file format.

You are expected to use this module via the Bio.AlignIO functions (or the Bio.SeqIO functions if you want to work directly with the gapped sequences).

See also the Bio.Nexus module (which this code calls internally), as this offers more than just accessing the alignment or its sequences as SeqRecord objects.

Bio.AlignIO.NexusIO.**NexusIterator**(*handle: IO[str]*, *seq_count: int | None = None*) → Iterator[*MultipleSeqAlignment*]

Return SeqRecord objects from a Nexus file.

Thus uses the Bio.Nexus module to do the hard work.

You are expected to call this function via Bio.SeqIO or Bio.AlignIO (and not use it directly).

NOTE - We only expect ONE alignment matrix per Nexus file, meaning this iterator will only yield one MultipleSeqAlignment.

class Bio.AlignIO.NexusIO.**NexusWriter**(*handle*)

Bases: *AlignmentWriter*

Nexus alignment writer.

Note that Nexus files are only expected to hold ONE alignment matrix.

You are expected to call this class via the Bio.AlignIO.write() or Bio.SeqIO.write() functions.

write_file(*alignments*)

Use this to write an entire file containing the given alignments.

Arguments:

- alignments - A list or iterator returning MultipleSeqAlignment objects. This should hold ONE and only one alignment.

write_alignment(*alignment*, *interleave=None*)

Write an alignment to file.

Creates an empty Nexus object, adds the sequences and then gets Nexus to prepare the output. Default interleave behaviour: Interleave if columns > 1000 –> Override with interleave=[True/False]

__annotations__ = {}

Bio.AlignIO.PhylipIO module

AlignIO support for "phylip" format from Joe Felsenstein's PHYLIP tools.

You are expected to use this module via the Bio.AlignIO functions (or the Bio.SeqIO functions if you want to work directly with the gapped sequences).

Support for "relaxed phylip" format is also provided. Relaxed phylip differs from standard phylip format in the following ways:

- No whitespace is allowed in the sequence ID.
- No truncation is performed. Instead, sequence IDs are padded to the longest ID length, rather than 10 characters. A space separates the sequence identifier from the sequence.

Relaxed phylip is supported by RAxML and PHYML.

Note

In TREE_PUZZLE (Schmidt et al. 2003) and PHYML (Guindon and Gascuel 2003) a dot/period (".") in a sequence is interpreted as meaning the same character as in the first sequence. The PHYLIP documentation from 3.3 to 3.69 http://evolution.genetics.washington.edu/phylip/doc/sequence.html says:

"a period was also previously allowed but it is no longer allowed, because it sometimes is used in different senses in other programs"

Biopython 1.58 or later treats dots/periods in the sequence as invalid, both for reading and writing. Older versions did nothing special with a dot/period.

class Bio.AlignIO.PhylipIO.**PhylipWriter**(*handle*)

Bases: *SequentialAlignmentWriter*

Phylip alignment writer.

write_alignment(*alignment*, *id_width=_PHYLIP_ID_WIDTH*)

Use this to write (another) single alignment to an open file.

This code will write interlaced alignments (when the sequences are longer than 50 characters).

Note that record identifiers are strictly truncated to id_width, defaulting to the value required to comply with the PHYLIP standard.

For more information on the file format, please see: http://evolution.genetics.washington.edu/phylip/doc/sequence.html http://evolution.genetics.washington.edu/phylip/doc/main.html#inputfiles

__annotations__ = {}

class Bio.AlignIO.PhylipIO.**PhylipIterator**(*handle*, *seq_count=None*)

Bases: *AlignmentIterator*

Reads a Phylip alignment file returning a MultipleSeqAlignment iterator.

Record identifiers are limited to at most 10 characters.

It only copes with interlaced phylip files! Sequential files won't work where the sequences are split over multiple lines.

For more information on the file format, please see: http://evolution.genetics.washington.edu/phylip/doc/sequence.html http://evolution.genetics.washington.edu/phylip/doc/main.html#inputfiles

id_width = 10

__next__()

Parse the next alignment from the handle.

__annotations__ = {}

class Bio.AlignIO.PhylipIO.**RelaxedPhylipWriter**(*handle*)

Bases: *PhylipWriter*

Relaxed Phylip format writer.

write_alignment(*alignment*)

Write a relaxed phylip alignment.

__annotations__ = {}

class Bio.AlignIO.PhylipIO.**RelaxedPhylipIterator**(*handle*, *seq_count=None*)

Bases: *PhylipIterator*

Relaxed Phylip format Iterator.

__annotations__ = {}

class Bio.AlignIO.PhylipIO.**SequentialPhylipWriter**(*handle*)

Bases: *SequentialAlignmentWriter*

Sequential Phylip format Writer.

write_alignment(*alignment*, *id_width=_PHYLIP_ID_WIDTH*)

Write a Phylip alignment to the handle.

__annotations__ = {}

class Bio.AlignIO.PhylipIO.**SequentialPhylipIterator**(*handle*, *seq_count=None*)

Bases: *PhylipIterator*

Sequential Phylip format Iterator.

The sequential format carries the same restrictions as the normal interleaved one, with the difference being that the sequences are listed sequentially, each sequence written in its entirety before the start of the next. According to the PHYLIP documentation for input file formatting, newlines and spaces may optionally be entered at any point in the sequences.

__next__()

Parse the next alignment from the handle.

__annotations__ = {}

Bio.AlignIO.PhylipIO.**sanitize_name**(*name*, *width=None*)

Sanitise sequence identifier for output.

Removes the banned characters "[]()" and replaces the characters ":;" with "|". The name is truncated to "width" characters if specified.

Bio.AlignIO.StockholmIO module

Bio.AlignIO support for "stockholm" format (used in the PFAM database).

You are expected to use this module via the Bio.AlignIO functions (or the Bio.SeqIO functions if you want to work directly with the gapped sequences).

For example, consider a Stockholm alignment file containing the following:

```
# STOCKHOLM 1.0
#=GC SS_cons        .................<<<<<<<<...<<<<<<<........>>>>>>>..
AP001509.1          UUAAUCGAGCUCAACACUCUUCGUAUAUCCUC-UCAAUAUGG-GAUGAGGGU
#=GR AP001509.1 SS -----------------<<<<<<<<---..<<-<<-------->>->>..--
AE007476.1          AAAAUUGAAUAUCGUUUUACUUGUUUAU-GUCGUGAAU-UGG-CACGA-CGU
#=GR AE007476.1 SS -----------------<<<<<<<<-----<<.<<-------->>.>>----

#=GC SS_cons        ......<<<<<<<.......>>>>>>>..>>>>>>>>...............
AP001509.1          CUCUAC-AGGUA-CCGUAAA-UACCUAGCUACGAAAAGAAUGCAGUUAAUGU
#=GR AP001509.1 SS -------<<<<<---------->>>>>---<<<<<<<<---------------
AE007476.1          UUCUACAAGGUG-CCGG-AA-CACCUAACAAUAAGUAAGUCAGCAGUGAGAU
#=GR AE007476.1 SS ------.<<<<<---------->>>>>.--<<<<<<<<---------------
//
```

This is a single multiple sequence alignment, so you would probably load this using the Bio.AlignIO.read() function:

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("Stockholm/simple.sth", "stockholm")
>>> print(align)
Alignment with 2 rows and 104 columns
UUAAUCGAGCUCAACACUCUUCGUAUAUCCUC-UCAAUAUGG-G...UGU AP001509.1
AAAAUUGAAUAUCGUUUUACUUGUUUAU-GUCGUGAAU-UGG-C...GAU AE007476.1
>>> for record in align:
...     print("%s %i" % (record.id, len(record)))
```

(continues on next page)

(continued from previous page)

```
AP001509.1 104
AE007476.1 104
```

In addition to the sequences themselves, this example alignment also includes some GR lines for the secondary structure of the sequences. These are strings, with one character for each letter in the associated sequence:

```
>>> for record in align:
...     print(record.id)
...     print(record.seq)
...     print(record.letter_annotations['secondary_structure'])
AP001509.1
UUAAUCGAGCUCAACACUCUUCGUAUAUCCUC-UCAAUAUGG-GAUGAGGGUCUCUAC-AGGUA-CCGUAAA-
↪UACCUAGCUACGAAAAGAAUGCAGUUAAUGU
-----------------<<<<<<<<<---..<<-<<-------->>->>..---------<<<<<--------->>>>>--->>>>>>>>
↪---------------
AE007476.1
AAAAUUGAAUAUCGUUUUACUUGUUUAU-GUCGUGAAU-UGG-CACGA-CGUUUCUACAAGGUG-CCGG-AA-
↪CACCUAACAAUAAGUAAGUCAGCAGUGAGAU
-----------------<<<<<<<<<-----<<.<<-------->>.>>----------.<<<<<--------->>>>>.-->>>>>>>>>
↪---------------
```

Any general annotation for each row is recorded in the SeqRecord's annotations dictionary. Any per-column annotation for the entire alignment in in the alignment's column annotations dictionary, such as the secondary structure consensus in this example:

```
>>> sorted(align.column_annotations.keys())
['secondary_structure']
>>> align.column_annotations["secondary_structure"]
'.................<<<<<<<<...<<<<<<<........>>>>>>>........<<<<<<<.......>>>>>>>..>>>>>>>
↪>.............'
```

You can output this alignment in many different file formats using Bio.AlignIO.write(), or the MultipleSeqAlignment object's format method:

```
>>> print(format(align, "fasta"))
>AP001509.1
UUAAUCGAGCUCAACACUCUUCGUAUAUCCUC-UCAAUAUGG-GAUGAGGGUCUCUAC-A
GGUA-CCGUAAA-UACCUAGCUACGAAAAGAAUGCAGUUAAUGU
>AE007476.1
AAAAUUGAAUAUCGUUUUACUUGUUUAU-GUCGUGAAU-UGG-CACGA-CGUUUCUACAA
GGUG-CCGG-AA-CACCUAACAAUAAGUAAGUCAGCAGUGAGAU
```

Most output formats won't be able to hold the annotation possible in a Stockholm file:

```
>>> print(format(align, "stockholm"))
# STOCKHOLM 1.0
#=GF SQ 2
AP001509.1 UUAAUCGAGCUCAACACUCUUCGUAUAUCCUC-UCAAUAUGG-GAUGAGGGUCUCUAC-AGGUA-CCGUAAA-
↪UACCUAGCUACGAAAAGAAUGCAGUUAAUGU
#=GS AP001509.1 AC AP001509.1
#=GS AP001509.1 DE AP001509.1
#=GR AP001509.1 SS -----------------<<<<<<<<<---..<<-<<-------->>->>..---------<<<<<------
↪--->>>>>---->>>>>>>>>---------------
```

(continues on next page)

(continued from previous page)

```
AE007476.1 AAAAUUGAAUAUCGUUUUACUUGUUUAU-GUCGUGAAU-UGG-CACGA-CGUUUCUACAAGGUG-CCGG-AA-
↪CACCUAACAAUAAGUAAGUCAGCAGUGAGAU
#=GS AE007476.1 AC AE007476.1
#=GS AE007476.1 DE AE007476.1
#=GR AE007476.1 SS ------------------<<<<<<<<-----<<.<<-------->>.>>----------.<<<<<------
↪---->>>>>.-->>>>>>>>>---------------
#=GC SS_cons .................<<<<<<<<...<<<<<<<........>>>>>>>........<<<<<<<.......>>>>
↪>>>..>>>>>>>>..............
//
```

Note that when writing Stockholm files, AlignIO does not break long sequences up and interleave them (as in the input file shown above). The standard allows this simpler layout, and it is more likely to be understood by other tools.

Finally, as an aside, it can sometimes be useful to use Bio.SeqIO.parse() to iterate over the alignment rows as SeqRecord objects - rather than working with Alignnment objects.

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("Stockholm/simple.sth", "stockholm"):
...     print(record.id)
...     print(record.seq)
...     print(record.letter_annotations['secondary_structure'])
AP001509.1
UUAAUCGAGCUCAACACUCUUCGUAUAUCCUC-UCAAUAUGG-GAUGAGGGUCUCUAC-AGGUA-CCGUAAA-
↪UACCUAGCUACGAAAAGAAUGCAGUUAAUGU
------------------<<<<<<<<<---..<<-<<-------->>->>..---------<<<<<---------->>>>>--->>>>>>>>
↪---------------
AE007476.1
AAAAUUGAAUAUCGUUUUACUUGUUUAU-GUCGUGAAU-UGG-CACGA-CGUUUCUACAAGGUG-CCGG-AA-
↪CACCUAACAAUAAGUAAGUCAGCAGUGAGAU
------------------<<<<<<<<-----<<.<<-------->>.>>----------.<<<<<---------->>>>>.-->>>>>>>>>
↪---------------
```

Remember that if you slice a SeqRecord, the per-letter-annotations like the secondary structure string here, are also sliced:

```
>>> sub_record = record[10:20]
>>> print(sub_record.seq)
AUCGUUUUAC
>>> print(sub_record.letter_annotations['secondary_structure'])
-------<<<
```

Likewise with the alignment object, as long as you are not dropping any rows, slicing specific columns of an alignment will slice any per-column-annotations:

```
>>> align.column_annotations["secondary_structure"]
'.................<<<<<<<<...<<<<<<<........>>>>>>>........<<<<<<<.......>>>>>>>..>>>>>>>
↪>..............'
>>> part_align = align[:,10:20]
>>> part_align.column_annotations["secondary_structure"]
'.......<<<'
```

You can also see this in the Stockholm output of this partial-alignment:

```
>>> print(format(part_align, "stockholm"))
# STOCKHOLM 1.0
#=GF SQ 2
AP001509.1 UCAACACUCU
#=GS AP001509.1 AC AP001509.1
#=GS AP001509.1 DE AP001509.1
#=GR AP001509.1 SS -------<<<
AE007476.1 AUCGUUUUAC
#=GS AE007476.1 AC AE007476.1
#=GS AE007476.1 DE AE007476.1
#=GR AE007476.1 SS -------<<<
#=GC SS_cons .......<<<
//
```

class Bio.AlignIO.StockholmIO.**StockholmWriter**(*handle*)

Bases: *SequentialAlignmentWriter*

Stockholm/PFAM alignment writer.

pfam_gr_mapping = {'active_site': 'AS', 'intron': 'IN', 'ligand_binding': 'LI', 'posterior_probability': 'PP', 'secondary_structure': 'SS', 'surface_accessibility': 'SA', 'transmembrane': 'TM'}

pfam_gc_mapping = {'model_mask': 'MM', 'reference_annotation': 'RF'}

pfam_gs_mapping = {'look': 'LO', 'organism': 'OS', 'organism_classification': 'OC'}

write_alignment(*alignment*)

Use this to write (another) single alignment to an open file.

Note that sequences and their annotation are recorded together (rather than having a block of annotation followed by a block of aligned sequences).

__annotations__ = {}

class Bio.AlignIO.StockholmIO.**StockholmIterator**(*handle*, *seq_count=None*)

Bases: *AlignmentIterator*

Loads a Stockholm file from PFAM into MultipleSeqAlignment objects.

The file may contain multiple concatenated alignments, which are loaded and returned incrementally.

This parser will detect if the Stockholm file follows the PFAM conventions for sequence specific meta-data (lines starting #=GS and #=GR) and populates the SeqRecord fields accordingly.

Any annotation which does not follow the PFAM conventions is currently ignored.

If an accession is provided for an entry in the meta data, IT WILL NOT be used as the record.id (it will be recorded in the record's annotations). This is because some files have (sub) sequences from different parts of the same accession (differentiated by different start-end positions).

Wrap-around alignments are not supported - each sequences must be on a single line. However, interlaced sequences should work.

For more information on the file format, please see: http://sonnhammer.sbc.su.se/Stockholm.html https://en.wikipedia.org/wiki/Stockholm_format http://bioperl.org/formats/alignment_formats/Stockholm_multiple_alignment_format.html

For consistency with BioPerl and EMBOSS we call this the "stockholm" format.

```
pfam_gr_mapping = {'AS': 'active_site', 'IN': 'intron', 'LI': 'ligand_binding',
'PP': 'posterior_probability', 'SA': 'surface_accessibility', 'SS':
'secondary_structure', 'TM': 'transmembrane'}

pfam_gc_mapping = {'MM': 'model_mask', 'RF': 'reference_annotation'}

pfam_gs_mapping = {'LO': 'look', 'OC': 'organism_classification', 'OS': 'organism'}
```

__next__()

Parse the next alignment from the handle.

__annotations__ = {}

Module contents

Multiple sequence alignment input/output as alignment objects.

The Bio.AlignIO interface is deliberately very similar to Bio.SeqIO, and in fact the two are connected internally. Both modules use the same set of file format names (lower case strings). From the user's perspective, you can read in a PHYLIP file containing one or more alignments using Bio.AlignIO, or you can read in the sequences within these alignments using Bio.SeqIO.

Bio.AlignIO is also documented at http://biopython.org/wiki/AlignIO and by a whole chapter in our tutorial:

- HTML Tutorial
- PDF Tutorial

Input

For the typical special case when your file or handle contains one and only one alignment, use the function Bio.AlignIO.read(). This takes an input file handle (or in recent versions of Biopython a filename as a string), format string and optional number of sequences per alignment. It will return a single MultipleSeqAlignment object (or raise an exception if there isn't just one alignment):

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("Phylip/interlaced.phy", "phylip")
>>> print(align)
Alignment with 3 rows and 384 columns
-----MKVILLFVLAVFTVFVSS---------------RGIPPE...I-- CYS1_DICDI
MAHARVLLLALAVLATAAVAVASSSSFADSNPIRPVTDRAASTL...VAA ALEU_HORVU
------MWATLPLLCAGAWLLGV--------PVCGAAELSVNSL...PLV CATH_HUMAN
```

For the general case, when the handle could contain any number of alignments, use the function Bio.AlignIO.parse(...) which takes the same arguments, but returns an iterator giving MultipleSeqAlignment objects (typically used in a for loop). If you want random access to the alignments by number, turn this into a list:

```
>>> from Bio import AlignIO
>>> alignments = list(AlignIO.parse("Emboss/needle.txt", "emboss"))
>>> print(alignments[2])
Alignment with 2 rows and 120 columns
-KILIVDDQYGIRILLNEVFNKEGYQTFQAANGLQALDIVTKER...--- ref_rec
LHIVVVDDDPGTCVYIESVFAELGHTCKSFVRPEAAEEYILTHP...HKE gi|94967506|receiver
```

Most alignment file formats can be concatenated so as to hold as many different multiple sequence alignments as possible. One common example is the output of the tool seqboot in the PHLYIP suite. Sometimes there can be a file header and footer, as seen in the EMBOSS alignment output.

Output

Use the function Bio.AlignIO.write(...), which takes a complete set of Alignment objects (either as a list, or an iterator), an output file handle (or filename in recent versions of Biopython) and of course the file format:

```
from Bio import AlignIO
alignments = ...
count = SeqIO.write(alignments, "example.faa", "fasta")
```

If using a handle make sure to close it to flush the data to the disk:

```
from Bio import AlignIO
alignments = ...
with open("example.faa", "w") as handle:
    count = SeqIO.write(alignments, handle, "fasta")
```

In general, you are expected to call this function once (with all your alignments) and then close the file handle. However, for file formats like PHYLIP where multiple alignments are stored sequentially (with no file header and footer), then multiple calls to the write function should work as expected when using handles.

If you are using a filename, the repeated calls to the write functions will overwrite the existing file each time.

Conversion

The Bio.AlignIO.convert(...) function allows an easy interface for simple alignment file format conversions. Additionally, it may use file format specific optimisations so this should be the fastest way too.

In general however, you can combine the Bio.AlignIO.parse(...) function with the Bio.AlignIO.write(...) function for sequence file conversion. Using generator expressions provides a memory efficient way to perform filtering or other extra operations as part of the process.

File Formats

When specifying the file format, use lowercase strings. The same format names are also used in Bio.SeqIO and include the following:

- clustal - Output from Clustal W or X.
- emboss - EMBOSS tools' "pairs" and "simple" alignment formats.
- fasta - The generic sequence file format where each record starts with an identifier line starting with a ">" character, followed by lines of sequence.
- fasta-m10 - For the pairwise alignments output by Bill Pearson's FASTA tools when used with the -m 10 command line option for machine readable output.
- ig - The IntelliGenetics file format, apparently the same as the MASE alignment format.
- msf - The GCG MSF alignment format, originally from PileUp tool.
- nexus - Output from NEXUS, see also the module Bio.Nexus which can also read any phylogenetic trees in these files.

- phylip - Interlaced PHYLIP, as used by the PHYLIP tools.
- phylip-sequential - Sequential PHYLIP.
- phylip-relaxed - PHYLIP like format allowing longer names.
- stockholm - A richly annotated alignment file format used by PFAM.
- mauve - Output from progressiveMauve/Mauve

Note that while Bio.AlignIO can read all the above file formats, it cannot write to all of them.

You can also use any file format supported by Bio.SeqIO, such as "fasta" or "ig" (which are listed above), PROVIDED the sequences in your file are all the same length.

Bio.AlignIO.**write**(*alignments*, *handle*, *format*)

Write complete set of alignments to a file.

Arguments:

- alignments - A list (or iterator) of MultipleSeqAlignment objects, or a single alignment object.
- handle - File handle object to write to, or filename as string (note older versions of Biopython only took a handle).
- format - lower case string describing the file format to write.

You should close the handle after calling this function.

Returns the number of alignments written (as an integer).

Bio.AlignIO.**parse**(*handle*, *format*, *seq_count=None*)

Iterate over an alignment file as MultipleSeqAlignment objects.

Arguments:

- handle - handle to the file, or the filename as a string (note older versions of Biopython only took a handle).
- format - string describing the file format.
- seq_count - Optional integer, number of sequences expected in each alignment. Recommended for fasta format files.

If you have the file name in a string 'filename', use:

```
>>> from Bio import AlignIO
>>> filename = "Emboss/needle.txt"
>>> format = "emboss"
>>> for alignment in AlignIO.parse(filename, format):
...     print("Alignment of length %i" % alignment.get_alignment_length())
Alignment of length 124
Alignment of length 119
Alignment of length 120
Alignment of length 118
Alignment of length 125
```

If you have a string 'data' containing the file contents, use:

```
from Bio import AlignIO
from io import StringIO
my_iterator = AlignIO.parse(StringIO(data), format)
```

Use the Bio.AlignIO.read() function when you expect a single record only.

Bio.AlignIO.**read**(*handle*, *format*, *seq_count=None*)

Turn an alignment file into a single MultipleSeqAlignment object.

Arguments:

- handle - handle to the file, or the filename as a string (note older versions of Biopython only took a handle).
- format - string describing the file format.
- seq_count - Optional integer, number of sequences expected in each alignment. Recommended for fasta format files.

If the handle contains no alignments, or more than one alignment, an exception is raised. For example, using a PFAM/Stockholm file containing one alignment:

```
>>> from Bio import AlignIO
>>> filename = "Clustalw/protein.aln"
>>> format = "clustal"
>>> alignment = AlignIO.read(filename, format)
>>> print("Alignment of length %i" % alignment.get_alignment_length())
Alignment of length 411
```

If however you want the first alignment from a file containing multiple alignments this function would raise an exception.

```
>>> from Bio import AlignIO
>>> filename = "Emboss/needle.txt"
>>> format = "emboss"
>>> alignment = AlignIO.read(filename, format)
Traceback (most recent call last):
    ...
ValueError: More than one record found in handle
```

Instead use:

```
>>> from Bio import AlignIO
>>> filename = "Emboss/needle.txt"
>>> format = "emboss"
>>> alignment = next(AlignIO.parse(filename, format))
>>> print("First alignment has length %i" % alignment.get_alignment_length())
First alignment has length 124
```

You must use the Bio.AlignIO.parse() function if you want to read multiple records from the handle.

Bio.AlignIO.**convert**(*in_file*, *in_format*, *out_file*, *out_format*, *molecule_type=None*)

Convert between two alignment files, returns number of alignments.

Arguments:

- in_file - an input handle or filename
- in_format - input file format, lower case string
- output - an output handle or filename
- out_file - output file format, lower case string
- molecule_type - optional molecule type to apply, string containing “DNA”, “RNA” or “protein”.

NOTE - If you provide an output filename, it will be opened which will overwrite any existing file without warning. This may happen if even the conversion is aborted (e.g. an invalid out_format name is given).

Some output formats require the molecule type be specified where this cannot be determined by the parser. For example, converting to FASTA, Clustal, or PHYLIP format to NEXUS:

```
>>> from io import StringIO
>>> from Bio import AlignIO
>>> handle = StringIO()
>>> AlignIO.convert("Phylip/horses.phy", "phylip", handle, "nexus", "DNA")
1
>>> print(handle.getvalue())
#NEXUS
begin data;
dimensions ntax=10 nchar=40;
format datatype=dna missing=? gap=-;
matrix
Mesohippus   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hypohippus   AAACCCCCCCAAAAAAAAACAAAAAAAAAAAAAAAAAAAA
Archaeohip   CAAAAAAAAAAAAAAAACACAAAAAAAAAAAAAAAAAAAA
Parahippus   CAAACAACAACAAAAAAAACAAAAAAAAAAAAAAAAAAAA
Merychippu   CCAACCACCACCCCACACCCAAAAAAAAAAAAAAAAAAAA
'M. secundu' CCAACCACCACCCACACCCCAAAAAAAAAAAAAAAAAAAA
Nannipus     CCAACCACAACCCCACACCCAAAAAAAAAAAAAAAAAAAA
Neohippari   CCAACCCCCCCCCCACACCCAAAAAAAAAAAAAAAAAAAA
Calippus     CCAACCACAACCCACACCCCAAAAAAAAAAAAAAAAAAAA
Pliohippus   CCCACCCCCCCCCCACACCCCAAAAAAAAAAAAAAAAAAAA
;
end;
```

28.1.4 Bio.Application package

Module contents

General mechanisms to access applications in Biopython (DEPRECATED).

This module is not intended for direct use. It provides the basic objects which are subclassed by our command line wrappers, such as:

- Bio.Align.Applications
- Bio.Blast.Applications
- Bio.Emboss.Applications
- Bio.Sequencing.Applications

These modules provide wrapper classes for command line tools to help you construct command line strings by setting the values of each parameter. The finished command line strings are then normally invoked via the built-in Python module subprocess.

Due to the on going maintenance burden or keeping command line application wrappers up to date, we have decided to deprecate and eventually remove them. We instead now recommend building your command line and invoking it directly with the subprocess module.

exception Bio.Application.**ApplicationError**(*returncode*, *cmd*, *stdout=''*, *stderr=''*)

Bases: CalledProcessError

Raised when an application returns a non-zero exit status (OBSOLETE).

The exit status will be stored in the returncode attribute, similarly the command line string used in the cmd attribute, and (if captured) stdout and stderr as strings.

This exception is a subclass of subprocess.CalledProcessError.

```
>>> err = ApplicationError(-11, "helloworld", "", "Some error text")
>>> err.returncode, err.cmd, err.stdout, err.stderr
(-11, 'helloworld', '', 'Some error text')
>>> print(err)
Non-zero return code -11 from 'helloworld', message 'Some error text'
```

__init__(*returncode*, *cmd*, *stdout=''*, *stderr=''*)

Initialize the class.

__str__()

Format the error as a string.

__repr__()

Represent the error as a string.

class Bio.Application.**AbstractCommandline**(*cmd*, ***kwargs*)

Bases: object

Generic interface for constructing command line strings (OBSOLETE).

This class shouldn't be called directly; it should be subclassed to provide an implementation for a specific application.

For a usage example we'll show one of the EMBOSS wrappers. You can set options when creating the wrapper object using keyword arguments - or later using their corresponding properties:

```
>>> from Bio.Emboss.Applications import WaterCommandline
>>> cline = WaterCommandline(gapopen=10, gapextend=0.5)
>>> cline
WaterCommandline(cmd='water', gapopen=10, gapextend=0.5)
```

You can instead manipulate the parameters via their properties, e.g.

```
>>> cline.gapopen
10
>>> cline.gapopen = 20
>>> cline
WaterCommandline(cmd='water', gapopen=20, gapextend=0.5)
```

You can clear a parameter you have already added by 'deleting' the corresponding property:

```
>>> del cline.gapopen
>>> cline.gapopen
>>> cline
WaterCommandline(cmd='water', gapextend=0.5)
```

Once you have set the parameters you need, you can turn the object into a string (e.g. to log the command):

```
>>> str(cline)
Traceback (most recent call last):
...
ValueError: You must either set outfile (output filename), or enable filter or␣
↪stdout (output to stdout).
```

In this case the wrapper knows certain arguments are required to construct a valid command line for the tool. For a complete example,

```
>>> from Bio.Emboss.Applications import WaterCommandline
>>> water_cmd = WaterCommandline(gapopen=10, gapextend=0.5)
>>> water_cmd.asequence = "asis:ACCCGGGCGCGGT"
>>> water_cmd.bsequence = "asis:ACCCGAGCGCGGT"
>>> water_cmd.outfile = "temp_water.txt"
>>> print(water_cmd)
water -outfile=temp_water.txt -asequence=asis:ACCCGGGCGCGGT -
↪bsequence=asis:ACCCGAGCGCGGT -gapopen=10 -gapextend=0.5
>>> water_cmd
WaterCommandline(cmd='water', outfile='temp_water.txt', asequence=
↪'asis:ACCCGGGCGCGGT', bsequence='asis:ACCCGAGCGCGGT', gapopen=10, gapextend=0.5)
```

You would typically run the command line via a standard Python operating system call using the subprocess module for full control. For the simple case where you just want to run the command and get the output:

stdout, stderr = water_cmd()

Note that by default we assume the underlying tool is installed on the system $PATH environment variable. This is normal under Linux/Unix, but may need to be done manually under Windows. Alternatively, you can specify the full path to the binary as the first argument (cmd):

```
>>> from Bio.Emboss.Applications import WaterCommandline
>>> water_cmd = WaterCommandline(r"C:\Program Files\EMBOSS\water.exe",
...                              gapopen=10, gapextend=0.5,
...                              asequence="asis:ACCCGGGCGCGGT",
...                              bsequence="asis:ACCCGAGCGCGGT",
...                              outfile="temp_water.txt")
>>> print(water_cmd)
"C:\Program Files\EMBOSS\water.exe" -outfile=temp_water.txt -
↪asequence=asis:ACCCGGGCGCGGT -bsequence=asis:ACCCGAGCGCGGT -gapopen=10 -
↪gapextend=0.5
```

Notice that since the path name includes a space it has automatically been quoted.

parameters = None

__init__(*cmd*, ***kwargs*)

Create a new instance of a command line wrapper object.

__str__()

Make the commandline string with the currently set options.

e.g.

```
>>> from Bio.Emboss.Applications import WaterCommandline
>>> cline = WaterCommandline(gapopen=10, gapextend=0.5)
>>> cline.asequence = "asis:ACCCGGGCGCGGT"
```

(continues on next page)

(continued from previous page)

```
>>> cline.bsequence = "asis:ACCCGAGCGCGGT"
>>> cline.outfile = "temp_water.txt"
>>> print(cline)
water -outfile=temp_water.txt -asequence=asis:ACCCGGGCGCGGT -
↪bsequence=asis:ACCCGAGCGCGGT -gapopen=10 -gapextend=0.5
>>> str(cline)
'water -outfile=temp_water.txt -asequence=asis:ACCCGGGCGCGGT -
↪bsequence=asis:ACCCGAGCGCGGT -gapopen=10 -gapextend=0.5'
```

__repr__()

Return a representation of the command line object for debugging.

e.g.

```
>>> from Bio.Emboss.Applications import WaterCommandline
>>> cline = WaterCommandline(gapopen=10, gapextend=0.5)
>>> cline.asequence = "asis:ACCCGGGCGCGGT"
>>> cline.bsequence = "asis:ACCCGAGCGCGGT"
>>> cline.outfile = "temp_water.txt"
>>> print(cline)
water -outfile=temp_water.txt -asequence=asis:ACCCGGGCGCGGT -
↪bsequence=asis:ACCCGAGCGCGGT -gapopen=10 -gapextend=0.5
>>> cline
WaterCommandline(cmd='water', outfile='temp_water.txt', asequence=
↪'asis:ACCCGGGCGCGGT', bsequence='asis:ACCCGAGCGCGGT', gapopen=10, gapextend=0.
↪5)
```

set_parameter(*name*, *value=None*)

Set a commandline option for a program (OBSOLETE).

Every parameter is available via a property and as a named keyword when creating the instance. Using either of these is preferred to this legacy set_parameter method which is now OBSOLETE, and likely to be DEPRECATED and later REMOVED in future releases.

__setattr__(*name*, *value*)

Set attribute name to value (PRIVATE).

This code implements a workaround for a user interface issue. Without this __setattr__ attribute-based assignment of parameters will silently accept invalid parameters, leading to known instances of the user assuming that parameters for the application are set, when they are not.

```
>>> from Bio.Emboss.Applications import WaterCommandline
>>> cline = WaterCommandline(gapopen=10, gapextend=0.5, stdout=True)
>>> cline.asequence = "a.fasta"
>>> cline.bsequence = "b.fasta"
>>> cline.csequence = "c.fasta"
Traceback (most recent call last):
...
ValueError: Option name csequence was not found.
>>> print(cline)
water -stdout -asequence=a.fasta -bsequence=b.fasta -gapopen=10 -gapextend=0.5
```

This workaround uses a whitelist of object attributes, and sets the object attribute list as normal, for these. Other attributes are assumed to be parameters, and passed to the self.set_parameter method for validation and assignment.

__call__(*stdin=None*, *stdout=True*, *stderr=True*, *cwd=None*, *env=None*)

Execute command, wait for it to finish, return (stdout, stderr).

Runs the command line tool and waits for it to finish. If it returns a non-zero error level, an exception is raised. Otherwise two strings are returned containing stdout and stderr.

The optional stdin argument should be a string of data which will be passed to the tool as standard input.

The optional stdout and stderr argument may be filenames (string), but otherwise are treated as a booleans, and control if the output should be captured as strings (True, default), or ignored by sending it to /dev/null to avoid wasting memory (False). If sent to a file or ignored, then empty string(s) are returned.

The optional cwd argument is a string giving the working directory to run the command from. See Python's subprocess module documentation for more details.

The optional env argument is a dictionary setting the environment variables to be used in the new process. By default the current process' environment variables are used. See Python's subprocess module documentation for more details.

Default example usage:

```
from Bio.Emboss.Applications import WaterCommandline
water_cmd = WaterCommandline(gapopen=10, gapextend=0.5,
                             stdout=True, auto=True,
                             asequence="a.fasta", bsequence="b.fasta")
print("About to run: %s" % water_cmd)
std_output, err_output = water_cmd()
```

This functionality is similar to subprocess.check_output(). In general if you require more control over running the command, use subprocess directly.

When the program called returns a non-zero error level, a custom ApplicationError exception is raised. This includes any stdout and stderr strings captured as attributes of the exception object, since they may be useful for diagnosing what went wrong.

__annotations__ = {}

28.1.5 Bio.Blast package

Submodules

Bio.Blast.Applications module

Definitions for interacting with BLAST related applications (OBSOLETE).

Wrappers for the new NCBI BLAST+ tools (written in C++):

- NcbiblastpCommandline - Protein-Protein BLAST
- NcbiblastnCommandline - Nucleotide-Nucleotide BLAST
- NcbiblastxCommandline - Translated Query-Protein Subject BLAST
- NcbitblastnCommandline - Protein Query-Translated Subject BLAST
- NcbitblastxCommandline - Translated Query-Protein Subject BLAST
- NcbipsiblastCommandline - Position-Specific Initiated BLAST
- NcbirpsblastCommandline - Reverse Position Specific BLAST

- NcbirpstblastnCommandline - Translated Reverse Position Specific BLAST
- NcbideltablastCommandline - Protein-Protein domain enhanced lookup time accelerated blast
- NcbiblastformatterCommandline - Convert ASN.1 to other BLAST output formats
- NcbimakeblastdbCommandline - Application to create BLAST databases

For further details, see:

Camacho et al. BLAST+: architecture and applications BMC Bioinformatics 2009, 10:421 https://doi.org/10.1186/1471-2105-10-421

We have decided to remove this module in future, and instead recommend building your command and invoking it via the subprocess module directly.

class Bio.Blast.Applications.**NcbiblastpCommandline**(*cmd='blastp'*, ***kwargs*)

Bases: _NcbiblastMain2SeqCommandline

Create a commandline for the NCBI BLAST+ program blastp (for proteins).

With the release of BLAST+ (BLAST rewritten in C++ instead of C), the NCBI replaced the old blastall tool with separate tools for each of the searches. This wrapper therefore replaces BlastallCommandline with option -p blastp.

```
>>> from Bio.Blast.Applications import NcbiblastpCommandline
>>> cline = NcbiblastpCommandline(query="rosemary.pro", db="nr",
...                               evalue=0.001, remote=True, ungapped=True)
>>> cline
NcbiblastpCommandline(cmd='blastp', query='rosemary.pro', db='nr', evalue=0.001,␣
↪remote=True, ungapped=True)
>>> print(cline)
blastp -query rosemary.pro -db nr -evalue 0.001 -remote -ungapped
```

You would typically run the command line with cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='blastp'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property best_hit_overhang

Best Hit algorithm overhang value (float, recommended value: 0.1)

Float between 0.0 and 0.5 inclusive. Incompatible with: culling_limit.

This controls the addition of the -best_hit_overhang parameter and its associated value. Set this property to the argument value required.

property best_hit_score_edge

Best Hit algorithm score edge value (float).

Float between 0.0 and 0.5 inclusive. Recommended value: 0.1

Incompatible with: culling_limit.

This controls the addition of the -best_hit_score_edge parameter and its associated value. Set this property to the argument value required.

property comp_based_stats

Use composition-based statistics (string, default 2, i.e. True).

0, F or f: no composition-based statistics

2, T or t, D or d : Composition-based score adjustment as in Bioinformatics 21:902-911, 2005, conditioned on sequence properties

Note that tblastn also supports values of 1 and 3.

This controls the addition of the -comp_based_stats parameter and its associated value. Set this property to the argument value required.

property culling_limit

Hit culling limit (integer).

If the query range of a hit is enveloped by that of at least this many higher-scoring hits, delete the hit.

Incompatible with: best_hit_overhang, best_hit_score_edge.

This controls the addition of the -culling_limit parameter and its associated value. Set this property to the argument value required.

property db

The database to BLAST against.

This controls the addition of the -db parameter and its associated value. Set this property to the argument value required.

property db_hard_mask

Filtering algorithm for hard masking (integer).

Filtering algorithm ID to apply to BLAST database as hard masking. Incompatible with: db_soft_mask, subject, subject_loc

This controls the addition of the -db_hard_mask parameter and its associated value. Set this property to the argument value required.

property db_soft_mask

Filtering algorithm for soft masking (integer).

Filtering algorithm ID to apply to BLAST database as soft masking. Incompatible with: db_hard_mask, subject, subject_loc

This controls the addition of the -db_soft_mask parameter and its associated value. Set this property to the argument value required.

property dbsize

Effective length of the database (integer).

This controls the addition of the -dbsize parameter and its associated value. Set this property to the argument value required.

property entrez_query

Restrict search with the given Entrez query (requires remote).

This controls the addition of the -entrez_query parameter and its associated value. Set this property to the argument value required.

property evalue

Expectation value cutoff.

This controls the addition of the -evalue parameter and its associated value. Set this property to the argument value required.

property export_search_strategy

File name to record the search strategy used.

Incompatible with: import_search_strategy

This controls the addition of the -export_search_strategy parameter and its associated value. Set this property to the argument value required.

property gapextend

Cost to extend a gap (integer).

This controls the addition of the -gapextend parameter and its associated value. Set this property to the argument value required.

property gapopen

Cost to open a gap (integer).

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property gilist

Restrict search of database to list of GI's.

Incompatible with: negative_gilist, seqidlist, negative_seqidlist, remote, subject, subject_loc

This controls the addition of the -gilist parameter and its associated value. Set this property to the argument value required.

property h

Print USAGE and DESCRIPTION; ignore other arguments.

This property controls the addition of the -h switch, treat this property as a boolean.

property help

Print USAGE, DESCRIPTION and ARGUMENTS description; ignore other arguments.

This property controls the addition of the -help switch, treat this property as a boolean.

property html

Produce HTML output? See also the outfmt option.

This property controls the addition of the -html switch, treat this property as a boolean.

property import_search_strategy

Search strategy to use.

Incompatible with: export_search_strategy

This controls the addition of the -import_search_strategy parameter and its associated value. Set this property to the argument value required.

property lcase_masking

Use lower case filtering in query and subject sequence(s)?

This property controls the addition of the -lcase_masking switch, treat this property as a boolean.

property line_length

Line length for formatting alignments (integer, at least 1, default 60).

Not applicable for outfmt > 4. Added in BLAST+ 2.2.30.

This controls the addition of the -line_length parameter and its associated value. Set this property to the argument value required.

property matrix

Scoring matrix name (default BLOSUM62).

This controls the addition of the -matrix parameter and its associated value. Set this property to the argument value required.

property max_hsps

Set max number of HSPs saved per subject sequence

Ddefault 0 means no limit.

This controls the addition of the -max_hsps parameter and its associated value. Set this property to the argument value required.

property max_hsps_per_subject

Override max number of HSPs per subject saved for ungapped searches (integer).

This controls the addition of the -max_hsps_per_subject parameter and its associated value. Set this property to the argument value required.

property max_target_seqs

Maximum number of aligned sequences to keep (integer, at least one).

This controls the addition of the -max_target_seqs parameter and its associated value. Set this property to the argument value required.

property negative_gilist

Restrict search of database to everything except the listed GIs.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_gilist parameter and its associated value. Set this property to the argument value required.

property negative_seqidlist

Restrict search of database to everything except listed SeqID's.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_seqidlist parameter and its associated value. Set this property to the argument value required.

property num_alignments

Number of database sequences to show num_alignments for.

Integer argument (at least zero). Default is 200. See also num_alignments.

This controls the addition of the -num_alignments parameter and its associated value. Set this property to the argument value required.

property num_descriptions

Number of database sequences to show one-line descriptions for.

Integer argument (at least zero). Default is 500. See also num_alignments.

This controls the addition of the -num_descriptions parameter and its associated value. Set this property to the argument value required.

property num_threads

Number of threads to use in the BLAST search.

Integer, at least one. Default is one. Incompatible with: remote

This controls the addition of the -num_threads parameter and its associated value. Set this property to the argument value required.

property out

Output file for alignment.

This controls the addition of the -out parameter and its associated value. Set this property to the argument value required.

property outfmt

Alignment view. Typically an integer 0-14 but for some formats can be named columns like '6 qseqid sseqid'. Use 5 for XML output (differs from classic BLAST which used 7 for XML).

This controls the addition of the -outfmt parameter and its associated value. Set this property to the argument value required.

property parse_deflines

Should the query and subject defline(s) be parsed?

This property controls the addition of the -parse_deflines switch, treat this property as a boolean.

property qcov_hsp_perc

Percent query coverage per hsp (float, 0 to 100).

Added in BLAST+ 2.2.30.

This controls the addition of the -qcov_hsp_perc parameter and its associated value. Set this property to the argument value required.

property query

The sequence to search with.

This controls the addition of the -query parameter and its associated value. Set this property to the argument value required.

property query_loc

Location on the query sequence (Format: start-stop).

This controls the addition of the -query_loc parameter and its associated value. Set this property to the argument value required.

property remote

Execute search remotely?

Incompatible with: gilist, negative_gilist, subject_loc, num_threads, ...

This property controls the addition of the -remote switch, treat this property as a boolean.

property searchsp

Effective length of the search space (integer).

This controls the addition of the -searchsp parameter and its associated value. Set this property to the argument value required.

property seg

Filter query sequence with SEG (string).

Format: “yes”, “window locut hicut”, or “no” to disable Default is “12 2.2 2.5”

This controls the addition of the -seg parameter and its associated value. Set this property to the argument value required.

property seqidlist

Restrict search of database to list of SeqID’s.

Incompatible with: gilist, negative_gilist, remote, subject, subject_loc

This controls the addition of the -seqidlist parameter and its associated value. Set this property to the argument value required.

property show_gis

Show NCBI GIs in deflines?

This property controls the addition of the -show_gis switch, treat this property as a boolean.

property soft_masking

Apply filtering locations as soft masks (Boolean, Default = true).

This controls the addition of the -soft_masking parameter and its associated value. Set this property to the argument value required.

property subject

Subject sequence(s) to search.

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask

See also subject_loc.

This controls the addition of the -subject parameter and its associated value. Set this property to the argument value required.

property subject_loc

Location on the subject sequence (Format: start-stop).

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask, remote.

See also subject.

This controls the addition of the -subject_loc parameter and its associated value. Set this property to the argument value required.

property sum_statistics

Use sum statistics.

This property controls the addition of the -sum_statistics switch, treat this property as a boolean.

property sum_stats

Use sum statistics (boolean).

Added in BLAST+ 2.2.30.

This controls the addition of the -sum_stats parameter and its associated value. Set this property to the argument value required.

property task

Task to execute (string, blastp (default), blastp-fast or blastp-short).

This controls the addition of the -task parameter and its associated value. Set this property to the argument value required.

property threshold

Minimum score for words to be added to the BLAST lookup table (float).

This controls the addition of the -threshold parameter and its associated value. Set this property to the argument value required.

property ungapped

Perform ungapped alignment only?

This property controls the addition of the -ungapped switch, treat this property as a boolean.

property use_sw_tback

Compute locally optimal Smith-Waterman alignments?

This property controls the addition of the -use_sw_tback switch, treat this property as a boolean.

property version

Print version number; ignore other arguments.

This property controls the addition of the -version switch, treat this property as a boolean.

property window_size

Multiple hits window size, use 0 to specify 1-hit algorithm (integer).

This controls the addition of the -window_size parameter and its associated value. Set this property to the argument value required.

property word_size

Word size for wordfinder algorithm.

Integer. Minimum 2.

This controls the addition of the -word_size parameter and its associated value. Set this property to the argument value required.

property xdrop_gap

X-dropoff value (in bits) for preliminary gapped extensions (float).

This controls the addition of the -xdrop_gap parameter and its associated value. Set this property to the argument value required.

property xdrop_gap_final

X-dropoff value (in bits) for final gapped alignment (float).

This controls the addition of the -xdrop_gap_final parameter and its associated value. Set this property to the argument value required.

property xdrop_ungap

X-dropoff value (in bits) for ungapped extensions (float).

This controls the addition of the -xdrop_ungap parameter and its associated value. Set this property to the argument value required.

class Bio.Blast.Applications.**NcbiblastnCommandline**(*cmd='blastn'*, ***kwargs*)

Bases: _NcbiblastMain2SeqCommandline

Wrapper for the NCBI BLAST+ program blastn (for nucleotides).

With the release of BLAST+ (BLAST rewritten in C++ instead of C), the NCBI replaced the old blastall tool with separate tools for each of the searches. This wrapper therefore replaces BlastallCommandline with option -p blastn.

For example, to run a search against the "nt" nucleotide database using the FASTA nucleotide file "m_code.fasta" as the query, with an expectation value cut off of 0.001, saving the output to a file in XML format:

```
>>> from Bio.Blast.Applications import NcbiblastnCommandline
>>> cline = NcbiblastnCommandline(query="m_cold.fasta", db="nt", strand="plus",
...                               evalue=0.001, out="m_cold.xml", outfmt=5)
>>> cline
NcbiblastnCommandline(cmd='blastn', out='m_cold.xml', outfmt=5, query='m_cold.fasta
↪', db='nt', evalue=0.001, strand='plus')
>>> print(cline)
blastn -out m_cold.xml -outfmt 5 -query m_cold.fasta -db nt -evalue 0.001 -strand␣
↪plus
```

You would typically run the command line with cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='blastn'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property best_hit_overhang

Best Hit algorithm overhang value (float, recommended value: 0.1)

Float between 0.0 and 0.5 inclusive. Incompatible with: culling_limit.

This controls the addition of the -best_hit_overhang parameter and its associated value. Set this property to the argument value required.

property best_hit_score_edge

Best Hit algorithm score edge value (float).

Float between 0.0 and 0.5 inclusive. Recommended value: 0.1

Incompatible with: culling_limit.

This controls the addition of the -best_hit_score_edge parameter and its associated value. Set this property to the argument value required.

property culling_limit

Hit culling limit (integer).

If the query range of a hit is enveloped by that of at least this many higher-scoring hits, delete the hit.

Incompatible with: best_hit_overhang, best_hit_score_edge.

This controls the addition of the -culling_limit parameter and its associated value. Set this property to the argument value required.

property db

The database to BLAST against.

This controls the addition of the -db parameter and its associated value. Set this property to the argument value required.

property db_hard_mask

Filtering algorithm for hard masking (integer).

Filtering algorithm ID to apply to BLAST database as hard masking. Incompatible with: db_soft_mask, subject, subject_loc

This controls the addition of the -db_hard_mask parameter and its associated value. Set this property to the argument value required.

property db_soft_mask

Filtering algorithm for soft masking (integer).

Filtering algorithm ID to apply to BLAST database as soft masking. Incompatible with: db_hard_mask, subject, subject_loc

This controls the addition of the -db_soft_mask parameter and its associated value. Set this property to the argument value required.

property dbsize

Effective length of the database (integer).

This controls the addition of the -dbsize parameter and its associated value. Set this property to the argument value required.

property dust

Filter query sequence with DUST (string).

Format: 'yes', 'level window linker', or 'no' to disable.

Default = '20 64 1'.

This controls the addition of the -dust parameter and its associated value. Set this property to the argument value required.

property entrez_query

Restrict search with the given Entrez query (requires remote).

This controls the addition of the -entrez_query parameter and its associated value. Set this property to the argument value required.

property evalue

Expectation value cutoff.

This controls the addition of the -evalue parameter and its associated value. Set this property to the argument value required.

property export_search_strategy

File name to record the search strategy used.

Incompatible with: import_search_strategy

This controls the addition of the -export_search_strategy parameter and its associated value. Set this property to the argument value required.

property filtering_db

BLAST database containing filtering elements (i.e. repeats).

This controls the addition of the -filtering_db parameter and its associated value. Set this property to the argument value required.

property gapextend

Cost to extend a gap (integer).

This controls the addition of the -gapextend parameter and its associated value. Set this property to the argument value required.

property gapopen

Cost to open a gap (integer).

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property gilist

Restrict search of database to list of GI's.

Incompatible with: negative_gilist, seqidlist, negative_seqidlist, remote, subject, subject_loc

This controls the addition of the -gilist parameter and its associated value. Set this property to the argument value required.

property h

Print USAGE and DESCRIPTION; ignore other arguments.

This property controls the addition of the -h switch, treat this property as a boolean.

property help

Print USAGE, DESCRIPTION and ARGUMENTS description; ignore other arguments.

This property controls the addition of the -help switch, treat this property as a boolean.

property html

Produce HTML output? See also the outfmt option.

This property controls the addition of the -html switch, treat this property as a boolean.

property import_search_strategy

Search strategy to use.

Incompatible with: export_search_strategy

This controls the addition of the -import_search_strategy parameter and its associated value. Set this property to the argument value required.

property index_name

MegaBLAST database index name.

This controls the addition of the -index_name parameter and its associated value. Set this property to the argument value required.

property lcase_masking

Use lower case filtering in query and subject sequence(s)?

This property controls the addition of the -lcase_masking switch, treat this property as a boolean.

property line_length

Line length for formatting alignments (integer, at least 1, default 60).

Not applicable for outfmt > 4. Added in BLAST+ 2.2.30.

This controls the addition of the -line_length parameter and its associated value. Set this property to the argument value required.

property max_hsps

Set max number of HSPs saved per subject sequence

Ddefault 0 means no limit.

This controls the addition of the -max_hsps parameter and its associated value. Set this property to the argument value required.

property max_hsps_per_subject

Override max number of HSPs per subject saved for ungapped searches (integer).

This controls the addition of the -max_hsps_per_subject parameter and its associated value. Set this property to the argument value required.

property max_target_seqs

Maximum number of aligned sequences to keep (integer, at least one).

This controls the addition of the -max_target_seqs parameter and its associated value. Set this property to the argument value required.

property min_raw_gapped_score

Minimum raw gapped score to keep an alignment in the preliminary gapped and traceback stages (integer).

This controls the addition of the -min_raw_gapped_score parameter and its associated value. Set this property to the argument value required.

property negative_gilist

Restrict search of database to everything except the listed GIs.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_gilist parameter and its associated value. Set this property to the argument value required.

property negative_seqidlist

Restrict search of database to everything except listed SeqID's.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_seqidlist parameter and its associated value. Set this property to the argument value required.

property no_greedy

Use non-greedy dynamic programming extension

This property controls the addition of the -no_greedy switch, treat this property as a boolean.

property num_alignments

Number of database sequences to show num_alignments for.

Integer argument (at least zero). Default is 200. See also num_alignments.

This controls the addition of the -num_alignments parameter and its associated value. Set this property to the argument value required.

property num_descriptions

Number of database sequences to show one-line descriptions for.

Integer argument (at least zero). Default is 500. See also num_alignments.

This controls the addition of the -num_descriptions parameter and its associated value. Set this property to the argument value required.

property num_threads

Number of threads to use in the BLAST search.

Integer, at least one. Default is one. Incompatible with: remote

This controls the addition of the -num_threads parameter and its associated value. Set this property to the argument value required.

property off_diagonal_range

Number of off-diagonals to search for the 2nd hit (integer).

Expects a positive integer, or 0 (default) to turn off.Added in BLAST 2.2.23+

This controls the addition of the -off_diagonal_range parameter and its associated value. Set this property to the argument value required.

property out

Output file for alignment.

This controls the addition of the -out parameter and its associated value. Set this property to the argument value required.

property outfmt

Alignment view. Typically an integer 0-14 but for some formats can be named columns like '6 qseqid sseqid'. Use 5 for XML output (differs from classic BLAST which used 7 for XML).

This controls the addition of the -outfmt parameter and its associated value. Set this property to the argument value required.

property parse_deflines

Should the query and subject defline(s) be parsed?

This property controls the addition of the -parse_deflines switch, treat this property as a boolean.

property penalty

Penalty for a nucleotide mismatch (integer, at most zero).

This controls the addition of the -penalty parameter and its associated value. Set this property to the argument value required.

property perc_identity

Percent identity (real, 0 to 100 inclusive).

This controls the addition of the -perc_identity parameter and its associated value. Set this property to the argument value required.

property qcov_hsp_perc

Percent query coverage per hsp (float, 0 to 100).

Added in BLAST+ 2.2.30.

This controls the addition of the -qcov_hsp_perc parameter and its associated value. Set this property to the argument value required.

property query

The sequence to search with.

This controls the addition of the -query parameter and its associated value. Set this property to the argument value required.

property query_loc

Location on the query sequence (Format: start-stop).

This controls the addition of the -query_loc parameter and its associated value. Set this property to the argument value required.

property remote

Execute search remotely?

Incompatible with: gilist, negative_gilist, subject_loc, num_threads, ...

This property controls the addition of the -remote switch, treat this property as a boolean.

property reward

Reward for a nucleotide match (integer, at least zero).

This controls the addition of the -reward parameter and its associated value. Set this property to the argument value required.

property searchsp

Effective length of the search space (integer).

This controls the addition of the -searchsp parameter and its associated value. Set this property to the argument value required.

property seqidlist

Restrict search of database to list of SeqID's.

Incompatible with: gilist, negative_gilist, remote, subject, subject_loc

This controls the addition of the -seqidlist parameter and its associated value. Set this property to the argument value required.

property show_gis

Show NCBI GIs in deflines?

This property controls the addition of the -show_gis switch, treat this property as a boolean.

property soft_masking

Apply filtering locations as soft masks (Boolean, Default = true).

This controls the addition of the -soft_masking parameter and its associated value. Set this property to the argument value required.

property strand

Query strand(s) to search against database/subject.

Values allowed are "both" (default), "minus", "plus".

This controls the addition of the -strand parameter and its associated value. Set this property to the argument value required.

property subject

Subject sequence(s) to search.

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask

See also subject_loc.

This controls the addition of the -subject parameter and its associated value. Set this property to the argument value required.

property subject_loc

Location on the subject sequence (Format: start-stop).

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask, remote.

See also subject.

This controls the addition of the -subject_loc parameter and its associated value. Set this property to the argument value required.

property sum_statistics

Use sum statistics.

This property controls the addition of the -sum_statistics switch, treat this property as a boolean.

property sum_stats

Use sum statistics (boolean).

Added in BLAST+ 2.2.30.

This controls the addition of the -sum_stats parameter and its associated value. Set this property to the argument value required.

property task

Task to execute (string, default 'megablast')

Allowed values 'blastn', 'blastn-short', 'dc-megablast', 'megablast' (the default), or 'vecscreen'.

This controls the addition of the -task parameter and its associated value. Set this property to the argument value required.

property template_length

Discontiguous MegaBLAST template length (integer).

Allowed values: 16, 18, 21.

Requires: template_type.

This controls the addition of the -template_length parameter and its associated value. Set this property to the argument value required.

property template_type

Discontiguous MegaBLAST template type (string).

Allowed values: 'coding', 'coding_and_optimal' or 'optimal'. Requires: template_length.

This controls the addition of the -template_type parameter and its associated value. Set this property to the argument value required.

property ungapped

Perform ungapped alignment only?

This property controls the addition of the -ungapped switch, treat this property as a boolean.

property use_index

Use MegaBLAST database index (Boolean, Default = False)

This controls the addition of the -use_index parameter and its associated value. Set this property to the argument value required.

property version

Print version number; ignore other arguments.

This property controls the addition of the -version switch, treat this property as a boolean.

property window_masker_db

Enable WindowMasker filtering using this repeats database (string).

This controls the addition of the -window_masker_db parameter and its associated value. Set this property to the argument value required.

property window_masker_taxid

Enable WindowMasker filtering using a Taxonomic ID (integer).

This controls the addition of the -window_masker_taxid parameter and its associated value. Set this property to the argument value required.

property window_size

Multiple hits window size, use 0 to specify 1-hit algorithm (integer).

This controls the addition of the -window_size parameter and its associated value. Set this property to the argument value required.

property word_size

Word size for wordfinder algorithm.

Integer. Minimum 2.

This controls the addition of the -word_size parameter and its associated value. Set this property to the argument value required.

property xdrop_gap

X-dropoff value (in bits) for preliminary gapped extensions (float).

This controls the addition of the -xdrop_gap parameter and its associated value. Set this property to the argument value required.

property xdrop_gap_final

X-dropoff value (in bits) for final gapped alignment (float).

This controls the addition of the -xdrop_gap_final parameter and its associated value. Set this property to the argument value required.

property xdrop_ungap

X-dropoff value (in bits) for ungapped extensions (float).

This controls the addition of the -xdrop_ungap parameter and its associated value. Set this property to the argument value required.

class Bio.Blast.Applications.**NcbiblastxCommandline**(*cmd='blastx'*, ***kwargs*)

Bases: _NcbiblastMain2SeqCommandline

Wrapper for the NCBI BLAST+ program blastx (nucleotide query, protein database).

With the release of BLAST+ (BLAST rewritten in C++ instead of C), the NCBI replaced the old blastall tool with separate tools for each of the searches. This wrapper therefore replaces BlastallCommandline with option -p blastx.

```
>>> from Bio.Blast.Applications import NcbiblastxCommandline
>>> cline = NcbiblastxCommandline(query="m_cold.fasta", db="nr", evalue=0.001)
>>> cline
NcbiblastxCommandline(cmd='blastx', query='m_cold.fasta', db='nr', evalue=0.001)
>>> print(cline)
blastx -query m_cold.fasta -db nr -evalue 0.001
```

You would typically run the command line with cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='blastx'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property best_hit_overhang

Best Hit algorithm overhang value (float, recommended value: 0.1)

Float between 0.0 and 0.5 inclusive. Incompatible with: culling_limit.

This controls the addition of the -best_hit_overhang parameter and its associated value. Set this property to the argument value required.

property best_hit_score_edge

Best Hit algorithm score edge value (float).

Float between 0.0 and 0.5 inclusive. Recommended value: 0.1

Incompatible with: culling_limit.

This controls the addition of the -best_hit_score_edge parameter and its associated value. Set this property to the argument value required.

property comp_based_stats

Use composition-based statistics for blastp, blastx, or tblastn.

D or d: default (equivalent to 2)

0 or F or f: no composition-based statistics

1: Composition-based statistics as in NAR 29:2994-3005, 2001

2 or T or t : Composition-based score adjustment as in Bioinformatics 21:902-911, 2005, conditioned on sequence properties

3: Composition-based score adjustment as in Bioinformatics 21:902-911, 2005, unconditionally.

For programs other than tblastn, must either be absent or be D, F or 0

Default = 2.

This controls the addition of the -comp_based_stats parameter and its associated value. Set this property to the argument value required.

property culling_limit

Hit culling limit (integer).

If the query range of a hit is enveloped by that of at least this many higher-scoring hits, delete the hit.

Incompatible with: best_hit_overhang, best_hit_score_edge.

This controls the addition of the -culling_limit parameter and its associated value. Set this property to the argument value required.

property db

The database to BLAST against.

This controls the addition of the -db parameter and its associated value. Set this property to the argument value required.

property db_hard_mask

Filtering algorithm for hard masking (integer).

Filtering algorithm ID to apply to BLAST database as hard masking. Incompatible with: db_soft_mask, subject, subject_loc

This controls the addition of the -db_hard_mask parameter and its associated value. Set this property to the argument value required.

property db_soft_mask

Filtering algorithm for soft masking (integer).

Filtering algorithm ID to apply to BLAST database as soft masking. Incompatible with: db_hard_mask, subject, subject_loc

This controls the addition of the -db_soft_mask parameter and its associated value. Set this property to the argument value required.

property dbsize

Effective length of the database (integer).

This controls the addition of the -dbsize parameter and its associated value. Set this property to the argument value required.

property entrez_query

Restrict search with the given Entrez query (requires remote).

This controls the addition of the -entrez_query parameter and its associated value. Set this property to the argument value required.

property evalue

Expectation value cutoff.

This controls the addition of the -evalue parameter and its associated value. Set this property to the argument value required.

property export_search_strategy

File name to record the search strategy used.

Incompatible with: import_search_strategy

This controls the addition of the -export_search_strategy parameter and its associated value. Set this property to the argument value required.

property frame_shift_penalty

Frame shift penalty (integer, at least 1, default ignored) (OBSOLETE).

This was removed in BLAST 2.2.27+

This controls the addition of the -frame_shift_penalty parameter and its associated value. Set this property to the argument value required.

property gapextend

Cost to extend a gap (integer).

This controls the addition of the -gapextend parameter and its associated value. Set this property to the argument value required.

property gapopen

Cost to open a gap (integer).

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property gilist

Restrict search of database to list of GI's.

Incompatible with: negative_gilist, seqidlist, negative_seqidlist, remote, subject, subject_loc

This controls the addition of the -gilist parameter and its associated value. Set this property to the argument value required.

property h

Print USAGE and DESCRIPTION; ignore other arguments.

This property controls the addition of the -h switch, treat this property as a boolean.

property help

Print USAGE, DESCRIPTION and ARGUMENTS description; ignore other arguments.

This property controls the addition of the -help switch, treat this property as a boolean.

property html

Produce HTML output? See also the outfmt option.

This property controls the addition of the -html switch, treat this property as a boolean.

property import_search_strategy

Search strategy to use.

Incompatible with: export_search_strategy

This controls the addition of the -import_search_strategy parameter and its associated value. Set this property to the argument value required.

property lcase_masking

Use lower case filtering in query and subject sequence(s)?

This property controls the addition of the -lcase_masking switch, treat this property as a boolean.

property line_length

Line length for formatting alignments (integer, at least 1, default 60).

Not applicable for outfmt > 4. Added in BLAST+ 2.2.30.

This controls the addition of the -line_length parameter and its associated value. Set this property to the argument value required.

property matrix

Scoring matrix name (default BLOSUM62).

This controls the addition of the -matrix parameter and its associated value. Set this property to the argument value required.

property max_hsps

Set max number of HSPs saved per subject sequence

Ddefault 0 means no limit.

This controls the addition of the -max_hsps parameter and its associated value. Set this property to the argument value required.

property max_hsps_per_subject

Override max number of HSPs per subject saved for ungapped searches (integer).

This controls the addition of the -max_hsps_per_subject parameter and its associated value. Set this property to the argument value required.

property max_intron_length

Maximum intron length (integer).

Length of the largest intron allowed in a translated nucleotide sequence when linking multiple distinct alignments (a negative value disables linking). Default zero.

This controls the addition of the -max_intron_length parameter and its associated value. Set this property to the argument value required.

property max_target_seqs

Maximum number of aligned sequences to keep (integer, at least one).

This controls the addition of the -max_target_seqs parameter and its associated value. Set this property to the argument value required.

property negative_gilist

Restrict search of database to everything except the listed GIs.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_gilist parameter and its associated value. Set this property to the argument value required.

property negative_seqidlist

Restrict search of database to everything except listed SeqID's.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_seqidlist parameter and its associated value. Set this property to the argument value required.

property num_alignments

Number of database sequences to show num_alignments for.

Integer argument (at least zero). Default is 200. See also num_alignments.

This controls the addition of the -num_alignments parameter and its associated value. Set this property to the argument value required.

property num_descriptions

Number of database sequences to show one-line descriptions for.

Integer argument (at least zero). Default is 500. See also num_alignments.

This controls the addition of the -num_descriptions parameter and its associated value. Set this property to the argument value required.

property num_threads

Number of threads to use in the BLAST search.

Integer, at least one. Default is one. Incompatible with: remote

This controls the addition of the -num_threads parameter and its associated value. Set this property to the argument value required.

property out

Output file for alignment.

This controls the addition of the -out parameter and its associated value. Set this property to the argument value required.

property outfmt

Alignment view. Typically an integer 0-14 but for some formats can be named columns like '6 qseqid sseqid'. Use 5 for XML output (differs from classic BLAST which used 7 for XML).

This controls the addition of the -outfmt parameter and its associated value. Set this property to the argument value required.

property parse_deflines

Should the query and subject defline(s) be parsed?

This property controls the addition of the -parse_deflines switch, treat this property as a boolean.

property qcov_hsp_perc

Percent query coverage per hsp (float, 0 to 100).

Added in BLAST+ 2.2.30.

This controls the addition of the -qcov_hsp_perc parameter and its associated value. Set this property to the argument value required.

property query

The sequence to search with.

This controls the addition of the -query parameter and its associated value. Set this property to the argument value required.

property query_gencode

Genetic code to use to translate query (integer, default 1).

This controls the addition of the -query_gencode parameter and its associated value. Set this property to the argument value required.

property query_loc

Location on the query sequence (Format: start-stop).

This controls the addition of the -query_loc parameter and its associated value. Set this property to the argument value required.

property remote

Execute search remotely?

Incompatible with: gilist, negative_gilist, subject_loc, num_threads, ...

This property controls the addition of the -remote switch, treat this property as a boolean.

property searchsp

Effective length of the search space (integer).

This controls the addition of the -searchsp parameter and its associated value. Set this property to the argument value required.

property seg

Filter query sequence with SEG (string).

Format: "yes", "window locut hicut", or "no" to disable.Default is "12 2.2 2.5"

This controls the addition of the -seg parameter and its associated value. Set this property to the argument value required.

property seqidlist

Restrict search of database to list of SeqID's.

Incompatible with: gilist, negative_gilist, remote, subject, subject_loc

This controls the addition of the -seqidlist parameter and its associated value. Set this property to the argument value required.

property show_gis

Show NCBI GIs in deflines?

This property controls the addition of the -show_gis switch, treat this property as a boolean.

property soft_masking

Apply filtering locations as soft masks (Boolean, Default = true).

This controls the addition of the -soft_masking parameter and its associated value. Set this property to the argument value required.

property strand

Query strand(s) to search against database/subject.

Values allowed are "both" (default), "minus", "plus".

This controls the addition of the -strand parameter and its associated value. Set this property to the argument value required.

property subject

Subject sequence(s) to search.

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask

See also subject_loc.

This controls the addition of the -subject parameter and its associated value. Set this property to the argument value required.

property subject_loc

Location on the subject sequence (Format: start-stop).

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask, remote.

See also subject.

This controls the addition of the -subject_loc parameter and its associated value. Set this property to the argument value required.

property sum_statistics

Use sum statistics.

This property controls the addition of the -sum_statistics switch, treat this property as a boolean.

property sum_stats

Use sum statistics (boolean).

Added in BLAST+ 2.2.30.

This controls the addition of the -sum_stats parameter and its associated value. Set this property to the argument value required.

property task

Task to execute (string, blastx (default) or blastx-fast).

This controls the addition of the -task parameter and its associated value. Set this property to the argument value required.

property threshold

Minimum score for words to be added to the BLAST lookup table (float).

This controls the addition of the -threshold parameter and its associated value. Set this property to the argument value required.

property ungapped

Perform ungapped alignment only?

This property controls the addition of the -ungapped switch, treat this property as a boolean.

property use_sw_tback

Compute locally optimal Smith-Waterman alignments?

This property controls the addition of the -use_sw_tback switch, treat this property as a boolean.

property version

Print version number; ignore other arguments.

This property controls the addition of the -version switch, treat this property as a boolean.

property window_size

Multiple hits window size, use 0 to specify 1-hit algorithm (integer).

This controls the addition of the -window_size parameter and its associated value. Set this property to the argument value required.

property word_size

Word size for wordfinder algorithm.

Integer. Minimum 2.

This controls the addition of the -word_size parameter and its associated value. Set this property to the argument value required.

property xdrop_gap

X-dropoff value (in bits) for preliminary gapped extensions (float).

This controls the addition of the -xdrop_gap parameter and its associated value. Set this property to the argument value required.

property xdrop_gap_final

X-dropoff value (in bits) for final gapped alignment (float).

This controls the addition of the -xdrop_gap_final parameter and its associated value. Set this property to the argument value required.

property xdrop_ungap

X-dropoff value (in bits) for ungapped extensions (float).

This controls the addition of the -xdrop_ungap parameter and its associated value. Set this property to the argument value required.

class Bio.Blast.Applications.**NcbitblastnCommandline**(*cmd='tblastn'*, ***kwargs*)

Bases: _NcbiblastMain2SeqCommandline

Wrapper for the NCBI BLAST+ program tblastn.

With the release of BLAST+ (BLAST rewritten in C++ instead of C), the NCBI replaced the old blastall tool with separate tools for each of the searches. This wrapper therefore replaces BlastallCommandline with option -p tblastn.

```
>>> from Bio.Blast.Applications import NcbitblastnCommandline
>>> cline = NcbitblastnCommandline(help=True)
>>> cline
NcbitblastnCommandline(cmd='tblastn', help=True)
>>> print(cline)
tblastn -help
```

You would typically run the command line with cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='tblastn'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property best_hit_overhang

Best Hit algorithm overhang value (float, recommended value: 0.1)

Float between 0.0 and 0.5 inclusive. Incompatible with: culling_limit.

This controls the addition of the -best_hit_overhang parameter and its associated value. Set this property to the argument value required.

property best_hit_score_edge

Best Hit algorithm score edge value (float).

Float between 0.0 and 0.5 inclusive. Recommended value: 0.1

Incompatible with: culling_limit.

This controls the addition of the -best_hit_score_edge parameter and its associated value. Set this property to the argument value required.

property comp_based_stats

Use composition-based statistics (string, default 2, i.e. True).

0, F or f: no composition-based statistics

1: Composition-based statistics as in NAR 29:2994-3005, 2001

2, T or t, D or d : Composition-based score adjustment as in Bioinformatics 21:902-911, 2005, conditioned on sequence properties

3: Composition-based score adjustment as in Bioinformatics 21:902-911, 2005, unconditionally

Note that only tblastn supports values of 1 and 3.

This controls the addition of the -comp_based_stats parameter and its associated value. Set this property to the argument value required.

property culling_limit

Hit culling limit (integer).

If the query range of a hit is enveloped by that of at least this many higher-scoring hits, delete the hit.

Incompatible with: best_hit_overhang, best_hit_score_edge.

This controls the addition of the -culling_limit parameter and its associated value. Set this property to the argument value required.

property db

The database to BLAST against.

This controls the addition of the -db parameter and its associated value. Set this property to the argument value required.

property db_gencode

Genetic code to use to translate query (integer, default 1).

This controls the addition of the -db_gencode parameter and its associated value. Set this property to the argument value required.

property db_hard_mask

Filtering algorithm for hard masking (integer).

Filtering algorithm ID to apply to BLAST database as hard masking. Incompatible with: db_soft_mask, subject, subject_loc

This controls the addition of the -db_hard_mask parameter and its associated value. Set this property to the argument value required.

property db_soft_mask

Filtering algorithm for soft masking (integer).

Filtering algorithm ID to apply to BLAST database as soft masking. Incompatible with: db_hard_mask, subject, subject_loc

This controls the addition of the -db_soft_mask parameter and its associated value. Set this property to the argument value required.

property dbsize

Effective length of the database (integer).

This controls the addition of the -dbsize parameter and its associated value. Set this property to the argument value required.

property entrez_query

Restrict search with the given Entrez query (requires remote).

This controls the addition of the -entrez_query parameter and its associated value. Set this property to the argument value required.

property evalue

Expectation value cutoff.

This controls the addition of the -evalue parameter and its associated value. Set this property to the argument value required.

property export_search_strategy

File name to record the search strategy used.

Incompatible with: import_search_strategy

This controls the addition of the -export_search_strategy parameter and its associated value. Set this property to the argument value required.

property frame_shift_penalty

Frame shift penalty (integer, at least 1, default ignored) (OBSOLETE).

This was removed in BLAST 2.2.27+

This controls the addition of the -frame_shift_penalty parameter and its associated value. Set this property to the argument value required.

property gapextend

Cost to extend a gap (integer).

This controls the addition of the -gapextend parameter and its associated value. Set this property to the argument value required.

property gapopen

Cost to open a gap (integer).

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property gilist

Restrict search of database to list of GI's.

Incompatible with: negative_gilist, seqidlist, negative_seqidlist, remote, subject, subject_loc

This controls the addition of the -gilist parameter and its associated value. Set this property to the argument value required.

property h

Print USAGE and DESCRIPTION; ignore other arguments.

This property controls the addition of the -h switch, treat this property as a boolean.

property help

Print USAGE, DESCRIPTION and ARGUMENTS description; ignore other arguments.

This property controls the addition of the -help switch, treat this property as a boolean.

property html

Produce HTML output? See also the outfmt option.

This property controls the addition of the -html switch, treat this property as a boolean.

property import_search_strategy

Search strategy to use.

Incompatible with: export_search_strategy

This controls the addition of the -import_search_strategy parameter and its associated value. Set this property to the argument value required.

property in_pssm

PSI-BLAST checkpoint file.

Incompatible with: remote, query

This controls the addition of the -in_pssm parameter and its associated value. Set this property to the argument value required.

property lcase_masking

Use lower case filtering in query and subject sequence(s)?

This property controls the addition of the -lcase_masking switch, treat this property as a boolean.

property line_length

Line length for formatting alignments (integer, at least 1, default 60).

Not applicable for outfmt > 4. Added in BLAST+ 2.2.30.

This controls the addition of the -line_length parameter and its associated value. Set this property to the argument value required.

property matrix

Scoring matrix name (default BLOSUM62).

This controls the addition of the -matrix parameter and its associated value. Set this property to the argument value required.

property max_hsps

Set max number of HSPs saved per subject sequence

Ddefault 0 means no limit.

This controls the addition of the -max_hsps parameter and its associated value. Set this property to the argument value required.

property max_hsps_per_subject

Override max number of HSPs per subject saved for ungapped searches (integer).

This controls the addition of the -max_hsps_per_subject parameter and its associated value. Set this property to the argument value required.

property max_intron_length

Maximum intron length (integer).

Length of the largest intron allowed in a translated nucleotide sequence when linking multiple distinct alignments (a negative value disables linking). Default zero.

This controls the addition of the -max_intron_length parameter and its associated value. Set this property to the argument value required.

property max_target_seqs

Maximum number of aligned sequences to keep (integer, at least one).

This controls the addition of the -max_target_seqs parameter and its associated value. Set this property to the argument value required.

property negative_gilist

Restrict search of database to everything except the listed GIs.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_gilist parameter and its associated value. Set this property to the argument value required.

property negative_seqidlist

Restrict search of database to everything except listed SeqID's.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_seqidlist parameter and its associated value. Set this property to the argument value required.

property num_alignments

Number of database sequences to show num_alignments for.

Integer argument (at least zero). Default is 200. See also num_alignments.

This controls the addition of the -num_alignments parameter and its associated value. Set this property to the argument value required.

property num_descriptions

Number of database sequences to show one-line descriptions for.

Integer argument (at least zero). Default is 500. See also num_alignments.

This controls the addition of the -num_descriptions parameter and its associated value. Set this property to the argument value required.

property num_threads

Number of threads to use in the BLAST search.

Integer, at least one. Default is one. Incompatible with: remote

This controls the addition of the -num_threads parameter and its associated value. Set this property to the argument value required.

property out

Output file for alignment.

This controls the addition of the -out parameter and its associated value. Set this property to the argument value required.

property outfmt

Alignment view. Typically an integer 0-14 but for some formats can be named columns like '6 qseqid sseqid'. Use 5 for XML output (differs from classic BLAST which used 7 for XML).

This controls the addition of the -outfmt parameter and its associated value. Set this property to the argument value required.

property parse_deflines

Should the query and subject defline(s) be parsed?

This property controls the addition of the -parse_deflines switch, treat this property as a boolean.

property qcov_hsp_perc

Percent query coverage per hsp (float, 0 to 100).

Added in BLAST+ 2.2.30.

This controls the addition of the -qcov_hsp_perc parameter and its associated value. Set this property to the argument value required.

property query

The sequence to search with.

This controls the addition of the -query parameter and its associated value. Set this property to the argument value required.

property query_loc

Location on the query sequence (Format: start-stop).

This controls the addition of the -query_loc parameter and its associated value. Set this property to the argument value required.

property remote

Execute search remotely?

Incompatible with: gilist, negative_gilist, subject_loc, num_threads, ...

This property controls the addition of the -remote switch, treat this property as a boolean.

property searchsp

Effective length of the search space (integer).

This controls the addition of the -searchsp parameter and its associated value. Set this property to the argument value required.

property seg

Filter query sequence with SEG (string).

Format: "yes", "window locut hicut", or "no" to disable.

Default is "12 2.2 2.5"

This controls the addition of the -seg parameter and its associated value. Set this property to the argument value required.

property seqidlist

Restrict search of database to list of SeqID's.

Incompatible with: gilist, negative_gilist, remote, subject, subject_loc

This controls the addition of the -seqidlist parameter and its associated value. Set this property to the argument value required.

property show_gis

Show NCBI GIs in deflines?

This property controls the addition of the -show_gis switch, treat this property as a boolean.

property soft_masking

Apply filtering locations as soft masks (Boolean, Default = true).

This controls the addition of the -soft_masking parameter and its associated value. Set this property to the argument value required.

property subject

Subject sequence(s) to search.

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask

See also subject_loc.

This controls the addition of the -subject parameter and its associated value. Set this property to the argument value required.

property subject_loc

Location on the subject sequence (Format: start-stop).

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask, remote.

See also subject.

This controls the addition of the -subject_loc parameter and its associated value. Set this property to the argument value required.

property sum_statistics

Use sum statistics.

This property controls the addition of the -sum_statistics switch, treat this property as a boolean.

property sum_stats

Use sum statistics (boolean).

Added in BLAST+ 2.2.30.

This controls the addition of the -sum_stats parameter and its associated value. Set this property to the argument value required.

property task

Task to execute (string, tblastn (default) or tblastn-fast).

This controls the addition of the -task parameter and its associated value. Set this property to the argument value required.

property threshold

Minimum score for words to be added to the BLAST lookup table (float).

This controls the addition of the -threshold parameter and its associated value. Set this property to the argument value required.

property ungapped

Perform ungapped alignment only?

This property controls the addition of the -ungapped switch, treat this property as a boolean.

property use_sw_tback

Compute locally optimal Smith-Waterman alignments?

This property controls the addition of the -use_sw_tback switch, treat this property as a boolean.

property version

Print version number; ignore other arguments.

This property controls the addition of the -version switch, treat this property as a boolean.

property window_size

Multiple hits window size, use 0 to specify 1-hit algorithm (integer).

This controls the addition of the -window_size parameter and its associated value. Set this property to the argument value required.

property word_size

Word size for wordfinder algorithm.

Integer. Minimum 2.

This controls the addition of the -word_size parameter and its associated value. Set this property to the argument value required.

property xdrop_gap

X-dropoff value (in bits) for preliminary gapped extensions (float).

This controls the addition of the -xdrop_gap parameter and its associated value. Set this property to the argument value required.

property xdrop_gap_final

X-dropoff value (in bits) for final gapped alignment (float).

This controls the addition of the -xdrop_gap_final parameter and its associated value. Set this property to the argument value required.

property xdrop_ungap

X-dropoff value (in bits) for ungapped extensions (float).

This controls the addition of the -xdrop_ungap parameter and its associated value. Set this property to the argument value required.

class Bio.Blast.Applications.**NcbitblastxCommandline**(*cmd='tblastx'*, ***kwargs*)

Bases: _NcbiblastMain2SeqCommandline

Wrapper for the NCBI BLAST+ program tblastx.

With the release of BLAST+ (BLAST rewritten in C++ instead of C), the NCBI replaced the old blastall tool with separate tools for each of the searches. This wrapper therefore replaces BlastallCommandline with option -p tblastx.

```
>>> from Bio.Blast.Applications import NcbitblastxCommandline
>>> cline = NcbitblastxCommandline(help=True)
>>> cline
NcbitblastxCommandline(cmd='tblastx', help=True)
>>> print(cline)
tblastx -help
```

You would typically run the command line with cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='tblastx'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property best_hit_overhang

Best Hit algorithm overhang value (float, recommended value: 0.1)

Float between 0.0 and 0.5 inclusive. Incompatible with: culling_limit.

This controls the addition of the -best_hit_overhang parameter and its associated value. Set this property to the argument value required.

property best_hit_score_edge

Best Hit algorithm score edge value (float).

Float between 0.0 and 0.5 inclusive. Recommended value: 0.1

Incompatible with: culling_limit.

This controls the addition of the -best_hit_score_edge parameter and its associated value. Set this property to the argument value required.

property culling_limit

Hit culling limit (integer).

If the query range of a hit is enveloped by that of at least this many higher-scoring hits, delete the hit.

Incompatible with: best_hit_overhang, best_hit_score_edge.

This controls the addition of the -culling_limit parameter and its associated value. Set this property to the argument value required.

property db

The database to BLAST against.

This controls the addition of the -db parameter and its associated value. Set this property to the argument value required.

property db_gencode

Genetic code to use to translate query (integer, default 1).

This controls the addition of the -db_gencode parameter and its associated value. Set this property to the argument value required.

property db_hard_mask

Filtering algorithm for hard masking (integer).

Filtering algorithm ID to apply to BLAST database as hard masking. Incompatible with: db_soft_mask, subject, subject_loc

This controls the addition of the -db_hard_mask parameter and its associated value. Set this property to the argument value required.

property db_soft_mask

Filtering algorithm for soft masking (integer).

Filtering algorithm ID to apply to BLAST database as soft masking. Incompatible with: db_hard_mask, subject, subject_loc

This controls the addition of the -db_soft_mask parameter and its associated value. Set this property to the argument value required.

property dbsize

Effective length of the database (integer).

This controls the addition of the -dbsize parameter and its associated value. Set this property to the argument value required.

property entrez_query

Restrict search with the given Entrez query (requires remote).

This controls the addition of the -entrez_query parameter and its associated value. Set this property to the argument value required.

property evalue

Expectation value cutoff.

This controls the addition of the -evalue parameter and its associated value. Set this property to the argument value required.

property export_search_strategy

File name to record the search strategy used.

Incompatible with: import_search_strategy

This controls the addition of the -export_search_strategy parameter and its associated value. Set this property to the argument value required.

property gapextend

Cost to extend a gap (integer).

This controls the addition of the -gapextend parameter and its associated value. Set this property to the argument value required.

property gapopen

Cost to open a gap (integer).

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property gilist

Restrict search of database to list of GI's.

Incompatible with: negative_gilist, seqidlist, negative_seqidlist, remote, subject, subject_loc

This controls the addition of the -gilist parameter and its associated value. Set this property to the argument value required.

property h

Print USAGE and DESCRIPTION; ignore other arguments.

This property controls the addition of the -h switch, treat this property as a boolean.

property help

Print USAGE, DESCRIPTION and ARGUMENTS description; ignore other arguments.

This property controls the addition of the -help switch, treat this property as a boolean.

property html

Produce HTML output? See also the outfmt option.

This property controls the addition of the -html switch, treat this property as a boolean.

property import_search_strategy

Search strategy to use.

Incompatible with: export_search_strategy

This controls the addition of the -import_search_strategy parameter and its associated value. Set this property to the argument value required.

property lcase_masking

Use lower case filtering in query and subject sequence(s)?

This property controls the addition of the -lcase_masking switch, treat this property as a boolean.

property line_length

Line length for formatting alignments (integer, at least 1, default 60).

Not applicable for outfmt > 4. Added in BLAST+ 2.2.30.

This controls the addition of the -line_length parameter and its associated value. Set this property to the argument value required.

property matrix

Scoring matrix name (default BLOSUM62).

This controls the addition of the -matrix parameter and its associated value. Set this property to the argument value required.

property max_hsps

Set max number of HSPs saved per subject sequence

Ddefault 0 means no limit.

This controls the addition of the -max_hsps parameter and its associated value. Set this property to the argument value required.

property max_hsps_per_subject

Override max number of HSPs per subject saved for ungapped searches (integer).

This controls the addition of the -max_hsps_per_subject parameter and its associated value. Set this property to the argument value required.

property max_intron_length

Maximum intron length (integer).

Length of the largest intron allowed in a translated nucleotide sequence when linking multiple distinct alignments (a negative value disables linking). Default zero.

This controls the addition of the -max_intron_length parameter and its associated value. Set this property to the argument value required.

property max_target_seqs

Maximum number of aligned sequences to keep (integer, at least one).

This controls the addition of the -max_target_seqs parameter and its associated value. Set this property to the argument value required.

property negative_gilist

Restrict search of database to everything except the listed GIs.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_gilist parameter and its associated value. Set this property to the argument value required.

property negative_seqidlist

Restrict search of database to everything except listed SeqID's.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_seqidlist parameter and its associated value. Set this property to the argument value required.

property num_alignments

Number of database sequences to show num_alignments for.

Integer argument (at least zero). Default is 200. See also num_alignments.

This controls the addition of the -num_alignments parameter and its associated value. Set this property to the argument value required.

property num_descriptions

Number of database sequences to show one-line descriptions for.

Integer argument (at least zero). Default is 500. See also num_alignments.

This controls the addition of the -num_descriptions parameter and its associated value. Set this property to the argument value required.

property num_threads

Number of threads to use in the BLAST search.

Integer, at least one. Default is one. Incompatible with: remote

This controls the addition of the -num_threads parameter and its associated value. Set this property to the argument value required.

property out

Output file for alignment.

This controls the addition of the -out parameter and its associated value. Set this property to the argument value required.

property outfmt

Alignment view. Typically an integer 0-14 but for some formats can be named columns like '6 qseqid sseqid'. Use 5 for XML output (differs from classic BLAST which used 7 for XML).

This controls the addition of the -outfmt parameter and its associated value. Set this property to the argument value required.

property parse_deflines

Should the query and subject defline(s) be parsed?

This property controls the addition of the -parse_deflines switch, treat this property as a boolean.

property qcov_hsp_perc

Percent query coverage per hsp (float, 0 to 100).

Added in BLAST+ 2.2.30.

This controls the addition of the -qcov_hsp_perc parameter and its associated value. Set this property to the argument value required.

property query

The sequence to search with.

This controls the addition of the -query parameter and its associated value. Set this property to the argument value required.

property query_gencode

Genetic code to use to translate query (integer, default 1).

This controls the addition of the -query_gencode parameter and its associated value. Set this property to the argument value required.

property query_loc

Location on the query sequence (Format: start-stop).

This controls the addition of the -query_loc parameter and its associated value. Set this property to the argument value required.

property remote

Execute search remotely?

Incompatible with: gilist, negative_gilist, subject_loc, num_threads, ...

This property controls the addition of the -remote switch, treat this property as a boolean.

property searchsp

Effective length of the search space (integer).

This controls the addition of the -searchsp parameter and its associated value. Set this property to the argument value required.

property seg

Filter query sequence with SEG (string).

Format: "yes", "window locut hicut", or "no" to disable.

Default is "12 2.2 2.5"

This controls the addition of the -seg parameter and its associated value. Set this property to the argument value required.

property seqidlist

Restrict search of database to list of SeqID's.

Incompatible with: gilist, negative_gilist, remote, subject, subject_loc

This controls the addition of the -seqidlist parameter and its associated value. Set this property to the argument value required.

property show_gis

Show NCBI GIs in deflines?

This property controls the addition of the -show_gis switch, treat this property as a boolean.

property soft_masking

Apply filtering locations as soft masks (Boolean, Default = true).

This controls the addition of the -soft_masking parameter and its associated value. Set this property to the argument value required.

property strand

Query strand(s) to search against database/subject.

Values allowed are "both" (default), "minus", "plus".

This controls the addition of the -strand parameter and its associated value. Set this property to the argument value required.

property subject

Subject sequence(s) to search.

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask

See also subject_loc.

This controls the addition of the -subject parameter and its associated value. Set this property to the argument value required.

property subject_loc

Location on the subject sequence (Format: start-stop).

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask, remote.

See also subject.

This controls the addition of the -subject_loc parameter and its associated value. Set this property to the argument value required.

property sum_statistics

Use sum statistics.

This property controls the addition of the -sum_statistics switch, treat this property as a boolean.

property sum_stats

Use sum statistics (boolean).

Added in BLAST+ 2.2.30.

This controls the addition of the -sum_stats parameter and its associated value. Set this property to the argument value required.

property threshold

Minimum score for words to be added to the BLAST lookup table (float).

This controls the addition of the -threshold parameter and its associated value. Set this property to the argument value required.

property version

Print version number; ignore other arguments.

This property controls the addition of the -version switch, treat this property as a boolean.

property window_size

Multiple hits window size, use 0 to specify 1-hit algorithm (integer).

This controls the addition of the -window_size parameter and its associated value. Set this property to the argument value required.

property word_size

Word size for wordfinder algorithm.

Integer. Minimum 2.

This controls the addition of the -word_size parameter and its associated value. Set this property to the argument value required.

property xdrop_gap

X-dropoff value (in bits) for preliminary gapped extensions (float).

This controls the addition of the -xdrop_gap parameter and its associated value. Set this property to the argument value required.

property xdrop_gap_final

X-dropoff value (in bits) for final gapped alignment (float).

This controls the addition of the -xdrop_gap_final parameter and its associated value. Set this property to the argument value required.

property xdrop_ungap

X-dropoff value (in bits) for ungapped extensions (float).

This controls the addition of the -xdrop_ungap parameter and its associated value. Set this property to the argument value required.

class Bio.Blast.Applications.**NcbipsiblastCommandline**(*cmd='psiblast'*, ***kwargs*)

Bases: _Ncbiblast2SeqCommandline

Wrapper for the NCBI BLAST+ program psiblast.

With the release of BLAST+ (BLAST rewritten in C++ instead of C), the NCBI replaced the old blastpgp tool with a similar tool psiblast. This wrapper therefore replaces BlastpgpCommandline, the wrapper for blastpgp.

```
>>> from Bio.Blast.Applications import NcbipsiblastCommandline
>>> cline = NcbipsiblastCommandline(help=True)
>>> cline
NcbipsiblastCommandline(cmd='psiblast', help=True)
>>> print(cline)
psiblast -help
```

You would typically run the command line with cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='psiblast'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property best_hit_overhang

Best Hit algorithm overhang value (float, recommended value: 0.1)

Float between 0.0 and 0.5 inclusive. Incompatible with: culling_limit.

This controls the addition of the -best_hit_overhang parameter and its associated value. Set this property to the argument value required.

property best_hit_score_edge

Best Hit algorithm score edge value (float).

Float between 0.0 and 0.5 inclusive. Recommended value: 0.1

Incompatible with: culling_limit.

This controls the addition of the -best_hit_score_edge parameter and its associated value. Set this property to the argument value required.

property comp_based_stats

Use composition-based statistics (string, default 2, i.e. True).

0, F or f: no composition-based statistics

2, T or t, D or d : Composition-based score adjustment as in Bioinformatics 21:902-911, 2005, conditioned on sequence properties

Note that tblastn also supports values of 1 and 3.

This controls the addition of the -comp_based_stats parameter and its associated value. Set this property to the argument value required.

property culling_limit

Hit culling limit (integer).

If the query range of a hit is enveloped by that of at least this many higher-scoring hits, delete the hit.

Incompatible with: best_hit_overhang, best_hit_score_edge.

This controls the addition of the -culling_limit parameter and its associated value. Set this property to the argument value required.

property db

The database to BLAST against.

This controls the addition of the -db parameter and its associated value. Set this property to the argument value required.

property dbsize

Effective length of the database (integer).

This controls the addition of the -dbsize parameter and its associated value. Set this property to the argument value required.

property entrez_query

Restrict search with the given Entrez query (requires remote).

This controls the addition of the -entrez_query parameter and its associated value. Set this property to the argument value required.

property evalue

Expectation value cutoff.

This controls the addition of the -evalue parameter and its associated value. Set this property to the argument value required.

property export_search_strategy

File name to record the search strategy used.

Incompatible with: import_search_strategy

This controls the addition of the -export_search_strategy parameter and its associated value. Set this property to the argument value required.

property gap_trigger

Number of bits to trigger gapping (float, default 22).

This controls the addition of the -gap_trigger parameter and its associated value. Set this property to the argument value required.

property gapextend

Cost to extend a gap (integer).

This controls the addition of the -gapextend parameter and its associated value. Set this property to the argument value required.

property gapopen

Cost to open a gap (integer).

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property gilist

Restrict search of database to list of GI's.

Incompatible with: negative_gilist, seqidlist, negative_seqidlist, remote, subject, subject_loc

This controls the addition of the -gilist parameter and its associated value. Set this property to the argument value required.

property h

Print USAGE and DESCRIPTION; ignore other arguments.

This property controls the addition of the -h switch, treat this property as a boolean.

property help

Print USAGE, DESCRIPTION and ARGUMENTS description; ignore other arguments.

This property controls the addition of the -help switch, treat this property as a boolean.

property html

Produce HTML output? See also the outfmt option.

This property controls the addition of the -html switch, treat this property as a boolean.

property ignore_msa_master

Ignore the master sequence when creating PSSM.

Requires: in_msa Incompatible with: msa_master_idx, in_pssm, query, query_loc, phi_pattern

This property controls the addition of the -ignore_msa_master switch, treat this property as a boolean.

property import_search_strategy

Search strategy to use.

Incompatible with: export_search_strategy

This controls the addition of the -import_search_strategy parameter and its associated value. Set this property to the argument value required.

property in_msa

File name of multiple sequence alignment to restart PSI-BLAST.

Incompatible with: in_pssm, query

This controls the addition of the -in_msa parameter and its associated value. Set this property to the argument value required.

property in_pssm

PSI-BLAST checkpoint file.

Incompatible with: in_msa, query, phi_pattern

This controls the addition of the -in_pssm parameter and its associated value. Set this property to the argument value required.

property inclusion_ethresh

E-value inclusion threshold for pairwise alignments (float, default 0.002).

This controls the addition of the -inclusion_ethresh parameter and its associated value. Set this property to the argument value required.

property lcase_masking

Use lower case filtering in query and subject sequence(s)?

This property controls the addition of the -lcase_masking switch, treat this property as a boolean.

property line_length

Line length for formatting alignments (integer, at least 1, default 60).

Not applicable for outfmt > 4. Added in BLAST+ 2.2.30.

This controls the addition of the -line_length parameter and its associated value. Set this property to the argument value required.

property matrix

Scoring matrix name (default BLOSUM62).

This controls the addition of the -matrix parameter and its associated value. Set this property to the argument value required.

property max_hsps

Set max number of HSPs saved per subject sequence

Ddefault 0 means no limit.

This controls the addition of the -max_hsps parameter and its associated value. Set this property to the argument value required.

property max_hsps_per_subject

Override max number of HSPs per subject saved for ungapped searches (integer).

This controls the addition of the -max_hsps_per_subject parameter and its associated value. Set this property to the argument value required.

property max_target_seqs

Maximum number of aligned sequences to keep (integer, at least one).

This controls the addition of the -max_target_seqs parameter and its associated value. Set this property to the argument value required.

property msa_master_idx

Index of sequence to use as master in MSA.

Index (1-based) of sequence to use as the master in the multiple sequence alignment. If not specified, the first sequence is used.

This controls the addition of the -msa_master_idx parameter and its associated value. Set this property to the argument value required.

property negative_gilist

Restrict search of database to everything except the listed GIs.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_gilist parameter and its associated value. Set this property to the argument value required.

property negative_seqidlist

Restrict search of database to everything except listed SeqID's.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_seqidlist parameter and its associated value. Set this property to the argument value required.

property num_alignments

Number of database sequences to show num_alignments for.

Integer argument (at least zero). Default is 200. See also num_alignments.

This controls the addition of the -num_alignments parameter and its associated value. Set this property to the argument value required.

property num_descriptions

Number of database sequences to show one-line descriptions for.

Integer argument (at least zero). Default is 500. See also num_alignments.

This controls the addition of the -num_descriptions parameter and its associated value. Set this property to the argument value required.

property num_iterations

Number of iterations to perform (integer, at least one).

Default is one. Incompatible with: remote

This controls the addition of the -num_iterations parameter and its associated value. Set this property to the argument value required.

property num_threads

Number of threads to use in the BLAST search.

Integer, at least one. Default is one. Incompatible with: remote

This controls the addition of the -num_threads parameter and its associated value. Set this property to the argument value required.

property out

Output file for alignment.

This controls the addition of the -out parameter and its associated value. Set this property to the argument value required.

property out_ascii_pssm

File name to store ASCII version of PSSM.

This controls the addition of the -out_ascii_pssm parameter and its associated value. Set this property to the argument value required.

property out_pssm

File name to store checkpoint file.

This controls the addition of the -out_pssm parameter and its associated value. Set this property to the argument value required.

property outfmt

Alignment view. Typically an integer 0-14 but for some formats can be named columns like '6 qseqid sseqid'. Use 5 for XML output (differs from classic BLAST which used 7 for XML).

This controls the addition of the -outfmt parameter and its associated value. Set this property to the argument value required.

property parse_deflines

Should the query and subject defline(s) be parsed?

This property controls the addition of the -parse_deflines switch, treat this property as a boolean.

property phi_pattern

File name containing pattern to search.

Incompatible with: in_pssm

This controls the addition of the -phi_pattern parameter and its associated value. Set this property to the argument value required.

property pseudocount

Pseudo-count value used when constructing PSSM.

Integer. Default is zero.

This controls the addition of the -pseudocount parameter and its associated value. Set this property to the argument value required.

property qcov_hsp_perc

Percent query coverage per hsp (float, 0 to 100).

Added in BLAST+ 2.2.30.

This controls the addition of the -qcov_hsp_perc parameter and its associated value. Set this property to the argument value required.

property query

The sequence to search with.

This controls the addition of the -query parameter and its associated value. Set this property to the argument value required.

property query_loc

Location on the query sequence (Format: start-stop).

This controls the addition of the -query_loc parameter and its associated value. Set this property to the argument value required.

property remote

Execute search remotely?

Incompatible with: gilist, negative_gilist, subject_loc, num_threads, ...

This property controls the addition of the -remote switch, treat this property as a boolean.

property save_each_pssm

Save PSSM after each iteration

File name is given in -save_pssm or -save_ascii_pssm options.

This property controls the addition of the -save_each_pssm switch, treat this property as a boolean.

property save_pssm_after_last_round

Save PSSM after the last database search.

This property controls the addition of the -save_pssm_after_last_round switch, treat this property as a boolean.

property searchsp

Effective length of the search space (integer).

This controls the addition of the -searchsp parameter and its associated value. Set this property to the argument value required.

property seg

Filter query sequence with SEG (string).

Format: "yes", "window locut hicut", or "no" to disable. Default is "12 2.2 2.5"

This controls the addition of the -seg parameter and its associated value. Set this property to the argument value required.

property seqidlist

Restrict search of database to list of SeqID's.

Incompatible with: gilist, negative_gilist, remote, subject, subject_loc

This controls the addition of the -seqidlist parameter and its associated value. Set this property to the argument value required.

property show_gis

Show NCBI GIs in deflines?

This property controls the addition of the -show_gis switch, treat this property as a boolean.

property soft_masking

Apply filtering locations as soft masks (Boolean, Default = true).

This controls the addition of the -soft_masking parameter and its associated value. Set this property to the argument value required.

property subject

Subject sequence(s) to search.

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask

See also subject_loc.

This controls the addition of the -subject parameter and its associated value. Set this property to the argument value required.

property subject_loc

Location on the subject sequence (Format: start-stop).

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask, remote.

See also subject.

This controls the addition of the -subject_loc parameter and its associated value. Set this property to the argument value required.

property sum_statistics

Use sum statistics.

This property controls the addition of the -sum_statistics switch, treat this property as a boolean.

property sum_stats

Use sum statistics (boolean).

Added in BLAST+ 2.2.30.

This controls the addition of the -sum_stats parameter and its associated value. Set this property to the argument value required.

property threshold

Minimum score for words to be added to the BLAST lookup table (float).

This controls the addition of the -threshold parameter and its associated value. Set this property to the argument value required.

property use_sw_tback

Compute locally optimal Smith-Waterman alignments?

This property controls the addition of the -use_sw_tback switch, treat this property as a boolean.

property version

Print version number; ignore other arguments.

This property controls the addition of the -version switch, treat this property as a boolean.

property window_size

Multiple hits window size, use 0 to specify 1-hit algorithm (integer).

This controls the addition of the -window_size parameter and its associated value. Set this property to the argument value required.

property word_size

Word size for wordfinder algorithm.

Integer. Minimum 2.

This controls the addition of the -word_size parameter and its associated value. Set this property to the argument value required.

property xdrop_gap

X-dropoff value (in bits) for preliminary gapped extensions (float).

This controls the addition of the -xdrop_gap parameter and its associated value. Set this property to the argument value required.

property xdrop_gap_final

X-dropoff value (in bits) for final gapped alignment (float).

This controls the addition of the -xdrop_gap_final parameter and its associated value. Set this property to the argument value required.

property xdrop_ungap

X-dropoff value (in bits) for ungapped extensions (float).

This controls the addition of the -xdrop_ungap parameter and its associated value. Set this property to the argument value required.

class Bio.Blast.Applications.**NcbirpsblastCommandline**(*cmd='rpsblast'*, ***kwargs*)

Bases: _NcbiblastCommandline

Wrapper for the NCBI BLAST+ program rpsblast.

With the release of BLAST+ (BLAST rewritten in C++ instead of C), the NCBI replaced the old rpsblast tool with a similar tool of the same name. This wrapper replaces RpsBlastCommandline, the wrapper for the old rpsblast.

```
>>> from Bio.Blast.Applications import NcbirpsblastCommandline
>>> cline = NcbirpsblastCommandline(help=True)
>>> cline
NcbirpsblastCommandline(cmd='rpsblast', help=True)
>>> print(cline)
rpsblast -help
```

You would typically run the command line with cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='rpsblast'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property best_hit_overhang

Best Hit algorithm overhang value (recommended value: 0.1).

Float between 0.0 and 0.5 inclusive. Incompatible with: culling_limit.

This controls the addition of the -best_hit_overhang parameter and its associated value. Set this property to the argument value required.

property best_hit_score_edge

Best Hit algorithm score edge value (recommended value: 0.1).

Float between 0.0 and 0.5 inclusive. Incompatible with: culling_limit.

This controls the addition of the -best_hit_score_edge parameter and its associated value. Set this property to the argument value required.

property comp_based_stats

Use composition-based statistics.

D or d: default (equivalent to 0)

0 or F or f: Simplified Composition-based statistics as in Bioinformatics 15:1000-1011, 1999

1 or T or t: Composition-based statistics as in NAR 29:2994-3005, 2001

Default = 0.

This controls the addition of the -comp_based_stats parameter and its associated value. Set this property to the argument value required.

property culling_limit

Hit culling limit (integer).

If the query range of a hit is enveloped by that of at least this many higher-scoring hits, delete the hit. Incompatible with: best_hit_overhang, best_hit_score_edge.

This controls the addition of the -culling_limit parameter and its associated value. Set this property to the argument value required.

property db

The database to BLAST against.

This controls the addition of the -db parameter and its associated value. Set this property to the argument value required.

property dbsize

Effective length of the database (integer).

This controls the addition of the -dbsize parameter and its associated value. Set this property to the argument value required.

property entrez_query

Restrict search with the given Entrez query (requires remote).

This controls the addition of the -entrez_query parameter and its associated value. Set this property to the argument value required.

property evalue

Expectation value cutoff.

This controls the addition of the -evalue parameter and its associated value. Set this property to the argument value required.

property export_search_strategy

File name to record the search strategy used.

Incompatible with: import_search_strategy

This controls the addition of the -export_search_strategy parameter and its associated value. Set this property to the argument value required.

property gilist

Restrict search of database to list of GI's.

Incompatible with: negative_gilist, seqidlist, negative_seqidlist, remote, subject, subject_loc

This controls the addition of the -gilist parameter and its associated value. Set this property to the argument value required.

property h

Print USAGE and DESCRIPTION; ignore other arguments.

This property controls the addition of the -h switch, treat this property as a boolean.

property help

Print USAGE, DESCRIPTION and ARGUMENTS description; ignore other arguments.

This property controls the addition of the -help switch, treat this property as a boolean.

property html

Produce HTML output? See also the outfmt option.

This property controls the addition of the -html switch, treat this property as a boolean.

property import_search_strategy

Search strategy to use.

Incompatible with: export_search_strategy

This controls the addition of the -import_search_strategy parameter and its associated value. Set this property to the argument value required.

property lcase_masking

Use lower case filtering in query and subject sequence(s)?

This property controls the addition of the -lcase_masking switch, treat this property as a boolean.

property line_length

Line length for formatting alignments (integer, at least 1, default 60).

Not applicable for outfmt > 4. Added in BLAST+ 2.2.30.

This controls the addition of the -line_length parameter and its associated value. Set this property to the argument value required.

property max_hsps

Set max number of HSPs saved per subject sequence

Ddefault 0 means no limit.

This controls the addition of the -max_hsps parameter and its associated value. Set this property to the argument value required.

property max_hsps_per_subject

Override max number of HSPs per subject saved for ungapped searches (integer).

This controls the addition of the -max_hsps_per_subject parameter and its associated value. Set this property to the argument value required.

property max_target_seqs

Maximum number of aligned sequences to keep (integer, at least one).

This controls the addition of the -max_target_seqs parameter and its associated value. Set this property to the argument value required.

property negative_gilist

Restrict search of database to everything except the listed GIs.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_gilist parameter and its associated value. Set this property to the argument value required.

property negative_seqidlist

Restrict search of database to everything except listed SeqID's.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_seqidlist parameter and its associated value. Set this property to the argument value required.

property num_alignments

Number of database sequences to show num_alignments for.

Integer argument (at least zero). Default is 200. See also num_alignments.

This controls the addition of the -num_alignments parameter and its associated value. Set this property to the argument value required.

property num_descriptions

Number of database sequences to show one-line descriptions for.

Integer argument (at least zero). Default is 500. See also num_alignments.

This controls the addition of the -num_descriptions parameter and its associated value. Set this property to the argument value required.

property num_threads

Number of threads to use in the BLAST search.

Integer, at least one. Default is one. Incompatible with: remote

This controls the addition of the -num_threads parameter and its associated value. Set this property to the argument value required.

property out

Output file for alignment.

This controls the addition of the -out parameter and its associated value. Set this property to the argument value required.

property outfmt

Alignment view. Typically an integer 0-14 but for some formats can be named columns like '6 qseqid sseqid'. Use 5 for XML output (differs from classic BLAST which used 7 for XML).

This controls the addition of the -outfmt parameter and its associated value. Set this property to the argument value required.

property parse_deflines

Should the query and subject defline(s) be parsed?

This property controls the addition of the -parse_deflines switch, treat this property as a boolean.

property qcov_hsp_perc

Percent query coverage per hsp (float, 0 to 100).

Added in BLAST+ 2.2.30.

This controls the addition of the -qcov_hsp_perc parameter and its associated value. Set this property to the argument value required.

property query

The sequence to search with.

This controls the addition of the -query parameter and its associated value. Set this property to the argument value required.

property query_loc

Location on the query sequence (Format: start-stop).

This controls the addition of the -query_loc parameter and its associated value. Set this property to the argument value required.

property remote

Execute search remotely?

Incompatible with: gilist, negative_gilist, subject_loc, num_threads, ...

This property controls the addition of the -remote switch, treat this property as a boolean.

property searchsp

Effective length of the search space (integer).

This controls the addition of the -searchsp parameter and its associated value. Set this property to the argument value required.

property seg

Filter query sequence with SEG (string).

Format: "yes", "window locut hicut", or "no" to disable.Default is "12 2.2 2.5"

This controls the addition of the -seg parameter and its associated value. Set this property to the argument value required.

property seqidlist

Restrict search of database to list of SeqID's.

Incompatible with: gilist, negative_gilist, remote, subject, subject_loc

This controls the addition of the -seqidlist parameter and its associated value. Set this property to the argument value required.

property show_gis

Show NCBI GIs in deflines?

This property controls the addition of the -show_gis switch, treat this property as a boolean.

property soft_masking

Apply filtering locations as soft masks (Boolean, Default = true).

This controls the addition of the -soft_masking parameter and its associated value. Set this property to the argument value required.

property sum_statistics

Use sum statistics.

This property controls the addition of the -sum_statistics switch, treat this property as a boolean.

property sum_stats

Use sum statistics (boolean).

Added in BLAST+ 2.2.30.

This controls the addition of the -sum_stats parameter and its associated value. Set this property to the argument value required.

property use_sw_tback

Compute locally optimal Smith-Waterman alignments?

This property controls the addition of the -use_sw_tback switch, treat this property as a boolean.

property version

Print version number; ignore other arguments.

This property controls the addition of the -version switch, treat this property as a boolean.

property window_size

Multiple hits window size, use 0 to specify 1-hit algorithm (integer).

This controls the addition of the -window_size parameter and its associated value. Set this property to the argument value required.

property word_size

Word size for wordfinder algorithm.

Integer. Minimum 2.

This controls the addition of the -word_size parameter and its associated value. Set this property to the argument value required.

property xdrop_gap

X-dropoff value (in bits) for preliminary gapped extensions (float).

This controls the addition of the -xdrop_gap parameter and its associated value. Set this property to the argument value required.

property xdrop_gap_final

X-dropoff value (in bits) for final gapped alignment (float).

This controls the addition of the -xdrop_gap_final parameter and its associated value. Set this property to the argument value required.

property xdrop_ungap

X-dropoff value (in bits) for ungapped extensions (float).

This controls the addition of the -xdrop_ungap parameter and its associated value. Set this property to the argument value required.

class Bio.Blast.Applications.**NcbirpstblastnCommandline**(*cmd='rpstblastn'*, ***kwargs*)

Bases: _NcbiblastCommandline

Wrapper for the NCBI BLAST+ program rpstblastn.

With the release of BLAST+ (BLAST rewritten in C++ instead of C), the NCBI replaced the old rpsblast tool with a similar tool of the same name, and a separate tool rpstblastn for Translated Reverse Position Specific BLAST.

```
>>> from Bio.Blast.Applications import NcbirpstblastnCommandline
>>> cline = NcbirpstblastnCommandline(help=True)
>>> cline
NcbirpstblastnCommandline(cmd='rpstblastn', help=True)
>>> print(cline)
rpstblastn -help
```

You would typically run the command line with cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='rpstblastn'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property comp_based_stats

Use composition-based statistics.

D or d: default (equivalent to 0)

0 or F or f: Simplified Composition-based statistics as in Bioinformatics 15:1000-1011, 1999

1 or T or t: Composition-based statistics as in NAR 29:2994-3005, 2001

Default = 0.

This controls the addition of the -comp_based_stats parameter and its associated value. Set this property to the argument value required.

property db

The database to BLAST against.

This controls the addition of the -db parameter and its associated value. Set this property to the argument value required.

property dbsize

Effective length of the database (integer).

This controls the addition of the -dbsize parameter and its associated value. Set this property to the argument value required.

property entrez_query

Restrict search with the given Entrez query (requires remote).

This controls the addition of the -entrez_query parameter and its associated value. Set this property to the argument value required.

property evalue

Expectation value cutoff.

This controls the addition of the -evalue parameter and its associated value. Set this property to the argument value required.

property export_search_strategy

File name to record the search strategy used.

Incompatible with: import_search_strategy

This controls the addition of the -export_search_strategy parameter and its associated value. Set this property to the argument value required.

property gilist

Restrict search of database to list of GI's.

Incompatible with: negative_gilist, seqidlist, negative_seqidlist, remote, subject, subject_loc

This controls the addition of the -gilist parameter and its associated value. Set this property to the argument value required.

property h

Print USAGE and DESCRIPTION; ignore other arguments.

This property controls the addition of the -h switch, treat this property as a boolean.

property help

Print USAGE, DESCRIPTION and ARGUMENTS description; ignore other arguments.

This property controls the addition of the -help switch, treat this property as a boolean.

property html

Produce HTML output? See also the outfmt option.

This property controls the addition of the -html switch, treat this property as a boolean.

property import_search_strategy

Search strategy to use.

Incompatible with: export_search_strategy

This controls the addition of the -import_search_strategy parameter and its associated value. Set this property to the argument value required.

property lcase_masking

Use lower case filtering in query and subject sequence(s)?

This property controls the addition of the -lcase_masking switch, treat this property as a boolean.

property line_length

Line length for formatting alignments (integer, at least 1, default 60).

Not applicable for outfmt > 4. Added in BLAST+ 2.2.30.

This controls the addition of the -line_length parameter and its associated value. Set this property to the argument value required.

property max_hsps

Set max number of HSPs saved per subject sequence

Ddefault 0 means no limit.

This controls the addition of the -max_hsps parameter and its associated value. Set this property to the argument value required.

property max_hsps_per_subject

Override max number of HSPs per subject saved for ungapped searches (integer).

This controls the addition of the -max_hsps_per_subject parameter and its associated value. Set this property to the argument value required.

property max_target_seqs

Maximum number of aligned sequences to keep (integer, at least one).

This controls the addition of the -max_target_seqs parameter and its associated value. Set this property to the argument value required.

property negative_gilist

Restrict search of database to everything except the listed GIs.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_gilist parameter and its associated value. Set this property to the argument value required.

property negative_seqidlist

Restrict search of database to everything except listed SeqID's.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_seqidlist parameter and its associated value. Set this property to the argument value required.

property num_alignments

Number of database sequences to show num_alignments for.

Integer argument (at least zero). Default is 200. See also num_alignments.

This controls the addition of the -num_alignments parameter and its associated value. Set this property to the argument value required.

property num_descriptions

Number of database sequences to show one-line descriptions for.

Integer argument (at least zero). Default is 500. See also num_alignments.

This controls the addition of the -num_descriptions parameter and its associated value. Set this property to the argument value required.

property num_threads

Number of threads to use in the BLAST search.

Integer, at least one. Default is one. Incompatible with: remote

This controls the addition of the -num_threads parameter and its associated value. Set this property to the argument value required.

property out

Output file for alignment.

This controls the addition of the -out parameter and its associated value. Set this property to the argument value required.

property outfmt

Alignment view. Typically an integer 0-14 but for some formats can be named columns like '6 qseqid sseqid'. Use 5 for XML output (differs from classic BLAST which used 7 for XML).

This controls the addition of the -outfmt parameter and its associated value. Set this property to the argument value required.

property parse_deflines

Should the query and subject defline(s) be parsed?

This property controls the addition of the -parse_deflines switch, treat this property as a boolean.

property qcov_hsp_perc

Percent query coverage per hsp (float, 0 to 100).

Added in BLAST+ 2.2.30.

This controls the addition of the -qcov_hsp_perc parameter and its associated value. Set this property to the argument value required.

property query

The sequence to search with.

This controls the addition of the -query parameter and its associated value. Set this property to the argument value required.

property query_gencode

Genetic code to use to translate query (integer, default 1).

This controls the addition of the -query_gencode parameter and its associated value. Set this property to the argument value required.

property query_loc

Location on the query sequence (Format: start-stop).

This controls the addition of the -query_loc parameter and its associated value. Set this property to the argument value required.

property remote

Execute search remotely?

Incompatible with: gilist, negative_gilist, subject_loc, num_threads, ...

This property controls the addition of the -remote switch, treat this property as a boolean.

property searchsp

Effective length of the search space (integer).

This controls the addition of the -searchsp parameter and its associated value. Set this property to the argument value required.

property seg

Filter query sequence with SEG (string).

Format: "yes", "window locut hicut", or "no" to disable. Default is "12 2.2 2.5"

This controls the addition of the -seg parameter and its associated value. Set this property to the argument value required.

property seqidlist

Restrict search of database to list of SeqID's.

Incompatible with: gilist, negative_gilist, remote, subject, subject_loc

This controls the addition of the -seqidlist parameter and its associated value. Set this property to the argument value required.

property show_gis

Show NCBI GIs in deflines?

This property controls the addition of the -show_gis switch, treat this property as a boolean.

property soft_masking

Apply filtering locations as soft masks (Boolean, Default = true).

This controls the addition of the -soft_masking parameter and its associated value. Set this property to the argument value required.

property strand

Query strand(s) to search against database/subject.

Values allowed are "both" (default), "minus", "plus".

This controls the addition of the -strand parameter and its associated value. Set this property to the argument value required.

property sum_statistics

Use sum statistics.

This property controls the addition of the -sum_statistics switch, treat this property as a boolean.

property sum_stats

Use sum statistics (boolean).

Added in BLAST+ 2.2.30.

This controls the addition of the -sum_stats parameter and its associated value. Set this property to the argument value required.

property ungapped

Perform ungapped alignment only?

This property controls the addition of the -ungapped switch, treat this property as a boolean.

property use_sw_tback

Compute locally optimal Smith-Waterman alignments?

This property controls the addition of the -use_sw_tback switch, treat this property as a boolean.

property version

Print version number; ignore other arguments.

This property controls the addition of the -version switch, treat this property as a boolean.

property window_size

Multiple hits window size, use 0 to specify 1-hit algorithm (integer).

This controls the addition of the -window_size parameter and its associated value. Set this property to the argument value required.

property word_size

Word size for wordfinder algorithm.

Integer. Minimum 2.

This controls the addition of the -word_size parameter and its associated value. Set this property to the argument value required.

property xdrop_gap

X-dropoff value (in bits) for preliminary gapped extensions (float).

This controls the addition of the -xdrop_gap parameter and its associated value. Set this property to the argument value required.

property xdrop_gap_final

X-dropoff value (in bits) for final gapped alignment (float).

This controls the addition of the -xdrop_gap_final parameter and its associated value. Set this property to the argument value required.

property xdrop_ungap

X-dropoff value (in bits) for ungapped extensions (float).

This controls the addition of the -xdrop_ungap parameter and its associated value. Set this property to the argument value required.

class Bio.Blast.Applications.**NcbiblastformatterCommandline**(*cmd='blast_formatter'*, ***kwargs*)

Bases: _NcbibaseblastCommandline

Wrapper for the NCBI BLAST+ program blast_formatter.

With the release of BLAST 2.2.24+ (i.e. the BLAST suite rewritten in C++ instead of C), the NCBI added the ASN.1 output format option to all the search tools, and extended the blast_formatter to support this as input.

The blast_formatter command allows you to convert the ASN.1 output into the other output formats (XML, tabular, plain text, HTML).

```
>>> from Bio.Blast.Applications import NcbiblastformatterCommandline
>>> cline = NcbiblastformatterCommandline(archive="example.asn", outfmt=5, out=
↪"example.xml")
>>> cline
NcbiblastformatterCommandline(cmd='blast_formatter', out='example.xml', outfmt=5,␣
↪archive='example.asn')
>>> print(cline)
blast_formatter -out example.xml -outfmt 5 -archive example.asn
```

You would typically run the command line with cline() or via the Python subprocess module, as described in the Biopython tutorial.

Note that this wrapper is for the version of blast_formatter from BLAST 2.2.24+ (or later) which is when the NCBI first announced the inclusion this tool. There was actually an early version in BLAST 2.2.23+ (and possibly in older releases) but this did not have the -archive option (instead -rid is a mandatory argument), and is not supported by this wrapper.

__init__(*cmd='blast_formatter'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property archive

Archive file of results, not compatible with rid arg.

This controls the addition of the -archive parameter and its associated value. Set this property to the argument value required.

property h

Print USAGE and DESCRIPTION; ignore other arguments.

This property controls the addition of the -h switch, treat this property as a boolean.

property help

Print USAGE, DESCRIPTION and ARGUMENTS description; ignore other arguments.

This property controls the addition of the -help switch, treat this property as a boolean.

property html

Produce HTML output? See also the outfmt option.

This property controls the addition of the -html switch, treat this property as a boolean.

property line_length

Line length for formatting alignments (integer, at least 1, default 60).

Not applicable for outfmt > 4. Added in BLAST+ 2.2.30.

This controls the addition of the -line_length parameter and its associated value. Set this property to the argument value required.

property max_target_seqs

Maximum number of aligned sequences to keep.

This controls the addition of the -max_target_seqs parameter and its associated value. Set this property to the argument value required.

property num_alignments

Number of database sequences to show num_alignments for.

Integer argument (at least zero). Default is 200. See also num_alignments.

This controls the addition of the -num_alignments parameter and its associated value. Set this property to the argument value required.

property num_descriptions

Number of database sequences to show one-line descriptions for.

Integer argument (at least zero). Default is 500. See also num_alignments.

This controls the addition of the -num_descriptions parameter and its associated value. Set this property to the argument value required.

property out

Output file for alignment.

This controls the addition of the -out parameter and its associated value. Set this property to the argument value required.

property outfmt

Alignment view. Typically an integer 0-14 but for some formats can be named columns like '6 qseqid sseqid'. Use 5 for XML output (differs from classic BLAST which used 7 for XML).

This controls the addition of the -outfmt parameter and its associated value. Set this property to the argument value required.

property parse_deflines

Should the query and subject defline(s) be parsed?

This property controls the addition of the -parse_deflines switch, treat this property as a boolean.

property rid

BLAST Request ID (RID), not compatible with archive arg.

This controls the addition of the -rid parameter and its associated value. Set this property to the argument value required.

property show_gis

Show NCBI GIs in deflines?

This property controls the addition of the -show_gis switch, treat this property as a boolean.

property version

Print version number; ignore other arguments.

This property controls the addition of the -version switch, treat this property as a boolean.

class Bio.Blast.Applications.**NcbideltablastCommandline**(*cmd='deltablast'*, ***kwargs*)

Bases: _Ncbiblast2SeqCommandline

Create a commandline for the NCBI BLAST+ program deltablast (for proteins).

This is a wrapper for the deltablast command line command included in the NCBI BLAST+ software (not present in the original BLAST).

```
>>> from Bio.Blast.Applications import NcbideltablastCommandline
>>> cline = NcbideltablastCommandline(query="rosemary.pro", db="nr",
...                                   evalue=0.001, remote=True)
>>> cline
NcbideltablastCommandline(cmd='deltablast', query='rosemary.pro', db='nr', evalue=0.
↪001, remote=True)
>>> print(cline)
deltablast -query rosemary.pro -db nr -evalue 0.001 -remote
```

You would typically run the command line with cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='deltablast'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property best_hit_overhang

Best Hit algorithm overhang value (float, recommended value: 0.1)

Float between 0.0 and 0.5 inclusive. Incompatible with: culling_limit.

This controls the addition of the -best_hit_overhang parameter and its associated value. Set this property to the argument value required.

property best_hit_score_edge

Best Hit algorithm score edge value (float).

Float between 0.0 and 0.5 inclusive. Recommended value: 0.1

Incompatible with: culling_limit.

This controls the addition of the -best_hit_score_edge parameter and its associated value. Set this property to the argument value required.

property comp_based_stats

Use composition-based statistics (string, default 2, i.e. True).

0, F or f: no composition-based statistics.

2, T or t, D or d : Composition-based score adjustment as in Bioinformatics 21:902-911, 2005, conditioned on sequence properties

Note that tblastn also supports values of 1 and 3.

This controls the addition of the -comp_based_stats parameter and its associated value. Set this property to the argument value required.

property culling_limit

Hit culling limit (integer).

If the query range of a hit is enveloped by that of at least this many higher-scoring hits, delete the hit.

Incompatible with: best_hit_overhang, best_hit_score_edge.

This controls the addition of the -culling_limit parameter and its associated value. Set this property to the argument value required.

property db

The database to BLAST against.

This controls the addition of the -db parameter and its associated value. Set this property to the argument value required.

property dbsize

Effective length of the database (integer).

This controls the addition of the -dbsize parameter and its associated value. Set this property to the argument value required.

property domain_inclusion_ethresh

E-value inclusion threshold for alignments with conserved domains.

(float, Default is 0.05)

This controls the addition of the -domain_inclusion_ethresh parameter and its associated value. Set this property to the argument value required.

property entrez_query

Restrict search with the given Entrez query (requires remote).

This controls the addition of the -entrez_query parameter and its associated value. Set this property to the argument value required.

property evalue

Expectation value cutoff.

This controls the addition of the -evalue parameter and its associated value. Set this property to the argument value required.

property export_search_strategy

File name to record the search strategy used.

Incompatible with: import_search_strategy

This controls the addition of the -export_search_strategy parameter and its associated value. Set this property to the argument value required.

property gap_trigger

Number of bits to trigger gapping. Default = 22.

This controls the addition of the -gap_trigger parameter and its associated value. Set this property to the argument value required.

property gapextend

Cost to extend a gap (integer).

This controls the addition of the -gapextend parameter and its associated value. Set this property to the argument value required.

property gapopen

Cost to open a gap (integer).

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property gilist

Restrict search of database to list of GI's.

Incompatible with: negative_gilist, seqidlist, negative_seqidlist, remote, subject, subject_loc

This controls the addition of the -gilist parameter and its associated value. Set this property to the argument value required.

property h

Print USAGE and DESCRIPTION; ignore other arguments.

This property controls the addition of the -h switch, treat this property as a boolean.

property help

Print USAGE, DESCRIPTION and ARGUMENTS description; ignore other arguments.

This property controls the addition of the -help switch, treat this property as a boolean.

property html

Produce HTML output? See also the outfmt option.

This property controls the addition of the -html switch, treat this property as a boolean.

property import_search_strategy

Search strategy to use.

Incompatible with: export_search_strategy

This controls the addition of the -import_search_strategy parameter and its associated value. Set this property to the argument value required.

property inclusion_ethresh

Pairwise alignment e-value inclusion threshold (float, default 0.002).

This controls the addition of the -inclusion_ethresh parameter and its associated value. Set this property to the argument value required.

property lcase_masking

Use lower case filtering in query and subject sequence(s)?

This property controls the addition of the -lcase_masking switch, treat this property as a boolean.

property line_length

Line length for formatting alignments (integer, at least 1, default 60).

Not applicable for outfmt > 4. Added in BLAST+ 2.2.30.

This controls the addition of the -line_length parameter and its associated value. Set this property to the argument value required.

property matrix

Scoring matrix name (default BLOSUM62).

This controls the addition of the -matrix parameter and its associated value. Set this property to the argument value required.

property max_hsps

Set max number of HSPs saved per subject sequence

Ddefault 0 means no limit.

This controls the addition of the -max_hsps parameter and its associated value. Set this property to the argument value required.

property max_hsps_per_subject

Override max number of HSPs per subject saved for ungapped searches (integer).

This controls the addition of the -max_hsps_per_subject parameter and its associated value. Set this property to the argument value required.

property max_target_seqs

Maximum number of aligned sequences to keep (integer, at least one).

This controls the addition of the -max_target_seqs parameter and its associated value. Set this property to the argument value required.

property negative_gilist

Restrict search of database to everything except the listed GIs.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_gilist parameter and its associated value. Set this property to the argument value required.

property negative_seqidlist

Restrict search of database to everything except listed SeqID's.

Incompatible with: gilist, seqidlist, remote, subject, subject_loc

This controls the addition of the -negative_seqidlist parameter and its associated value. Set this property to the argument value required.

property num_alignments

Number of database sequences to show num_alignments for.

Integer argument (at least zero). Default is 200. See also num_alignments.

This controls the addition of the -num_alignments parameter and its associated value. Set this property to the argument value required.

property num_descriptions

Number of database sequences to show one-line descriptions for.

Integer argument (at least zero). Default is 500. See also num_alignments.

This controls the addition of the -num_descriptions parameter and its associated value. Set this property to the argument value required.

property num_iterations

Number of iterations to perform. (integer >=1, Default is 1).

Incompatible with: remote

This controls the addition of the -num_iterations parameter and its associated value. Set this property to the argument value required.

property num_threads

Number of threads to use in the BLAST search.

Integer, at least one. Default is one. Incompatible with: remote

This controls the addition of the -num_threads parameter and its associated value. Set this property to the argument value required.

property out

Output file for alignment.

This controls the addition of the -out parameter and its associated value. Set this property to the argument value required.

property out_ascii_pssm

File name to store ASCII version of PSSM.

This controls the addition of the -out_ascii_pssm parameter and its associated value. Set this property to the argument value required.

property out_pssm

File name to store checkpoint file.

This controls the addition of the -out_pssm parameter and its associated value. Set this property to the argument value required.

property outfmt

Alignment view. Typically an integer 0-14 but for some formats can be named columns like '6 qseqid sseqid'. Use 5 for XML output (differs from classic BLAST which used 7 for XML).

This controls the addition of the -outfmt parameter and its associated value. Set this property to the argument value required.

property parse_deflines

Should the query and subject defline(s) be parsed?

This property controls the addition of the -parse_deflines switch, treat this property as a boolean.

property pseudocount

Pseudo-count value used when constructing PSSM (integer, default 0).

This controls the addition of the -pseudocount parameter and its associated value. Set this property to the argument value required.

property qcov_hsp_perc

Percent query coverage per hsp (float, 0 to 100).

Added in BLAST+ 2.2.30.

This controls the addition of the -qcov_hsp_perc parameter and its associated value. Set this property to the argument value required.

property query

The sequence to search with.

This controls the addition of the -query parameter and its associated value. Set this property to the argument value required.

property query_loc

Location on the query sequence (Format: start-stop).

This controls the addition of the -query_loc parameter and its associated value. Set this property to the argument value required.

property remote

Execute search remotely?

Incompatible with: gilist, negative_gilist, subject_loc, num_threads, ...

This property controls the addition of the -remote switch, treat this property as a boolean.

property rpsdb

BLAST domain database name (dtring, Default = 'cdd_delta').

This controls the addition of the -rpsdb parameter and its associated value. Set this property to the argument value required.

property save_each_pssm

Save PSSM after each iteration.

File name is given in -save_pssm or -save_ascii_pssm options.

This property controls the addition of the -save_each_pssm switch, treat this property as a boolean.

property save_pssm_after_last_round

Save PSSM after the last database search.

This property controls the addition of the -save_pssm_after_last_round switch, treat this property as a boolean.

property searchsp

Effective length of the search space (integer).

This controls the addition of the -searchsp parameter and its associated value. Set this property to the argument value required.

property seg

Filter query sequence with SEG (string).

Format: "yes", "window locut hicut", or "no" to disable. Default is "12 2.2 2.5"

This controls the addition of the -seg parameter and its associated value. Set this property to the argument value required.

property seqidlist

Restrict search of database to list of SeqID's.

Incompatible with: gilist, negative_gilist, remote, subject, subject_loc

This controls the addition of the -seqidlist parameter and its associated value. Set this property to the argument value required.

property show_domain_hits

Show domain hits?

Incompatible with: remote, subject

This property controls the addition of the -show_domain_hits switch, treat this property as a boolean.

property show_gis

Show NCBI GIs in deflines?

This property controls the addition of the -show_gis switch, treat this property as a boolean.

property soft_masking

Apply filtering locations as soft masks (Boolean, Default = true).

This controls the addition of the -soft_masking parameter and its associated value. Set this property to the argument value required.

property subject

Subject sequence(s) to search.

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask

See also subject_loc.

This controls the addition of the -subject parameter and its associated value. Set this property to the argument value required.

property subject_loc

Location on the subject sequence (Format: start-stop).

Incompatible with: db, gilist, seqidlist, negative_gilist, negative_seqidlist, db_soft_mask, db_hard_mask, remote.

See also subject.

This controls the addition of the -subject_loc parameter and its associated value. Set this property to the argument value required.

property sum_statistics

Use sum statistics.

This property controls the addition of the -sum_statistics switch, treat this property as a boolean.

property sum_stats

Use sum statistics (boolean).

Added in BLAST+ 2.2.30.

This controls the addition of the -sum_stats parameter and its associated value. Set this property to the argument value required.

property threshold

Minimum score for words to be added to the BLAST lookup table (float).

This controls the addition of the -threshold parameter and its associated value. Set this property to the argument value required.

property use_sw_tback

Compute locally optimal Smith-Waterman alignments?

This property controls the addition of the -use_sw_tback switch, treat this property as a boolean.

property version

Print version number; ignore other arguments.

This property controls the addition of the -version switch, treat this property as a boolean.

property window_size

Multiple hits window size, use 0 to specify 1-hit algorithm (integer).

This controls the addition of the -window_size parameter and its associated value. Set this property to the argument value required.

property word_size

Word size for wordfinder algorithm.

Integer. Minimum 2.

This controls the addition of the -word_size parameter and its associated value. Set this property to the argument value required.

property xdrop_gap

X-dropoff value (in bits) for preliminary gapped extensions (float).

This controls the addition of the -xdrop_gap parameter and its associated value. Set this property to the argument value required.

property xdrop_gap_final

X-dropoff value (in bits) for final gapped alignment (float).

This controls the addition of the -xdrop_gap_final parameter and its associated value. Set this property to the argument value required.

property xdrop_ungap

X-dropoff value (in bits) for ungapped extensions (float).

This controls the addition of the -xdrop_ungap parameter and its associated value. Set this property to the argument value required.

class Bio.Blast.Applications.**NcbimakeblastdbCommandline**(*cmd='makeblastdb'*, ***kwargs*)

Bases: *AbstractCommandline*

Wrapper for the NCBI BLAST+ program makeblastdb.

This is a wrapper for the NCBI BLAST+ makeblastdb application to create BLAST databases. By default, this creates a blast database with the same name as the input file. The default output location is the same directory as the input.

```
>>> from Bio.Blast.Applications import NcbimakeblastdbCommandline
>>> cline = NcbimakeblastdbCommandline(dbtype="prot",
...                                    input_file="NC_005816.faa")
>>> cline
NcbimakeblastdbCommandline(cmd='makeblastdb', dbtype='prot', input_file='NC_005816.
↪faa')
>>> print(cline)
makeblastdb -dbtype prot -in NC_005816.faa
```

You would typically run the command line with cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='makeblastdb'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property blastdb_version

Version of BLAST database to be created. Tip: use BLAST database version 4 on 32 bit CPU. Default = 5

This controls the addition of the -blastdb_version parameter and its associated value. Set this property to the argument value required.

property dbtype

Molecule type of target db ('nucl' or 'prot').

This controls the addition of the -dbtype parameter and its associated value. Set this property to the argument value required.

property gi_mask

Create GI indexed masking data.

This property controls the addition of the -gi_mask switch, treat this property as a boolean.

property gi_mask_name

Comma-separated list of masking data output files.

This controls the addition of the -gi_mask_name parameter and its associated value. Set this property to the argument value required.

property h

Print USAGE and DESCRIPTION; ignore other arguments.

This property controls the addition of the -h switch, treat this property as a boolean.

property hash_index

Create index of sequence hash values.

This property controls the addition of the -hash_index switch, treat this property as a boolean.

property help

Print USAGE, DESCRIPTION and ARGUMENTS description; ignore other arguments.

This property controls the addition of the -help switch, treat this property as a boolean.

property input_file

Input file/database name.

This controls the addition of the -in parameter and its associated value. Set this property to the argument value required.

property input_type

Type of the data specified in input_file.

Default = 'fasta'. Added in BLAST 2.2.26.

This controls the addition of the -input_type parameter and its associated value. Set this property to the argument value required.

property logfile

File to which the program log should be redirected.

This controls the addition of the -logfile parameter and its associated value. Set this property to the argument value required.

property mask_data

Comma-separated list of input files containing masking data as produced by NCBI masking applications (e.g. dustmasker, segmasker, windowmasker).

This controls the addition of the -mask_data parameter and its associated value. Set this property to the argument value required.

property mask_desc

Comma-separated list of free form strings to describe the masking algorithm details.

This controls the addition of the -mask_desc parameter and its associated value. Set this property to the argument value required.

property mask_id

Comma-separated list of strings to uniquely identify the masking algorithm.

This controls the addition of the -mask_id parameter and its associated value. Set this property to the argument value required.

property max_file_sz

Maximum file size for BLAST database files. Default = '1GB'.

This controls the addition of the -max_file_sz parameter and its associated value. Set this property to the argument value required.

property out

Output file for alignment.

This controls the addition of the -out parameter and its associated value. Set this property to the argument value required.

property parse_seqids

Option to parse seqid for FASTA input if set.

For all other input types, seqids are parsed automatically

This property controls the addition of the -parse_seqids switch, treat this property as a boolean.

property taxid

Taxonomy ID to assign to all sequences.

This controls the addition of the -taxid parameter and its associated value. Set this property to the argument value required.

property taxid_map

Text file mapping sequence IDs to taxonomy IDs.

Format:<SequenceId> <TaxonomyId><newline>

This controls the addition of the -taxid_map parameter and its associated value. Set this property to the argument value required.

property title

Title for BLAST database.

This controls the addition of the -title parameter and its associated value. Set this property to the argument value required.

property version

Print version number; ignore other arguments.

This property controls the addition of the -version switch, treat this property as a boolean.

Bio.Blast.NCBIWWW module

Code to invoke the NCBI BLAST server over the internet.

This module provides code to work with the WWW version of BLAST provided by the NCBI. https://blast.ncbi.nlm.nih.gov/

Variables:

- email Set the Blast email parameter (default is None).
- tool Set the Blast tool parameter (default is `biopython`).

Bio.Blast.NCBIWWW.**qblast**(*program*, *database*, *sequence*, *url_base=NCBI_BLAST_URL*, *auto_format=None*, *composition_based_statistics=None*, *db_genetic_code=None*, *endpoints=None*, *entrez_query='(none)'*, *expect=10.0*, *filter=None*, *gapcosts=None*, *genetic_code=None*, *hitlist_size=50*, *i_thresh=None*, *layout=None*, *lcase_mask=None*, *matrix_name=None*, *nucl_penalty=None*, *nucl_reward=None*, *other_advanced=None*, *perc_ident=None*, *phi_pattern=None*, *query_file=None*, *query_believe_defline=None*, *query_from=None*, *query_to=None*, *searchsp_eff=None*, *service=None*, *threshold=None*, *ungapped_alignment=None*, *word_size=None*, *short_query=None*, *alignments=500*, *alignment_view=None*, *descriptions=500*, *entrez_links_new_window=None*, *expect_low=None*, *expect_high=None*, *format_entrez_query=None*, *format_object=None*, *format_type='XML'*, *ncbi_gi=None*, *results_file=None*, *show_overview=None*, *megablast=None*, *template_type=None*, *template_length=None*, *username='blast'*, *password=None*)

BLAST search using NCBI's QBLAST server or a cloud service provider.

Supports all parameters of the old qblast API for Put and Get.

Please note that NCBI uses the new Common URL API for BLAST searches on the internet (http://ncbi.github.io/blast-cloud/dev/api.html). Thus, some of the parameters used by this function are not (or are no longer) officially supported by NCBI. Although they are still functioning, this may change in the future.

The Common URL API (http://ncbi.github.io/blast-cloud/dev/api.html) allows doing BLAST searches on cloud servers. To use this feature, please set `url_base='http://host.my.cloud.service.provider.com/cgi-bin/blast.cgi'` and `format_object='Alignment'`. For more details, please see https://blast.ncbi.nlm.nih.gov/Blast.cgi?PAGE_TYPE=BlastDocs&DOC_TYPE=CloudBlast

Some useful parameters:

- program blastn, blastp, blastx, tblastn, or tblastx (lower case)
- database Which database to search against (e.g. "nr").
- sequence The sequence to search.
- ncbi_gi TRUE/FALSE whether to give 'gi' identifier.
- descriptions Number of descriptions to show. Def 500.
- alignments Number of alignments to show. Def 500.
- expect An expect value cutoff. Def 10.0.
- matrix_name Specify an alt. matrix (PAM30, PAM70, BLOSUM80, BLOSUM45).
- filter "none" turns off filtering. Default no filtering
- format_type "HTML", "Text", "ASN.1", or "XML". Def. "XML".
- entrez_query Entrez query to limit Blast search
- hitlist_size Number of hits to return. Default 50
- megablast TRUE/FALSE whether to use MEga BLAST algorithm (blastn only)
- **short_query TRUE/FALSE whether to adjust the search parameters for a**
  short query sequence. Note that this will override manually set parameters like word size and e value. Turns off when sequence length is > 30 residues. Default: None.
- service plain, psi, phi, rpsblast, megablast (lower case)

This function does no checking of the validity of the parameters and passes the values to the server as is. More help is available at: https://ncbi.github.io/blast-cloud/dev/api.html

Bio.Blast.NCBIXML module

Code to work with the BLAST XML output.

The BLAST XML DTD file is available on the NCBI site at: https://www.ncbi.nlm.nih.gov/dtd/NCBI_BlastOutput.dtd

Record classes to hold BLAST output are:

Classes: Blast Holds all the information from a blast search. PSIBlast Holds all the information from a psi-blast search.

Header Holds information from the header. Description Holds information about one hit description. Alignment Holds information about one alignment hit. HSP Holds information about one HSP. MultipleAlignment Holds information about a multiple alignment. DatabaseReport Holds information from the database report. Parameters Holds information from the parameters.

Bio.Blast.NCBIXML.**fmt_**(*value*, *format_spec='%s'*, *default_str='<unknown>'*)

Ensure the given value formats to a string correctly.

class Bio.Blast.NCBIXML.**Header**

Bases: object

Saves information from a blast header.

Members: application The name of the BLAST flavor that generated this data. version Version of blast used. date Date this data was generated. reference Reference for blast.

query Name of query sequence. query_letters Number of letters in the query sequence. (int)

database Name of the database. database_sequences Number of sequences in the database. (int) database_letters Number of letters in the database. (int)

__init__()

Initialize the class.

class Bio.Blast.NCBIXML.**Description**

Bases: object

Stores information about one hit in the descriptions section.

Members: title Title of the hit. score Number of bits. (int) bits Bit score. (float) e E value. (float) num_alignments Number of alignments for the same subject. (int)

__init__()

Initialize the class.

__str__()

Return the description as a string.

class Bio.Blast.NCBIXML.**DescriptionExt**

Bases: *Description*

Extended description record for BLASTXML version 2.

Members: items List of DescriptionExtItem

__init__()

Initialize the class.

append_item(*item*)

Add a description extended record.

__annotations__ = {}

class Bio.Blast.NCBIXML.**DescriptionExtItem**

Bases: object

Stores information about one record in hit description for BLASTXML version 2.

Members: id Database identifier title Title of the hit.

__init__()

Initialize the class.

__str__()

Return the description identifier and title as a string.

class Bio.Blast.NCBIXML.**Alignment**

Bases: object

Stores information about one hit in the alignments section.

Members: title Name. hit_id Hit identifier. (str) hit_def Hit definition. (str) length Length. (int) hsps A list of HSP objects.

__init__()

Initialize the class.

__str__()

Return the BLAST alignment as a formatted string.

class Bio.Blast.NCBIXML.**HSP**

Bases: object

Stores information about one hsp in an alignment hit.

Members:

- score BLAST score of hit. (float)
- bits Number of bits for that score. (float)
- expect Expect value. (float)
- num_alignments Number of alignments for same subject. (int)
- identities Number of identities (int) if using the XML parser. Tuple of number of identities/total aligned (int, int) if using the (obsolete) plain text parser.
- positives Number of positives (int) if using the XML parser. Tuple of number of positives/total aligned (int, int) if using the (obsolete) plain text parser.
- gaps Number of gaps (int) if using the XML parser. Tuple of number of gaps/total aligned (int, int) if using the (obsolete) plain text parser.
- align_length Length of the alignment. (int)
- strand Tuple of (query, target) strand.
- frame Tuple of 1 or 2 frame shifts, depending on the flavor.
- query The query sequence.
- query_start The start residue for the query sequence. (1-based)
- query_end The end residue for the query sequence. (1-based)
- match The match sequence.
- sbjct The sbjct sequence.

- sbjct_start The start residue for the sbjct sequence. (1-based)
- sbjct_end The end residue for the sbjct sequence. (1-based)

Not all flavors of BLAST return values for every attribute:

	score	expect	identities	positives	strand	frame
BLASTP	X	X	X	X		
BLASTN	X	X	X	X	X	
BLASTX	X	X	X	X		X
TBLASTN	X	X	X	X		X
TBLASTX	X	X	X	X		X/X

Note: for BLASTX, the query sequence is shown as a protein sequence, but the numbering is based on the nucleotides. Thus, the numbering is 3x larger than the number of amino acid residues. A similar effect can be seen for the sbjct sequence in TBLASTN, and for both sequences in TBLASTX.

Also, for negative frames, the sequence numbering starts from query_start and counts down.

__init__()

Initialize the class.

__str__()

Return the BLAST HSP as a formatted string.

class Bio.Blast.NCBIXML.**MultipleAlignment**

Bases: object

Holds information about a multiple alignment.

Members: alignment A list of tuples (name, start residue, sequence, end residue).

The start residue is 1-based. It may be blank, if that sequence is not aligned in the multiple alignment.

__init__()

Initialize the class.

to_generic()

Retrieve generic alignment object for the given alignment.

Instead of the tuples, this returns a MultipleSeqAlignment object from Bio.Align, through which you can manipulate and query the object.

Thanks to James Casbon for the code.

class Bio.Blast.NCBIXML.**Round**

Bases: object

Holds information from a PSI-BLAST round.

Members: number Round number. (int) reused_seqs Sequences in model, found again. List of Description objects. new_seqs Sequences not found, or below threshold. List of Description. alignments A list of Alignment objects. multiple_alignment A MultipleAlignment object.

__init__()

Initialize the class.

class Bio.Blast.NCBIXML.**DatabaseReport**

Bases: object

Holds information about a database report.

Members: database_name List of database names. (can have multiple dbs) num_letters_in_database Number of letters in the database. (int) num_sequences_in_database List of number of sequences in the database. posted_date List of the dates the databases were posted. ka_params A tuple of (lambda, k, h) values. (floats) gapped # XXX this isn't set right! ka_params_gap A tuple of (lambda, k, h) values. (floats)

__init__()

Initialize the class.

class Bio.Blast.NCBIXML.**Parameters**

Bases: object

Holds information about the parameters.

Members: matrix Name of the matrix. gap_penalties Tuple of (open, extend) penalties. (floats) sc_match Match score for nucleotide-nucleotide comparison sc_mismatch Mismatch penalty for nucleotide-nucleotide comparison num_hits Number of hits to the database. (int) num_sequences Number of sequences. (int) num_good_extends Number of extensions. (int) num_seqs_better_e Number of sequences better than e-value. (int) hsps_no_gap Number of HSP's better, without gapping. (int) hsps_prelim_gapped Number of HSP's gapped in prelim test. (int) hsps_prelim_gapped_attemped Number of HSP's attempted in prelim. (int) hsps_gapped Total number of HSP's gapped. (int) query_length Length of the query. (int) query_id Identifier of the query sequence. (str) database_length Number of letters in the database. (int) effective_hsp_length Effective HSP length. (int) effective_query_length Effective length of query. (int) effective_database_length Effective length of database. (int) effective_search_space Effective search space. (int) effective_search_space_used Effective search space used. (int) frameshift Frameshift window. Tuple of (int, float) threshold Threshold. (int) window_size Window size. (int) dropoff_1st_pass Tuple of (score, bits). (int, float) gap_x_dropoff Tuple of (score, bits). (int, float) gap_x_dropoff_final Tuple of (score, bits). (int, float) gap_trigger Tuple of (score, bits). (int, float) blast_cutoff Tuple of (score, bits). (int, float)

__init__()

Initialize the class.

class Bio.Blast.NCBIXML.**Blast**

Bases: *Header*, *DatabaseReport*, *Parameters*

Saves the results from a blast search.

Members: descriptions A list of Description objects. alignments A list of Alignment objects. multiple_alignment A MultipleAlignment object. + members inherited from base classes

__init__()

Initialize the class.

__annotations__ = {}

class Bio.Blast.NCBIXML.**PSIBlast**

Bases: *Header*, *DatabaseReport*, *Parameters*

Saves the results from a blastpgp search.

Members: rounds A list of Round objects. converged Whether the search converged. + members inherited from base classes

__init__()

Initialize the class.

__annotations__ = {}

class Bio.Blast.NCBIXML.**BlastParser**(*debug=0*)

Bases: _XMLparser

Parse XML BLAST data into a Blast object.

Parses XML output from BLAST (direct use discouraged). This (now) returns a list of Blast records. Historically it returned a single Blast record. You are expected to use this via the parse or read functions.

All XML 'action' methods are private methods and may be:

- _start_TAG called when the start tag is found
- _end_TAG called when the end tag is found

__init__(*debug=0*)

Initialize the parser.

Arguments:

- debug - integer, amount of debug information to print

reset()

Reset all the data allowing reuse of the BlastParser() object.

set_hit_id()

Record the identifier of the database sequence (PRIVATE).

set_hit_def()

Record the definition line of the database sequence (PRIVATE).

set_hit_accession()

Record the accession value of the database sequence (PRIVATE).

set_hit_len()

Record the length of the hit.

Bio.Blast.NCBIXML.**read**(*handle*, *debug=0*)

Return a single Blast record (assumes just one query).

Uses the BlastParser internally.

This function is for use when there is one and only one BLAST result in your XML file.

Use the Bio.Blast.NCBIXML.parse() function if you expect more than one BLAST record (i.e. if you have more than one query sequence).

Bio.Blast.NCBIXML.**parse**(*handle*, *debug=0*)

Return an iterator a Blast record for each query.

Incremental parser, this is an iterator that returns Blast records. It uses the BlastParser internally.

handle - file handle to and XML file to parse debug - integer, amount of debug information to print

This is a generator function that returns multiple Blast records objects - one for each query sequence given to blast. The file is read incrementally, returning complete records as they are read in.

Should cope with new BLAST 2.2.14+ which gives a single XML file for multiple query records.

Should also cope with XML output from older versions BLAST which gave multiple XML files concatenated together (giving a single file which strictly speaking wasn't valid XML).

Module contents

Code to parse and store BLAST XML output, and to invoke the NCBI BLAST web server.

This module provides code to parse and store BLAST XML output, following its definition in the associated BLAST XML DTD file: https://www.ncbi.nlm.nih.gov/dtd/NCBI_BlastOutput.dtd

This module also provides code to invoke the BLAST web server provided by NCBI. https://blast.ncbi.nlm.nih.gov/

Variables:

- email Set the Blast email parameter (default is None).
- tool Set the Blast tool parameter (default is `biopython`).

exception `Bio.Blast.`**NotXMLError**(*message*)

Bases: `ValueError`

Failed to parse file as XML.

__init__(*message*)

Initialize the class.

__str__()

Return a string summary of the exception.

exception `Bio.Blast.`**CorruptedXMLError**(*message*)

Bases: `ValueError`

Corrupted XML.

__init__(*message*)

Initialize the class.

__str__()

Return a string summary of the exception.

class `Bio.Blast.`**HSP**(*sequences*, *coordinates=None*)

Bases: *Alignment*

Stores an alignment of one query sequence against a target sequence.

An HSP (High-scoring Segment Pair) stores the alignment of one query sequence segment against one target (hit) sequence segment. The `Bio.Blast.HSP` class inherits from the `Bio.Align.Alignment` class.

In addition to the `target` and `query` attributes of a `Bio.Align.Alignment`, a `Bio.Blast.HSP` object has the following attributes:

- score: score of HSP;
- **annotations: a dictionary that may contain the following keys:**
  - 'bit score': score (in bits) of HSP (float);
  - 'evalue': e-value of HSP (float);
  - 'identity': number of identities in HSP (integer);
  - 'positive': number of positives in HSP (integer);
  - 'gaps': number of gaps in HSP (integer);
  - 'midline': formatting middle line.

A `Bio.Blast.HSP` object behaves the same as a *Bio.Align.Alignment* ` object and can be used as such. However, when printing a `Bio.Blast.HSP` object, the BLAST e-value and bit score are included in the output (in addition to the alignment itself).

See the documentation of `Bio.Blast.Record` for a more detailed explanation of how the information in BLAST records is stored in Biopython.

__repr__()

Return a representation of the alignment, including its shape.

The representation cannot be used with eval() to recreate the object, which is usually possible with simple python objects. For example:

<Alignment object (2 rows x 14 columns) at 0x10403d850>

The hex string is the memory address of the object and can be used to distinguish different Alignment objects. See help(id) for more information.

```
>>> import numpy as np
>>> from Bio.Align import Alignment
>>> alignment = Alignment(("ACCGT", "ACGT"),
...                       coordinates = np.array([[0, 2, 3, 5],
...                                               [0, 2, 2, 4],
...                                              ]))
>>> print(alignment)
target            0 ACCGT 5
                  0 ||-|| 5
query             0 AC-GT 4

>>> alignment
<Alignment object (2 rows x 5 columns) at 0x...>
```

__str__()

Return a human-readable string representation of the alignment.

For sequence alignments, each line has at most 80 columns. The first 10 columns show the (possibly truncated) sequence name, which may be the id attribute of a SeqRecord, or otherwise 'target' or 'query' for pairwise alignments. The next 10 columns show the sequence coordinate, using zero-based counting as usual in Python. The remaining 60 columns shown the sequence, using dashes to represent gaps. At the end of the alignment, the end coordinates are shown on the right of the sequence, again in zero-based coordinates.

Pairwise alignments have an additional line between the two sequences showing whether the sequences match ('|') or mismatch ('.'), or if there is a gap ('-'). The coordinates shown for this line are the column indices, which can be useful when extracting a subalignment.

For example,

```
>>> from Bio.Align import PairwiseAligner
>>> aligner = PairwiseAligner()
```

```
>>> seqA = "TTAACCCCATTTG"
>>> seqB = "AAGCCCCTTT"
>>> seqC = "AAAGGGGCTT"
```

```
>>> alignments = aligner.align(seqA, seqB)
>>> len(alignments)
```

(continues on next page)

(continued from previous page)

```
1
>>> alignment = alignments[0]
>>> print(alignment)
target            0 TTAA-CCCCATTTG 13
                  0 --||-||||-|||- 14
query             0 --AAGCCCC-TTT- 10
```

Note that seqC is the reverse complement of seqB. Aligning it to the reverse strand gives the same alignment, but the query coordinates are switched:

```
>>> alignments = aligner.align(seqA, seqC, strand="-")
>>> len(alignments)
1
>>> alignment = alignments[0]
>>> print(alignment)
target            0 TTAA-CCCCATTTG 13
                  0 --||-||||-|||- 14
query            10 --AAGCCCC-TTT-  0
```

__annotations__ = {}

class Bio.Blast.**Hit**(*alignments=()*)

Bases: *Alignments*

Stores a single BLAST hit of one single query against one target.

The `Bio.Blast.Hit` class inherits from the `Bio.Align.Alignments` class, which is a subclass of a Python list. The `Bio.Blast.Hit` class stores `Bio.Blast.HSP` objwcts, which inherit from `Bio.Align.Alignment`. A `Bio.Blast.Hit` object is therefore effectively a list of `Bio.Align.Alignment` objects. Most hits consist of only 1 or a few Alignment objects.

Each `Bio.Blast.Hit` object has a `target` attribute containing the following information:

- target.id: seqId of subject;
- target.description: definition line of subject;
- target.name: accession of subject;
- len(target.seq): sequence length of subject.

See the documentation of `Bio.Blast.Record` for a more detailed explanation of the information stored in the alignments contained in the `Bio.Blast.Hit` object.

__getitem__(*key*)

x.__getitem__(y) <==> x[y]

__repr__()

Return repr(self).

__str__()

Return a human readable summary of the Hit object.

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.Blast.Record

Bases: list

Stores the BLAST results for a single query.

A Bio.Blast.Record object is a list of Bio.Blast.Hit objects, each corresponding to one hit for the query in the BLAST output.

The Bio.Blast.Record object may have the following attributes:

- **query: A SeqRecord object which may contain some or all of the following information:**
  - query.id: SeqId of query;
  - query.description: Definition line of query;
  - len(query.seq): Length of the query sequence.
- **stat: A dictionary with summary statistics of the BLAST run. It may contain the following keys:**
  - 'db-num': number of sequences in BLAST db (integer);
  - 'db-len': length of BLAST db (integer);
  - 'hsp-len': effective HSP length (integer);
  - 'eff-space': effective search space (float);
  - 'kappa': Karlin-Altschul parameter K (float);
  - 'lambda': Karlin-Altschul parameter Lambda (float);
  - 'entropy': Karlin-Altschul parameter H (float).
- message: Some (error?) information.

Each Bio.Blast.Hit object has a target attribute containing the following information:

- target.id: seqId of subject;
- target.description: definition line of subject;
- target.name: accession of subject;
- len(target.seq): sequence length of subject.

The Bio.Blast.Hit class inherits from the Bio.Align.Alignments class, which inherits from a Python list. In this list, the Bio.Blast.Hit object stores Bio.Blast.HSP objects, which inherit from the Bio.Align.Alignment class. A Bio.Blast.Hit object is therefore effectively a list of alignment objects.

Each HSP in a Bio.Blast.Hit object has the attributes target and query attributes, as usual for of a Bio.Align.Alignment object storing a pairwise alignment, pointing to a SeqRecord object representing the target and query, respectively. For translated BLAST searches, the features attribute of the target or query may contain a SeqFeature of type CDS that stores the amino acid sequence region. The qualifiers attribute of such a feature is a dictionary with a single key 'coded_by'; the corresponding value specifies the nucleotide sequence region, in a GenBank-style string with 1-based coordinates, that encodes the amino acid sequence.

Each Bio.Blast.HSP object has the following additional attributes:

- score: score of HSP;
- **annotations: a dictionary that may contain the following keys:**
  - 'bit score': score (in bits) of HSP (float);

- 'evalue': e-value of HSP (float);
- 'identity': number of identities in HSP (integer);
- 'positive': number of positives in HSP (integer);
- 'gaps': number of gaps in HSP (integer);
- 'midline': formatting middle line.

```
>>> from Bio import Blast
>>> record = Blast.read("Blast/xml_2212L_blastx_001.xml")
>>> record.query
SeqRecord(seq=Seq(None, length=556), id='gi|1347369|gb|G25137.1|G25137', name='
↪<unknown name>', description='human STS EST48004, sequence tagged site',␣
↪dbxrefs=[])
>>> record.stat
{'db-num': 2934173, 'db-len': 1011751523, 'hsp-len': 0, 'eff-space': 0, 'kappa': 0.
↪041, 'lambda': 0.267, 'entropy': 0.14}
>>> len(record)
78
>>> hit = record[0]
>>> type(hit)
<class 'Bio.Blast.Hit'>
>>> from Bio.Align import Alignments
>>> isinstance(hit, Alignments)
True
>>> hit.target
SeqRecord(seq=Seq(None, length=319), id='gi|12654095|gb|AAH00859.1|', name='AAH00859
↪', description='Unknown (protein for IMAGE:3459481) [Homo sapiens]', dbxrefs=[])
```

Most hits consist of only 1 or a few Alignment objects:

```
>>> len(hit)
1
>>> alignment = hit[0]
>>> type(alignment)
<class 'Bio.Blast.HSP'>
>>> alignment.score
630.0
>>> alignment.annotations
{'bit score': 247.284, 'evalue': 1.69599e-64, 'identity': 122, 'positive': 123,
↪'gaps': 0, 'midline':
↪'DLQLLIKAVNLFPAGTNSRWEVIANYMNIHSSSGVKRTAKDVIGKAKSLQKLDPHQKDDINKKAFDKFKKEHGVVPQADNATPSERF␣
↪GPYTDFTP TTE QKL EQAL TYPVNT ERW  IA AVPGR K+'}
```

Target and query information are stored in the respective attributes of the alignment:

```
>>> alignment.target
SeqRecord(seq=Seq({155: 'DLQLLIKAVNLFPAGTNSRWEVIANYMNIHSSSGVKRTAKDVIGKAKSLQKLDP...
↪TKK'}, length=319), id='gi|12654095|gb|AAH00859.1|', name='AAH00859', description=
↪'Unknown (protein for IMAGE:3459481) [Homo sapiens]', dbxrefs=[])
>>> alignment.query
SeqRecord(seq=Seq('DLQLLIKAVNLFPAGTNSRWEVIANYMNIHSSSGVKRTAKDVIGKAKSLQKLDP...XKE'),␣
↪id='gi|1347369|gb|G25137.1|G25137', name='<unknown name>', description='human STS␣
↪EST48004, sequence tagged site', dbxrefs=[])
```

This was a BLASTX run, so the query sequence was translated:

```
>>> len(alignment.target.features)
0
>>> len(alignment.query.features)
1
>>> feature = alignment.query.features[0]
>>> feature
SeqFeature(SimpleLocation(ExactPosition(0), ExactPosition(133)), type='CDS',␣
↪qualifiers=...)
>>> feature.qualifiers
{'coded_by': 'gi|1347369|gb|G25137.1|G25137:1..399'}
```

i.e., nucleotides 0:399 (in zero-based coordinates) encode the amino acids of the query in the alignment.

For an alignment against the reverse strand, the location in the qualifier is shown as in this example:

```
>>> record[72][0].query.features[0].qualifiers
{'coded_by': 'complement(gi|1347369|gb|G25137.1|G25137:345..530)'}
```

__init__()
Initialize the Record object.

__repr__()
Return repr(self).

__str__()
Return str(self).

__getitem__(*key*)
x.__getitem__(y) <==> x[y]

keys()
Return a list of the target.id of each hit.

__contains__(*key*)
Return key in self.

index(*key*)
Return the index of the hit for which the target.id is equal to the key.

class Bio.Blast.**Records**(*source*)
Bases: UserList

Stores the BLAST results of a single BLAST run.

A Bio.Blast.Records object is an iterator. Iterating over it returns returns Bio.Blast.Record objects, each of which corresponds to one BLAST query.

Common attributes of a Bio.Blast.Records object are

- **source: The input data from which the Bio.Blast.Records object** was constructed.
- program: The specific BLAST program that was used (e.g., 'blastn').
- version: The version of the BLAST program (e.g., 'BLASTN 2.2.27+').
- reference: The literature reference to the BLAST publication.

- **db: The BLAST database against which the query was run**
  (e.g., 'nr').
- **query: A SeqRecord object which may contain some or all of the**

  following information:

  - query.id: SeqId of the query;
  - query.description: Definition line of the query;
  - **query.seq: The query sequence. The query sequence.**
    The query sequence.
- **param: A dictionary with the parameters used for the BLAST run.**

  You may find the following keys in this dictionary:

  - **'matrix': the scoring matrix used in the BLAST run**
    (e.g., 'BLOSUM62') (string);
  - **'expect': threshold on the expected number of chance**
    matches (float);
  - **'include': e-value threshold for inclusion in**
    multipass model in psiblast (float);
  - 'sc-match': score for matching nucleotides (integer);
  - **'sc-mismatch': score for mismatched nucleotides**
    (integer);
  - 'gap-open': gap opening cost (integer);
  - 'gap-extend': gap extension cost (integer);
  - **'filter': filtering options applied in the BLAST**
    run (string);
  - 'pattern': PHI-BLAST pattern (string);
  - 'entrez-query': Limit of request to Entrez query (string).
- **mbstat: A dictionary with Mega BLAST search statistics. As this**
  information is stored near the end of the XML file, this attribute can only be accessed after the file has been read completely (by iterating over the records until a `StopIteration` is issued. This dictionary can contain the same keys as the dictionary stored under the `stat` attribute of a `Record` object.

```
>>> from Bio import Blast
>>> path = "Blast/xml_2218_blastp_002.xml"
```

In a script, you would use a `with` block, as in

```
>>> with Blast.parse(path) as records:
...     print(records.source)
...
Blast/xml_2218_blastp_002.xml
```

to ensure that the file is closed at the end of the block. Here, we will simply do

```
>>> records = Blast.parse("Blast/xml_2218_blastp_002.xml")
```

so we can see the output of each command right away.

```
>>> type(records)
<class 'Bio.Blast.Records'>
>>> records.source
'Blast/xml_2218_blastp_002.xml'
>>> records.program
'blastp'
>>> records.version
'BLASTP 2.2.18+'
>>> records.reference
'Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang,␣
↪Zheng Zhang, Webb Miller, and David J. Lipman (1997), "Gapped BLAST and PSI-
↪BLAST: a new generation of protein database search programs", Nucleic Acids Res.␣
↪25:3389-3402.'
>>> records.db
'gpipe/9606/Previous/protein'
>>> records.param
{'matrix': 'BLOSUM62', 'expect': 0.01, 'gap-open': 11, 'gap-extend': 1, 'filter':
↪'m L; R -d repeat/repeat_9606;'}
```

Iterating over the records returns Bio.Blast.Record objects:

```
>>> record = next(records)
>>> type(record)
<class 'Bio.Blast.Record'>
>>> record.query.id
'gi|585505|sp|Q08386|MOPB_RHOCA'
>>> record = next(records)
>>> type(record)
<class 'Bio.Blast.Record'>
>>> record.query.id
'gi|129628|sp|P07175.1|PARA_AGRTU'
>>> record = next(records)
Traceback (most recent call last):
...
StopIteration
```

You can also use the records as a list, for example by extracting a record by index, or by calling `len` or `print` on the records. The parser will then automatically iterate over the records and store them:

```
>>> records = Blast.parse("Blast/wnts.xml")
>>> record = records[3]  # this causes all records to be read in and stored
>>> record.query.id
'Query_4'
>>> len(records)
5
```

After the records have been read in, you can still iterate over them:

```
>>> for i, record in enumerate(records):
...     print(i, record.query.id)
...
0 Query_1
1 Query_2
```

(continues on next page)

(continued from previous page)

```
2 Query_3
3 Query_4
4 Query_5
```

__init__(*source*)

Initialize the Records object.

__enter__()

__exit__(*exc_type*, *exc_value*, *exc_traceback*)

__iter__()

__next__()

__getitem__(*index*)

property data

Overrides the data attribute of UserList.

__repr__()

Return repr(self).

__str__()

Return str(self).

__abstractmethods__ = frozenset({})

Bio.Blast.**parse**(*source*)

Parse an XML file containing BLAST output and return a Bio.Blast.Records object.

This returns an iterator object; iterating over it returns Bio.Blast.Record objects one by one.

The source can be a file stream or the path to an XML file containing the BLAST output. If a file stream, source must be in binary mode. This allows the parser to detect the encoding from the XML file,and to use it to convert any text in the XML to the correct Unicode string. The qblast function in Bio.Blast returns a file stream in binary mode. For files, please use mode "rb" when opening the file, as in

```
>>> from Bio import Blast
>>> stream = open("Blast/wnts.xml", "rb")  # opened in binary mode
>>> records = Blast.parse(stream)
>>> for record in records:
...     print(record.query.id, record.query.description)
...
Query_1 gi|195230749:301-1383 Homo sapiens wingless-type MMTV integration site␣
↪family member 2 (WNT2), transcript variant 1, mRNA
Query_2 gi|325053704:108-1166 Homo sapiens wingless-type MMTV integration site␣
↪family, member 3A (WNT3A), mRNA
Query_3 gi|156630997:105-1160 Homo sapiens wingless-type MMTV integration site␣
↪family, member 4 (WNT4), mRNA
Query_4 gi|371502086:108-1205 Homo sapiens wingless-type MMTV integration site␣
↪family, member 5A (WNT5A), transcript variant 2, mRNA
Query_5 gi|53729353:216-1313 Homo sapiens wingless-type MMTV integration site␣
↪family, member 6 (WNT6), mRNA
>>> stream.close()
```

Bio.Blast.**read**(*source*)

Parse an XML file containing BLAST output for a single query and return it.

Internally, this function uses Bio.Blast.parse to obtain an iterator over BLAST records. The function then reads one record from the iterator, ensures that there are no further records, and returns the record it found as a Bio.Blast.Record object. An exception is raised if no records are found, or more than one record is found.

The source can be a file stream or the path to an XML file containing the BLAST output. If a file stream, source must be in binary mode. This allows the parser to detect the encoding from the XML file,and to use it to convert any text in the XML to the correct Unicode string. The qblast function in Bio.Blast returns a file stream in binary mode. For files, please use mode "rb" when opening the file, as in

```
>>> from Bio import Blast
>>> stream = open("Blast/xml_21500_blastn_001.xml", "rb")  # opened in binary mode
>>> record = Blast.read(stream)
>>> record.query.id
'Query_78041'
>>> record.query.description
'G26684.1 human STS STS_D11570, sequence tagged site'
>>> len(record)
11
>>> stream.close()
```

Use the Bio.Blast.parse function if you want to read a file containing BLAST output for more than one query.

Bio.Blast.**write**(*records*, *destination*, *fmt='XML'*)

Write BLAST records as an XML file, and return the number of records.

Arguments:

- records - A `Bio.Blast.Records` object.
- **destination - File or file-like object to write to, or filename as** string. The File object must have been opened for writing in binary mode, and must be closed (or flushed) by the caller after this function returns to ensure that all records are written.
- **fmt - string describing the file format to write** (case-insensitive). Currently, only "XML" and "XML2" are accepted.

Returns the number of records written (as an integer).

Bio.Blast.**qblast**(*program*, *database*, *sequence*, *url_base=NCBI_BLAST_URL*, *auto_format=None*, *composition_based_statistics=None*, *db_genetic_code=None*, *endpoints=None*, *entrez_query='(none)'*, *expect=10.0*, *filter=None*, *gapcosts=None*, *genetic_code=None*, *hitlist_size=50*, *i_thresh=None*, *layout=None*, *lcase_mask=None*, *matrix_name=None*, *nucl_penalty=None*, *nucl_reward=None*, *other_advanced=None*, *perc_ident=None*, *phi_pattern=None*, *query_file=None*, *query_believe_defline=None*, *query_from=None*, *query_to=None*, *searchsp_eff=None*, *service=None*, *threshold=None*, *ungapped_alignment=None*, *word_size=None*, *short_query=None*, *alignments=500*, *alignment_view=None*, *descriptions=500*, *entrez_links_new_window=None*, *expect_low=None*, *expect_high=None*, *format_entrez_query=None*, *format_object=None*, *format_type='XML'*, *ncbi_gi=None*, *results_file=None*, *show_overview=None*, *megablast=None*, *template_type=None*, *template_length=None*, *username='blast'*, *password=None*)

BLAST search using NCBI's QBLAST server or a cloud service provider.

Supports all parameters of the old qblast API for Put and Get.

Please note that NCBI uses the new Common URL API for BLAST searches on the internet (http://ncbi.github.io/blast-cloud/dev/api.html). Thus, some of the parameters used by this function are not (or are no longer) officially supported by NCBI. Although they are still functioning, this may change in the future.

The Common URL API (http://ncbi.github.io/blast-cloud/dev/api.html) allows doing BLAST searches on cloud servers. To use this feature, please set `url_base='http://host.my.cloud.service.provider.com/cgi-bin/blast.cgi'` and `format_object='Alignment'`. For more details, please see https://blast.ncbi.nlm.nih.gov/Blast.cgi?PAGE_TYPE=BlastDocs&DOC_TYPE=CloudBlast

Some useful parameters:

- program blastn, blastp, blastx, tblastn, or tblastx (lower case)
- database Which database to search against (e.g. "nr").
- sequence The sequence to search.
- ncbi_gi TRUE/FALSE whether to give 'gi' identifier.
- descriptions Number of descriptions to show. Def 500.
- alignments Number of alignments to show. Def 500.
- expect An expect value cutoff. Def 10.0.
- matrix_name Specify an alt. matrix (PAM30, PAM70, BLOSUM80, BLOSUM45).
- filter "none" turns off filtering. Default no filtering
- **format_type "XML" (default), "HTML", "Text", "XML2", "JSON2",** or "Tabular".
- entrez_query Entrez query to limit Blast search
- hitlist_size Number of hits to return. Default 50
- megablast TRUE/FALSE whether to use MEga BLAST algorithm (blastn only)
- **short_query TRUE/FALSE whether to adjust the search parameters for a** short query sequence. Note that this will override manually set parameters like word size and e value. Turns off when sequence length is > 30 residues. Default: None.
- service plain, psi, phi, rpsblast, megablast (lower case)

This function does no checking of the validity of the parameters and passes the values to the server as is. More help is available at: https://ncbi.github.io/blast-cloud/dev/api.html

The http.client.HTTPResponse object returned by this function has the additional attributes rid and rtoe with the Request ID and Request Time Of Execution for this BLAST search.

28.1.6 Bio.CAPS package

Module contents

Cleaved amplified polymorphic sequence (CAPS) markers.

A CAPS marker is a location a DifferentialCutsite as described below and a set of primers that can be used to visualize this. More information can be found in the paper Konieczny and Ausubel (1993) (PMID 8106085).

class Bio.CAPS.**DifferentialCutsite**(***kwds*)

Bases: `object`

Differential enzyme cutsite in an alignment.

A differential cutsite is a location in an alignment where an enzyme cuts at least one sequence and also cannot cut at least one other sequence.

Members:

- start - Where it lives in the alignment.
- enzyme - The enzyme that causes this.
- cuts_in - A list of sequences (as indexes into the alignment) the enzyme cuts in.
- blocked_in - A list of sequences (as indexes into the alignment) the enzyme is blocked in.

`__init__`(***kwds*)

Initialize a DifferentialCutsite.

Each member (as listed in the class description) should be included as a keyword.

exception Bio.CAPS.**AlignmentHasDifferentLengthsError**

Bases: `Exception`

Exception where sequences in alignment have different lengths.

class Bio.CAPS.**CAPSMap**(*alignment*, *enzymes=None*)

Bases: `object`

A map of an alignment showing all possible dcuts.

Members:

- alignment - The alignment that is mapped.
- dcuts - A list of possible CAPS markers in the form of DifferentialCutsites.

`__init__`(*alignment*, *enzymes=None*)

Initialize the CAPSMap.

Required:

- alignment - The alignment to be mapped.

Optional:

- enzymes - List of enzymes to be used to create the map. Defaults to an empty list.

28.1.7 Bio.Cluster package

Module contents

Cluster Analysis.

The Bio.Cluster provides commonly used clustering algorithms and was designed with the application to gene expression data in mind. However, this module can also be used for cluster analysis of other types of data.

Bio.Cluster and the underlying C Clustering Library is described in M. de Hoon et al. (2004) https://doi.org/10.1093/bioinformatics/bth078

class Bio.Cluster.**Node**

Bases: `Node`

A Node object describes a single node in a hierarchical clustering tree. The integer attributes 'left' and 'right' represent the two members that make up this node; the floating point attribute 'distance' contains the distance between the two members of this node.

class Bio.Cluster.**Tree**

Bases: Tree

Hierarchical clustering tree.

A Tree consists of Nodes.

sort(*order=None*)

Sort the hierarchical clustering tree.

Sort the hierarchical clustering tree by switching the left and right subnode of nodes such that the elements in the left-to-right order of the tree tend to have increasing order values.

Return the indices of the elements in the left-to-right order in the hierarchical clustering tree, such that the element with index indices[i] occurs at position i in the dendrogram.

cut(*nclusters=None*)

Create clusters by cutting the hierarchical clustering tree.

Divide the elements in a hierarchical clustering result mytree into clusters, and return an array with the number of the cluster to which each element was assigned.

Keyword arguments:

- nclusters: The desired number of clusters.

Bio.Cluster.**kcluster**(*data*, *nclusters=2*, *mask=None*, *weight=None*, *transpose=False*, *npass=1*, *method='a'*, *dist='e'*, *initialid=None*)

Perform k-means clustering.

This function performs k-means clustering on the values in data, and returns the cluster assignments, the within-cluster sum of distances of the optimal k-means clustering solution, and the number of times the optimal solution was found.

Keyword arguments:

- data: nrows x ncolumns array containing the data values.
- nclusters: number of clusters (the 'k' in k-means).
- mask: nrows x ncolumns array of integers, showing which data are missing. If mask[i,j]==0, then data[i,j] is missing.
- weight: the weights to be used when calculating distances
- transpose: - if False: rows are clustered; - if True: columns are clustered.
- npass: number of times the k-means clustering algorithm is performed, each time with a different (random) initial condition.
- method: specifies how the center of a cluster is found: - method == 'a': arithmetic mean; - method == 'm': median.
- dist: specifies the distance function to be used: - dist == 'e': Euclidean distance; - dist == 'b': City Block distance; - dist == 'c': Pearson correlation; - dist == 'a': absolute value of the correlation; - dist == 'u': uncentered correlation; - dist == 'x': absolute uncentered correlation; - dist == 's': Spearman's rank correlation; - dist == 'k': Kendall's tau.
- initialid: the initial clustering from which the algorithm should start. If initialid is None, the routine carries out npass repetitions of the EM algorithm, each time starting from a different random initial clustering. If initialid is given, the routine carries out the EM algorithm only once, starting from the given initial clustering and without randomizing the order in which items are assigned to clusters (i.e., using the same order as in the data matrix). In that case, the k-means algorithm is fully deterministic.

Return values:

- clusterid: array containing the index of the cluster to which each item was assigned in the best k-means clustering solution that was found in the npass runs;
- error: the within-cluster sum of distances for the returned k-means clustering solution;
- nfound: the number of times this solution was found.

`Bio.Cluster.`**kmedoids**(*distance*, *nclusters=2*, *npass=1*, *initialid=None*)

Perform k-medoids clustering.

This function performs k-medoids clustering, and returns the cluster assignments, the within-cluster sum of distances of the optimal k-medoids clustering solution, and the number of times the optimal solution was found.

Keyword arguments:

- distance: The distance matrix between the items. There are three ways in which you can pass a distance matrix: 1. a 2D NumPy array (in which only the left-lower part of the array will be accessed); 2. a 1D NumPy array containing the distances consecutively; 3. a list of rows containing the lower-triangular part of the distance matrix.

  Examples are:

```
>>> from numpy import array
>>> # option 1:
>>> distance = array([[0.0, 1.1, 2.3],
...                   [1.1, 0.0, 4.5],
...                   [2.3, 4.5, 0.0]])
>>> # option 2:
>>> distance = array([1.1, 2.3, 4.5])
>>> # option 3:
>>> distance = [array([]),
...             array([1.1]),
...             array([2.3, 4.5])]
```

  These three correspond to the same distance matrix.

- nclusters: number of clusters (the 'k' in k-medoids)
- npass: the number of times the k-medoids clustering algorithm is performed, each time with a different (random) initial condition.
- initialid: the initial clustering from which the algorithm should start. If initialid is not given, the routine carries out npass repetitions of the EM algorithm, each time starting from a different random initial clustering. If initialid is given, the routine carries out the EM algorithm only once, starting from the initial clustering specified by initialid and without randomizing the order in which items are assigned to clusters (i.e., using the same order as in the data matrix). In that case, the k-medoids algorithm is fully deterministic.

Return values:

- clusterid: array containing the index of the cluster to which each item was assigned in the best k-medoids clustering solution that was found in the npass runs; note that the index of a cluster is the index of the item that is the medoid of the cluster;
- error: the within-cluster sum of distances for the returned k-medoids clustering solution;
- nfound: the number of times this solution was found.

Bio.Cluster.**treecluster**(*data*, *mask=None*, *weight=None*, *transpose=False*, *method='m'*, *dist='e'*, *distancematrix=None*)

Perform hierarchical clustering, and return a Tree object.

This function implements the pairwise single, complete, centroid, and average linkage hierarchical clustering methods.

Keyword arguments:

- data: nrows x ncolumns array containing the data values.
- mask: nrows x ncolumns array of integers, showing which data are missing. If mask[i][j]==0, then data[i][j] is missing.
- weight: the weights to be used when calculating distances.
- transpose: - if False, rows are clustered; - if True, columns are clustered.
- dist: specifies the distance function to be used: - dist == 'e': Euclidean distance - dist == 'b': City Block distance - dist == 'c': Pearson correlation - dist == 'a': absolute value of the correlation - dist == 'u': uncentered correlation - dist == 'x': absolute uncentered correlation - dist == 's': Spearman's rank correlation - dist == 'k': Kendall's tau
- method: specifies which linkage method is used: - method == 's': Single pairwise linkage - method == 'm': Complete (maximum) pairwise linkage (default) - method == 'c': Centroid linkage - method == 'a': Average pairwise linkage
- distancematrix: The distance matrix between the items. There are three ways in which you can pass a distance matrix: 1. a 2D NumPy array (in which only the left-lower part of the array will be accessed); 2. a 1D NumPy array containing the distances consecutively; 3. a list of rows containing the lower-triangular part of the distance matrix.

  Examples are:

```
>>> from numpy import array
>>> # option 1:
>>> distance = array([[0.0, 1.1, 2.3],
...                   [1.1, 0.0, 4.5],
...                   [2.3, 4.5, 0.0]])
>>> # option 2:
>>> distance = array([1.1, 2.3, 4.5])
>>> # option 3:
>>> distance = [array([]),
...             array([1.1]),
...             array([2.3, 4.5])]
```

  These three correspond to the same distance matrix.

  PLEASE NOTE: As the treecluster routine may shuffle the values in the distance matrix as part of the clustering algorithm, be sure to save this array in a different variable before calling treecluster if you need it later.

Either data or distancematrix should be None. If distancematrix is None, the hierarchical clustering solution is calculated from the values stored in the argument data. If data is None, the hierarchical clustering solution is instead calculated from the distance matrix. Pairwise centroid-linkage clustering can be performed only from the data values and not from the distance matrix. Pairwise single-, maximum-, and average-linkage clustering can be calculated from the data values or from the distance matrix.

Return value: treecluster returns a Tree object describing the hierarchical clustering result. See the description of the Tree class for more information.

Bio.Cluster.**somcluster**(*data*, *mask=None*, *weight=None*, *transpose=False*, *nxgrid=2*, *nygrid=1*, *inittau=0.02*, *niter=1*, *dist='e'*)

Calculate a Self-Organizing Map.

This function implements a Self-Organizing Map on a rectangular grid.

Keyword arguments:

- data: nrows x ncolumns array containing the data values;
- mask: nrows x ncolumns array of integers, showing which data are missing. If mask[i][j]==0, then data[i][j] is missing.
- weight: the weights to be used when calculating distances
- transpose: - if False: rows are clustered; - if True: columns are clustered.
- nxgrid: the horizontal dimension of the rectangular SOM map
- nygrid: the vertical dimension of the rectangular SOM map
- inittau: the initial value of tau (the neighborbood function)
- niter: the number of iterations
- dist: specifies the distance function to be used: - dist == 'e': Euclidean distance - dist == 'b': City Block distance - dist == 'c': Pearson correlation - dist == 'a': absolute value of the correlation - dist == 'u': uncentered correlation - dist == 'x': absolute uncentered correlation - dist == 's': Spearman's rank correlation - dist == 'k': Kendall's tau

Return values:

- clusterid: array with two columns, with the number of rows equal to the items that are being clustered. Each row in the array contains the x and y coordinates of the cell in the rectangular SOM grid to which the item was assigned.
- celldata: an array with dimensions [nxgrid, nygrid, number of columns] if rows are being clustered, or [nxgrid, nygrid, number of rows) if columns are being clustered. Each element [ix, iy] of this array is a 1D vector containing the data values for the centroid of the cluster in the SOM grid cell with coordinates [ix, iy].

Bio.Cluster.**clusterdistance**(*data*, *mask=None*, *weight=None*, *index1=None*, *index2=None*, *method='a'*, *dist='e'*, *transpose=False*)

Calculate and return the distance between two clusters.

Keyword arguments:

- data: nrows x ncolumns array containing the data values.
- mask: nrows x ncolumns array of integers, showing which data are missing. If mask[i, j]==0, then data[i, j] is missing.
- weight: the weights to be used when calculating distances
- index1: 1D array identifying which items belong to the first cluster. If the cluster contains only one item, then index1 can also be written as a single integer.
- index2: 1D array identifying which items belong to the second cluster. If the cluster contains only one item, then index2 can also be written as a single integer.
- dist: specifies the distance function to be used: - dist == 'e': Euclidean distance - dist == 'b': City Block distance - dist == 'c': Pearson correlation - dist == 'a': absolute value of the correlation - dist == 'u': uncentered correlation - dist == 'x': absolute uncentered correlation - dist == 's': Spearman's rank correlation - dist == 'k': Kendall's tau

- method: specifies how the distance between two clusters is defined: - method == 'a': the distance between the arithmetic means of the two clusters - method == 'm': the distance between the medians of the two clusters - method == 's': the smallest pairwise distance between members of the two clusters - method == 'x': the largest pairwise distance between members of the two clusters - method == 'v': average of the pairwise distances between members of the two clusters
- transpose: - if False: clusters of rows are considered; - if True: clusters of columns are considered.

Bio.Cluster.**clustercentroids**(*data*, *mask=None*, *clusterid=None*, *method='a'*, *transpose=False*)

Calculate and return the centroid of each cluster.

The clustercentroids routine calculates the cluster centroids, given to which cluster each item belongs. The centroid is defined as either the mean or the median over all items for each dimension.

Keyword arguments:

- data: nrows x ncolumns array containing the data values.
- mask: nrows x ncolumns array of integers, showing which data are missing. If mask[i, j]==0, then data[i, j] is missing.
- clusterid: array containing the cluster number for each item. The cluster number should be non-negative.
- method: specifies whether the centroid is calculated from the arithmetic mean (method == 'a', default) or the median (method == 'm') over each dimension.
- **transpose: if False, each row contains the data for one item;**
  if True, each column contains the data for one item.

Return values:

- cdata: 2D array containing the cluster centroids. If transpose is False, then the dimensions of cdata are nclusters x ncolumns. If transpose is True, then the dimensions of cdata are nrows x nclusters.
- cmask: 2D array of integers describing which items in cdata, if any, are missing.

Bio.Cluster.**distancematrix**(*data*, *mask=None*, *weight=None*, *transpose=False*, *dist='e'*)

Calculate and return a distance matrix from the data.

This function returns the distance matrix calculated from the data.

Keyword arguments:

- data: nrows x ncolumns array containing the data values.
- mask: nrows x ncolumns array of integers, showing which data are missing. If mask[i, j]==0, then data[i, j] is missing.
- weight: the weights to be used when calculating distances.
- **transpose: if False: the distances between rows are calculated;**
  if True: the distances between columns are calculated.
- dist: specifies the distance function to be used: - dist == 'e': Euclidean distance - dist == 'b': City Block distance - dist == 'c': Pearson correlation - dist == 'a': absolute value of the correlation - dist == 'u': uncentered correlation - dist == 'x': absolute uncentered correlation - dist == 's': Spearman's rank correlation - dist == 'k': Kendall's tau

Return value: The distance matrix is returned as a list of 1D arrays containing the distance matrix calculated from the data. The number of columns in eac row is equal to the row number. Hence, the first row has zero length. For example:

```
>>> from numpy import array
>>> from Bio.Cluster import distancematrix
>>> data = array([[0, 1,  2,  3],
...               [4, 5,  6,  7],
...               [8, 9, 10, 11],
...               [1, 2,  3,  4]])
>>> distances = distancematrix(data, dist='e')
>>> distances
[array([], dtype=float64), array([16.]), array([64., 16.]), array([ 1.,  9., 49.])]
```

which can be rewritten as:

```
distances = [array([], dtype=float64),
             array([ 16.]),
             array([ 64.,  16.]),
             array([  1.,   9.,  49.])]
```

This corresponds to the distance matrix:

```
[ 0., 16., 64.,  1.]
[16.,  0., 16.,  9.]
[64., 16.,  0., 49.]
[ 1.,  9., 49.,  0.]
```

Bio.Cluster.**pca**(*data*)

Perform principal component analysis.

Keyword arguments:

- data: nrows x ncolumns array containing the data values.

Return value: This function returns an array containing the mean of each column, the principal components as an nmin x ncolumns array, as well as the coordinates (an nrows x nmin array) of the data along the principal components, and the associated eigenvalues. The principal components, the coordinates, and the eigenvalues are sorted by the magnitude of the eigenvalue, with the largest eigenvalues appearing first. Here, nmin is the smaller of nrows and ncolumns. Adding the column means to the dot product of the coordinates and the principal components recreates the data matrix:

```
>>> from numpy import array, dot
>>> from Bio.Cluster import pca
>>> matrix = array([[ 0.,  0.,  0.],
...                 [ 1.,  0.,  0.],
...                 [ 7.,  3.,  0.],
...                 [ 4.,  2.,  6.]])
>>> columnmean, coordinates, pc, _ = pca(matrix)
>>> m = matrix - (columnmean + dot(coordinates, pc))
>>> abs(m) < 1e-12
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

class Bio.Cluster.**Record**(*handle=None*)

Bases: `object`

Store gene expression data.

A Record stores the gene expression data and related information contained in a data file following the file format defined for Michael Eisen's Cluster/TreeView program.

Attributes:

- data: a matrix containing the gene expression data
- mask: a matrix containing only 1's and 0's, denoting which values are present (1) or missing (0). If all items of mask are one (no missing data), then mask is set to None.
- geneid: a list containing a unique identifier for each gene (e.g., ORF name)
- genename: a list containing an additional description for each gene (e.g., gene name)
- gweight: the weight to be used for each gene when calculating the distance
- gorder: an array of real numbers indicating the preferred order of the genes in the output file
- expid: a list containing a unique identifier for each sample.
- eweight: the weight to be used for each sample when calculating the distance
- eorder: an array of real numbers indication the preferred order of the samples in the output file
- uniqid: the string that was used instead of UNIQID in the input file.

`__init__`(*handle=None*)

Read gene expression data from the file handle and return a Record.

The file should be in the format defined for Michael Eisen's Cluster/TreeView program.

`treecluster`(*transpose=False*, *method='m'*, *dist='e'*)

Apply hierarchical clustering and return a Tree object.

The pairwise single, complete, centroid, and average linkage hierarchical clustering methods are available.

Keyword arguments:

- **transpose: if False: rows are clustered;**
  if True: columns are clustered.
- dist: specifies the distance function to be used: - dist == 'e': Euclidean distance - dist == 'b': City Block distance - dist == 'c': Pearson correlation - dist == 'a': absolute value of the correlation - dist == 'u': uncentered correlation - dist == 'x': absolute uncentered correlation - dist == 's': Spearman's rank correlation - dist == 'k': Kendall's tau
- method: specifies which linkage method is used: - method == 's': Single pairwise linkage - method == 'm': Complete (maximum) pairwise linkage (default) - method == 'c': Centroid linkage - method == 'a': Average pairwise linkage

See the description of the Tree class for more information about the Tree object returned by this method.

`kcluster`(*nclusters=2*, *transpose=False*, *npass=1*, *method='a'*, *dist='e'*, *initialid=None*)

Apply k-means or k-median clustering.

This method returns a tuple (clusterid, error, nfound).

Keyword arguments:

- nclusters: number of clusters (the 'k' in k-means)
- **transpose: if False, genes (rows) are clustered;**
  if True, samples (columns) are clustered.
- npass: number of times the k-means clustering algorithm is performed, each time with a different (random) initial condition.

- method: specifies how the center of a cluster is found: - method == 'a': arithmetic mean - method == 'm': median
- dist: specifies the distance function to be used: - dist == 'e': Euclidean distance - dist == 'b': City Block distance - dist == 'c': Pearson correlation - dist == 'a': absolute value of the correlation - dist == 'u': uncentered correlation - dist == 'x': absolute uncentered correlation - dist == 's': Spearman's rank correlation - dist == 'k': Kendall's tau
- initialid: the initial clustering from which the algorithm should start. If initialid is None, the routine carries out npass repetitions of the EM algorithm, each time starting from a different random initial clustering. If initialid is given, the routine carries out the EM algorithm only once, starting from the given initial clustering and without randomizing the order in which items are assigned to clusters (i.e., using the same order as in the data matrix). In that case, the k-means algorithm is fully deterministic.

Return values:

- clusterid: array containing the number of the cluster to which each gene/sample was assigned in the best k-means clustering solution that was found in the npass runs;
- error: the within-cluster sum of distances for the returned k-means clustering solution;
- nfound: the number of times this solution was found.

`somcluster`(*transpose=False*, *nxgrid=2*, *nygrid=1*, *inittau=0.02*, *niter=1*, *dist='e'*)

Calculate a self-organizing map on a rectangular grid.

The somcluster method returns a tuple (clusterid, celldata).

Keyword arguments:

- **transpose: if False, genes (rows) are clustered;**
  if True, samples (columns) are clustered.
- nxgrid: the horizontal dimension of the rectangular SOM map
- nygrid: the vertical dimension of the rectangular SOM map
- inittau: the initial value of tau (the neighborbood function)
- niter: the number of iterations
- dist: specifies the distance function to be used: - dist == 'e': Euclidean distance - dist == 'b': City Block distance - dist == 'c': Pearson correlation - dist == 'a': absolute value of the correlation - dist == 'u': uncentered correlation - dist == 'x': absolute uncentered correlation - dist == 's': Spearman's rank correlation - dist == 'k': Kendall's tau

Return values:

- clusterid: array with two columns, while the number of rows is equal to the number of genes or the number of samples depending on whether genes or samples are being clustered. Each row in the array contains the x and y coordinates of the cell in the rectangular SOM grid to which the gene or samples was assigned.
- celldata: an array with dimensions (nxgrid, nygrid, number of samples) if genes are being clustered, or (nxgrid, nygrid, number of genes) if samples are being clustered. Each item [ix, iy] of this array is a 1D vector containing the gene expression data for the centroid of the cluster in the SOM grid cell with coordinates [ix, iy].

`clustercentroids`(*clusterid=None*, *method='a'*, *transpose=False*)

Calculate the cluster centroids and return a tuple (cdata, cmask).

The centroid is defined as either the mean or the median over all items for each dimension.

Keyword arguments:

- data: nrows x ncolumns array containing the expression data
- mask: nrows x ncolumns array of integers, showing which data are missing. If mask[i, j]==0, then data[i, j] is missing.
- **transpose: if False, gene (row) clusters are considered;**
  if True, sample (column) clusters are considered.
- clusterid: array containing the cluster number for each gene or sample. The cluster number should be non-negative.
- method: specifies how the centroid is calculated: - method == 'a': arithmetic mean over each dimension. (default) - method == 'm': median over each dimension.

Return values:

- cdata: 2D array containing the cluster centroids. If transpose is False, then the dimensions of cdata are nclusters x ncolumns. If transpose is True, then the dimensions of cdata are nrows x nclusters.
- cmask: 2D array of integers describing which items in cdata, if any, are missing.

`clusterdistance`(*index1=0*, *index2=0*, *method='a'*, *dist='e'*, *transpose=False*)

Calculate the distance between two clusters.

Keyword arguments:

- index1: 1D array identifying which genes/samples belong to the first cluster. If the cluster contains only one gene, then index1 can also be written as a single integer.
- index2: 1D array identifying which genes/samples belong to the second cluster. If the cluster contains only one gene, then index2 can also be written as a single integer.
- **transpose: if False, genes (rows) are clustered;**
  if True, samples (columns) are clustered.
- dist: specifies the distance function to be used: - dist == 'e': Euclidean distance - dist == 'b': City Block distance - dist == 'c': Pearson correlation - dist == 'a': absolute value of the correlation - dist == 'u': uncentered correlation - dist == 'x': absolute uncentered correlation - dist == 's': Spearman's rank correlation - dist == 'k': Kendall's tau
- method: specifies how the distance between two clusters is defined: - method == 'a': the distance between the arithmetic means of the two clusters - method == 'm': the distance between the medians of the two clusters - method == 's': the smallest pairwise distance between members of the two clusters - method == 'x': the largest pairwise distance between members of the two clusters - method == 'v': average of the pairwise distances between members of the two clusters
- **transpose: if False: clusters of rows are considered;**
  if True: clusters of columns are considered.

`distancematrix`(*transpose=False*, *dist='e'*)

Calculate the distance matrix and return it as a list of arrays.

Keyword arguments:

- **transpose:**
  if False: calculate the distances between genes (rows); if True: calculate the distances between samples (columns).
- dist: specifies the distance function to be used: - dist == 'e': Euclidean distance - dist == 'b': City Block distance - dist == 'c': Pearson correlation - dist == 'a': absolute value of the correlation - dist == 'u': uncentered correlation - dist == 'x': absolute uncentered correlation - dist == 's': Spearman's rank correlation - dist == 'k': Kendall's tau

Return value:

The distance matrix is returned as a list of 1D arrays containing the distance matrix between the gene expression data. The number of columns in each row is equal to the row number. Hence, the first row has zero length. An example of the return value is:

matrix = [[],
array([1.]), array([7., 3.]), array([4., 2., 6.])]

This corresponds to the distance matrix:

[0., 1., 7., 4.] [1., 0., 3., 2.] [7., 3., 0., 6.] [4., 2., 6., 0.]

save(*jobname*, *geneclusters=None*, *expclusters=None*)

Save the clustering results.

The saved files follow the convention for the Java TreeView program, which can therefore be used to view the clustering result.

Keyword arguments:

- jobname: The base name of the files to be saved. The filenames are jobname.cdt, jobname.gtr, and jobname.atr for hierarchical clustering, and jobname-K*.cdt, jobname-K*.kgg, jobname-K*.kag for k-means clustering results.
- geneclusters: For hierarchical clustering results, geneclusters is a Tree object as returned by the treecluster method. For k-means clustering results, geneclusters is a vector containing ngenes integers, describing to which cluster a given gene belongs. This vector can be calculated by kcluster.
- expclusters: For hierarchical clustering results, expclusters is a Tree object as returned by the treecluster method. For k-means clustering results, expclusters is a vector containing nexps integers, describing to which cluster a given sample belongs. This vector can be calculated by kcluster.

Bio.Cluster.**read**(*handle*)

Read gene expression data from the file handle and return a Record.

The file should be in the file format defined for Michael Eisen's Cluster/TreeView program.

28.1.8 Bio.Compass package

Module contents

Code to deal with COMPASS output, a program for profile/profile comparison.

Compass is described in:

Sadreyev R, Grishin N. COMPASS: a tool for comparison of multiple protein alignments with assessment of statistical significance. J Mol Biol. 2003 Feb 7;326(1):317-36.

Tested with COMPASS 1.24.

Bio.Compass.**read**(*handle*)

Read a COMPASS file containing one COMPASS record.

Bio.Compass.**parse**(*handle*)

Iterate over records in a COMPASS file.

class Bio.Compass.**Record**

Bases: object

Hold information from one compass hit.

Ali1 is the query, Ali2 the hit.

__init__()

Initialize the class.

query_coverage()

Return the length of the query covered in the alignment.

hit_coverage()

Return the length of the hit covered in the alignment.

28.1.9 Bio.Data package

Submodules

Bio.Data.CodonTable module

Codon tables based on those from the NCBI.

These tables are based on parsing the NCBI file ftp://ftp.ncbi.nih.gov/entrez/misc/data/gc.prt using Scripts/update_ncbi_codon_table.py

Last updated at Version 4.4 (May 2019)

exception Bio.Data.CodonTable.**TranslationError**

Bases: Exception

Container for translation specific exceptions.

class Bio.Data.CodonTable.**CodonTable**(*nucleotide_alphabet: str | None = None*, *protein_alphabet: str | None = None*, *forward_table: dict[str, str] = forward_table*, *back_table: dict[str, str] = back_table*, *start_codons: list[str] = start_codons*, *stop_codons: list[str] = stop_codons*)

Bases: object

A codon-table, or genetic code.

__init__(*nucleotide_alphabet: str | None = None*, *protein_alphabet: str | None = None*, *forward_table: dict[str, str] = forward_table*, *back_table: dict[str, str] = back_table*, *start_codons: list[str] = start_codons*, *stop_codons: list[str] = stop_codons*) → None

Initialize the class.

forward_table: dict[str, str] = {}

back_table: dict[str, str] = {}

start_codons: list[str] = []

stop_codons: list[str] = []

__str__()

Return a simple text representation of the codon table.

e.g.:

```
>>> import Bio.Data.CodonTable
>>> print(Bio.Data.CodonTable.standard_dna_table)
Table 1 Standard, SGC0

  |  T      |  C      |  A      |  G      |
--+---------+---------+---------+---------+--
T | TTT F   | TCT S   | TAT Y   | TGT C   | T
T | TTC F   | TCC S   | TAC Y   | TGC C   | C
...
G | GTA V   | GCA A   | GAA E   | GGA G   | A
G | GTG V   | GCG A   | GAG E   | GGG G   | G
--+---------+---------+---------+---------+--
>>> print(Bio.Data.CodonTable.generic_by_id[1])
Table 1 Standard, SGC0

  |  U      |  C      |  A      |  G      |
--+---------+---------+---------+---------+--
U | UUU F   | UCU S   | UAU Y   | UGU C   | U
U | UUC F   | UCC S   | UAC Y   | UGC C   | C
...
G | GUA V   | GCA A   | GAA E   | GGA G   | A
G | GUG V   | GCG A   | GAG E   | GGG G   | G
--+---------+---------+---------+---------+--
```

__annotations__ = {'back_table': dict[str, str], 'forward_table': dict[str, str], 'start_codons': list[str], 'stop_codons': list[str]}

Bio.Data.CodonTable.**make_back_table**(*table*, *default_stop_codon*)

Back a back-table (naive single codon mapping).

ONLY RETURNS A SINGLE CODON, chosen from the possible alternatives based on their sort order.

class Bio.Data.CodonTable.**NCBICodonTable**(*id*, *names*, *table*, *start_codons*, *stop_codons*)

Bases: *CodonTable*

Codon table for generic nucleotide sequences.

nucleotide_alphabet: str | None = None

protein_alphabet = 'ACDEFGHIKLMNPQRSTVWY'

__init__(*id*, *names*, *table*, *start_codons*, *stop_codons*)

Initialize the class.

__repr__()

Represent the NCBI codon table class as a string for debugging.

__annotations__ = {'back_table': 'dict[str, str]', 'forward_table': 'dict[str, str]', 'nucleotide_alphabet': typing.Optional[str], 'start_codons': 'list[str]', 'stop_codons': 'list[str]'}

class Bio.Data.CodonTable.**NCBICodonTableDNA**(*id*, *names*, *table*, *start_codons*, *stop_codons*)

Bases: *NCBICodonTable*

Codon table for unambiguous DNA sequences.

nucleotide_alphabet: str | None = 'GATC'

__annotations__ = {'back_table': 'dict[str, str]', 'forward_table': 'dict[str, str]', 'nucleotide_alphabet': 'Optional[str]', 'start_codons': 'list[str]', 'stop_codons': 'list[str]'}

class Bio.Data.CodonTable.**NCBICodonTableRNA**(*id*, *names*, *table*, *start_codons*, *stop_codons*)

Bases: *NCBICodonTable*

Codon table for unambiguous RNA sequences.

nucleotide_alphabet: str | None = 'GAUC'

__annotations__ = {'back_table': 'dict[str, str]', 'forward_table': 'dict[str, str]', 'nucleotide_alphabet': 'Optional[str]', 'start_codons': 'list[str]', 'stop_codons': 'list[str]'}

class Bio.Data.CodonTable.**AmbiguousCodonTable**(*codon_table*, *ambiguous_nucleotide_alphabet*, *ambiguous_nucleotide_values*, *ambiguous_protein_alphabet*, *ambiguous_protein_values*)

Bases: *CodonTable*

Base codon table for ambiguous sequences.

__init__(*codon_table*, *ambiguous_nucleotide_alphabet*, *ambiguous_nucleotide_values*, *ambiguous_protein_alphabet*, *ambiguous_protein_values*)

Initialize the class.

__getattr__(*name*)

Forward attribute lookups to the original table.

__annotations__ = {}

Bio.Data.CodonTable.**list_possible_proteins**(*codon*, *forward_table*, *ambiguous_nucleotide_values*)

Return all possible encoded amino acids for ambiguous codon.

Bio.Data.CodonTable.**list_ambiguous_codons**(*codons*, *ambiguous_nucleotide_values*)

Extend a codon list to include all possible ambiguous codons.

e.g.:

```
['TAG', 'TAA'] -> ['TAG', 'TAA', 'TAR']
['UAG', 'UGA'] -> ['UAG', 'UGA', 'URA']
```

Note that [‘TAG’, ‘TGA’] -> [‘TAG’, ‘TGA’], this does not add ‘TRR’ (which could also mean ‘TAA’ or ‘TGG’). Thus only two more codons are added in the following:

e.g.:

```
['TGA', 'TAA', 'TAG'] -> ['TGA', 'TAA', 'TAG', 'TRA', 'TAR']
```

Returns a new (longer) list of codon strings.

class Bio.Data.CodonTable.**AmbiguousForwardTable**(*forward_table*, *ambiguous_nucleotide*, *ambiguous_protein*)

Bases: object

Forward table for translation of ambiguous nucleotide sequences.

__init__(*forward_table*, *ambiguous_nucleotide*, *ambiguous_protein*)

Initialize the class.

__contains__(*codon*)

Check if codon works as key for ambiguous forward_table.

Only returns 'True' if forward_table[codon] returns a value.

get(*codon*, *failobj=None*)

Implement get for dictionary-like behaviour.

__getitem__(*codon*)

Implement dictionary-like behaviour for AmbiguousForwardTable.

forward_table[codon] will either return an amino acid letter, or throws a KeyError (if codon does not encode an amino acid) or a TranslationError (if codon does encode for an amino acid, but either is also a stop codon or does encode several amino acids, for which no unique letter is available in the given alphabet.

Bio.Data.CodonTable.**register_ncbi_table**(*name*, *alt_name*, *id*, *table*, *start_codons*, *stop_codons*)

Turn codon table data into objects (PRIVATE).

The data is stored in the dictionaries.

Bio.Data.IUPACData module

Information about the IUPAC alphabets.

Bio.Data.PDBData module

Alphabets used by the wwPDB in structural file formats.

Module contents

Collections of various bits of useful biological data.

28.1.10 Bio.Emboss package

Submodules

Bio.Emboss.Applications module

Code to interact with and run various EMBOSS programs (OBSOLETE).

These classes follow the AbstractCommandline interfaces for running programs.

We have decided to remove this module in future, and instead recommend building your command and invoking it via the subprocess module directly.

class Bio.Emboss.Applications.**Primer3Commandline**(*cmd='eprimer3'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the Primer3 interface from EMBOSS.

The precise set of supported arguments depends on your version of EMBOSS. This version accepts arguments current at EMBOSS 6.1.0:

```
>>> cline = Primer3Commandline(sequence="mysequence.fas", auto=True,␣
↪hybridprobe=True)
>>> cline.explainflag = True
>>> cline.osizeopt=20
>>> cline.psizeopt=200
>>> cline.outfile = "myresults.out"
>>> cline.bogusparameter = 1967  # Invalid parameter
Traceback (most recent call last):
    ...
ValueError: Option name bogusparameter was not found.
>>> print(cline)
eprimer3 -auto -outfile=myresults.out -sequence=mysequence.fas -hybridprobe=True -
↪psizeopt=200 -osizeopt=20 -explainflag=True
```

__init__(*cmd='eprimer3'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property dnaconc

Nanomolar concentration of annealing oligos in the PCR.

This controls the addition of the -dnaconc parameter and its associated value. Set this property to the argument value required.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property excludedregion

Regions to exclude from primer picking.

This controls the addition of the -excludedregion parameter and its associated value. Set this property to the argument value required.

property explainflag

Produce output tags with eprimer3 statistics

This controls the addition of the -explainflag parameter and its associated value. Set this property to the argument value required.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property forwardinput

Sequence of a forward primer to check.

This controls the addition of the -forwardinput parameter and its associated value. Set this property to the argument value required.

property gcclamp

The required number of Gs and Cs at the 3' of each primer.

This controls the addition of the -gcclamp parameter and its associated value. Set this property to the argument value required.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property hybridprobe

Find an internal oligo to use as a hyb probe.

This controls the addition of the -hybridprobe parameter and its associated value. Set this property to the argument value required.

property includedregion

Subregion of the sequence in which to pick primers.

This controls the addition of the -includedregion parameter and its associated value. Set this property to the argument value required.

property maxdifftm

Maximum difference in melting temperatures between forward and reverse primers.

This controls the addition of the -maxdifftm parameter and its associated value. Set this property to the argument value required.

property maxgc

Maximum GC% for a primer.

This controls the addition of the -maxgc parameter and its associated value. Set this property to the argument value required.

property maxmispriming

Maximum allowed similarity of primers to sequences in library specified by -misprominglibrary

This controls the addition of the -maxmispriming parameter and its associated value. Set this property to the argument value required.

property maxpolyx

Maximum allowable mononucleotide repeat length in a primer.

This controls the addition of the -maxpolyx parameter and its associated value. Set this property to the argument value required.

property maxsize

Maximum length of a primer oligo.

This controls the addition of the -maxsize parameter and its associated value. Set this property to the argument value required.

property maxtm

Maximum melting temperature for a primer oligo.

This controls the addition of the -maxtm parameter and its associated value. Set this property to the argument value required.

property mingc

Minimum GC% for a primer.

This controls the addition of the -mingc parameter and its associated value. Set this property to the argument value required.

property minsize

Minimum length of a primer oligo.

This controls the addition of the -minsize parameter and its associated value. Set this property to the argument value required.

property mintm

Minimum melting temperature for a primer oligo.

This controls the addition of the -mintm parameter and its associated value. Set this property to the argument value required.

property mishyblibraryfile

Library file of seqs to avoid internal oligo hybridisation.

This controls the addition of the -mishyblibraryfile parameter and its associated value. Set this property to the argument value required.

property misprominglibraryfile

File containing library of sequences to avoid amplifying

This controls the addition of the -misprominglibraryfile parameter and its associated value. Set this property to the argument value required.

property numreturn

Maximum number of primer pairs to return.

This controls the addition of the -numreturn parameter and its associated value. Set this property to the argument value required.

property oanyself

Maximum allowable alignment score for self-complementarity.

This controls the addition of the -oanyself parameter and its associated value. Set this property to the argument value required.

property odnaconc

Nanomolar concentration of internal oligo in the hybridisation.

This controls the addition of the -odnaconc parameter and its associated value. Set this property to the argument value required.

property oendself

Max 3'-anchored self-complementarity global alignment score.

This controls the addition of the -oendself parameter and its associated value. Set this property to the argument value required.

property oexcludedregion

Do not pick internal oligos in this region.

This controls the addition of the -oexcludedregion parameter and its associated value. Set this property to the argument value required.

property ogcmax

Maximum GC% for internal oligo.

This controls the addition of the -ogcmax parameter and its associated value. Set this property to the argument value required.

property ogcmin

Minimum GC% for internal oligo.

This controls the addition of the -ogcmin parameter and its associated value. Set this property to the argument value required.

property ogcopt

Optimum GC% for internal oligo.

This controls the addition of the -ogcopt parameter and its associated value. Set this property to the argument value required.

property ogcpercent

Optimum GC% for a primer.

This controls the addition of the -ogcpercent parameter and its associated value. Set this property to the argument value required.

property oligoinput

Sequence of the internal oligo.

This controls the addition of the -oligoinput parameter and its associated value. Set this property to the argument value required.

property omaxsize

Maximum length of internal oligo.

This controls the addition of the -omaxsize parameter and its associated value. Set this property to the argument value required.

property ominsize

Minimum length of internal oligo.

This controls the addition of the -ominsize parameter and its associated value. Set this property to the argument value required.

property omishybmax

Maximum alignment score for hybridisation of internal oligo to library specified by -mishyblibraryfile.

This controls the addition of the -omishybmax parameter and its associated value. Set this property to the argument value required.

property opolyxmax

Maximum length of mononucleotide repeat in internal oligo.

This controls the addition of the -opolyxmax parameter and its associated value. Set this property to the argument value required.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property opttm

Optimum melting temperature for a primer oligo.

Option added in EMBOSS 6.6.0, replacing -otm

This controls the addition of the -opttm parameter and its associated value. Set this property to the argument value required.

property osaltconc

Millimolar concentration of salt in the hybridisation.

This controls the addition of the -osaltconc parameter and its associated value. Set this property to the argument value required.

property osize

Optimum length of a primer oligo.

This controls the addition of the -osize parameter and its associated value. Set this property to the argument value required.

property osizeopt

Optimum length of internal oligo.

This controls the addition of the -osizeopt parameter and its associated value. Set this property to the argument value required.

property otm

Melting temperature for primer oligo (OBSOLETE).

Option replaced in EMBOSS 6.6.0 by -opttm

This controls the addition of the -otm parameter and its associated value. Set this property to the argument value required.

property otmmax

Maximum melting temperature of internal oligo.

This controls the addition of the -otmmax parameter and its associated value. Set this property to the argument value required.

property otmmin

Minimum melting temperature of internal oligo.

This controls the addition of the -otmmin parameter and its associated value. Set this property to the argument value required.

property otmopt

Optimum melting temperature of internal oligo.

This controls the addition of the -otmopt parameter and its associated value. Set this property to the argument value required.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property prange

Acceptable range of length for the PCR product.

This controls the addition of the -prange parameter and its associated value. Set this property to the argument value required.

property psizeopt

Optimum size for the PCR product.

This controls the addition of the -psizeopt parameter and its associated value. Set this property to the argument value required.

property ptmmax

Maximum allowed melting temperature for the amplicon.

This controls the addition of the -ptmmax parameter and its associated value. Set this property to the argument value required.

property ptmmin

Minimum allowed melting temperature for the amplicon.

This controls the addition of the -ptmmin parameter and its associated value. Set this property to the argument value required.

property ptmopt

Optimum melting temperature for the PCR product.

This controls the addition of the -ptmopt parameter and its associated value. Set this property to the argument value required.

property reverseinput

Sequence of a reverse primer to check.

This controls the addition of the -reverseinput parameter and its associated value. Set this property to the argument value required.

property saltconc

Millimolar salt concentration in the PCR.

This controls the addition of the -saltconc parameter and its associated value. Set this property to the argument value required.

property sequence

Sequence to choose primers from.

This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property target

Sequence to target for flanking primers.

This controls the addition of the -target parameter and its associated value. Set this property to the argument value required.

property task

Tell eprimer3 what task to perform.

This controls the addition of the -task parameter and its associated value. Set this property to the argument value required.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**PrimerSearchCommandline**(*cmd='primersearch'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the primersearch program from EMBOSS.

__init__(*cmd='primersearch'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property infile

File containing the primer pairs to search for.

This controls the addition of the -infile parameter and its associated value. Set this property to the argument value required.

property mismatchpercent

Allowed percentage mismatch (any integer value, default 0).

This controls the addition of the -mismatchpercent parameter and its associated value. Set this property to the argument value required.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property seqall

Sequence to look for the primer pairs in.

This controls the addition of the -seqall parameter and its associated value. Set this property to the argument value required.

property snucleotide

Sequences are nucleotide (boolean)

This controls the addition of the -snucleotide parameter and its associated value. Set this property to the argument value required.

property sprotein

Sequences are protein (boolean)

This controls the addition of the -sprotein parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**FDNADistCommandline**(*cmd='fdnadist'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the fdnadist program from EMBOSS.

fdnadist is an EMBOSS wrapper for the PHYLIP program dnadist for calculating distance matrices from DNA sequence files.

__init__(*cmd='fdnadist'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property basefreq

specify basefreqs

This controls the addition of the -basefreq parameter and its associated value. Set this property to the argument value required.

property categories

File of substitution rate categories

This controls the addition of the -categories parameter and its associated value. Set this property to the argument value required.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property freqsfrom

use empirical base freqs

This controls the addition of the -freqsfrom parameter and its associated value. Set this property to the argument value required.

property gamma

This controls the addition of the -gamma parameter and its associated value. Set this property to the argument value required.

property gammacoefficient

value for gamma (> 0.001)

This controls the addition of the -gammacoefficient parameter and its associated value. Set this property to the argument value required.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property invarfrac

proportoin of invariant sites

This controls the addition of the -invarfrac parameter and its associated value. Set this property to the argument value required.

property lower

lower triangle matrix (y/N)

This controls the addition of the -lower parameter and its associated value. Set this property to the argument value required.

property method

sub. model [f,k,j,l,s]

This controls the addition of the -method parameter and its associated value. Set this property to the argument value required.

property ncategories

number of rate categories (1-9)

This controls the addition of the -ncategories parameter and its associated value. Set this property to the argument value required.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property rate

rate for each category

This controls the addition of the -rate parameter and its associated value. Set this property to the argument value required.

property sequence

seq file to use (phylip)

This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property ttratio

ts/tv ratio

This controls the addition of the -ttratio parameter and its associated value. Set this property to the argument value required.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

property weights

weights file

This controls the addition of the -weights parameter and its associated value. Set this property to the argument value required.

class Bio.Emboss.Applications.**FTreeDistCommandline**(*cmd='ftreedist'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the ftreedist program from EMBOSS.

ftreedist is an EMBOSS wrapper for the PHYLIP program treedist used for calculating distance measures between phylogentic trees.

__init__(*cmd='ftreedist'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property dtype

distance type ([S]ymetric, [b]ranch score)

This controls the addition of the -dtype parameter and its associated value. Set this property to the argument value required.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property intreefile

tree file to score (phylip)

This controls the addition of the -intreefile parameter and its associated value. Set this property to the argument value required.

property noroot

treat trees as rooted [N/y]

This controls the addition of the -noroot parameter and its associated value. Set this property to the argument value required.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property outgrno

which taxon to root the trees with (starts from 0)

This controls the addition of the -outgrno parameter and its associated value. Set this property to the argument value required.

property pairing

tree pairing method ([A]djacent pairs, all [p]ossible pairs)

This controls the addition of the -pairing parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property style

output style - [V]erbose, [f]ill, [s]parse

This controls the addition of the -style parameter and its associated value. Set this property to the argument value required.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**FNeighborCommandline**(*cmd='fneighbor'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the fneighbor program from EMBOSS.

fneighbor is an EMBOSS wrapper for the PHYLIP program neighbor used for calculating neighbor-joining or UPGMA trees from distance matrices.

__init__(*cmd='fneighbor'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property datafile
: dist file to use (phylip)

  This controls the addition of the -datafile parameter and its associated value. Set this property to the argument value required.

property debug
: Write debug output to program.dbg.

  This property controls the addition of the -debug switch, treat this property as a boolean.

property die
: Report dying program messages.

  This property controls the addition of the -die switch, treat this property as a boolean.

property error
: Report errors.

  This property controls the addition of the -error switch, treat this property as a boolean.

property filter
: Read standard input, write standard output.

  This property controls the addition of the -filter switch, treat this property as a boolean.

property help
: Report command line options.

  More information on associated and general qualifiers can be found with -help -verbose

  This property controls the addition of the -help switch, treat this property as a boolean.

property jumble
: randommise input order (Y/n)

  This controls the addition of the -jumble parameter and its associated value. Set this property to the argument value required.

property matrixtype
: is matrix square (S), upper (U) or lower (L)

  This controls the addition of the -matrixtype parameter and its associated value. Set this property to the argument value required.

property options
: Prompt for standard and additional values.

  If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

  This property controls the addition of the -options switch, treat this property as a boolean.

property outfile
: Output filename

  This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property outgrno
: taxon to use as OG

  This controls the addition of the -outgrno parameter and its associated value. Set this property to the argument value required.

property outtreefile

filename for output tree

This controls the addition of the -outtreefile parameter and its associated value. Set this property to the argument value required.

property progress

print progress (Y/n)

This controls the addition of the -progress parameter and its associated value. Set this property to the argument value required.

property seed

provide a random seed

This controls the addition of the -seed parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property treeprint

print tree (Y/n)

This controls the addition of the -treeprint parameter and its associated value. Set this property to the argument value required.

property treetype

nj or UPGMA tree (n/u)

This controls the addition of the -treetype parameter and its associated value. Set this property to the argument value required.

property trout

write tree (Y/n)

This controls the addition of the -trout parameter and its associated value. Set this property to the argument value required.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**FSeqBootCommandline**(*cmd='fseqboot'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the fseqboot program from EMBOSS.

fseqboot is an EMBOSS wrapper for the PHYLIP program seqboot used to pseudo-sample alignment files.

__init__(*cmd='fseqboot'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property blocksize

print progress (Y/n)

This controls the addition of the -blocksize parameter and its associated value. Set this property to the argument value required.

property catergories

file of input categories

This controls the addition of the -categories parameter and its associated value. Set this property to the argument value required.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property dotdiff

Use dot-differencing? [Y/n]

This controls the addition of the -dotdiff parameter and its associated value. Set this property to the argument value required.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property fracsample

fraction to resample

This controls the addition of the -fracsample parameter and its associated value. Set this property to the argument value required.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property jusweights

what to write out [D]atasets of just [w]eights

This controls the addition of the -justweights parameter and its associated value. Set this property to the argument value required.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property regular

absolute number to resample

This controls the addition of the -regular parameter and its associated value. Set this property to the argument value required.

property reps

how many replicates, defaults to 100)

This controls the addition of the -reps parameter and its associated value. Set this property to the argument value required.

property rewriteformat

output format ([P]hyilp, [n]exus, [x]ml

This controls the addition of the -rewriteformat parameter and its associated value. Set this property to the argument value required.

property seed

specify random seed

This controls the addition of the -seed parameter and its associated value. Set this property to the argument value required.

property seqtype

output format ([D]na, [p]rotein, [r]na

This controls the addition of the -seqtype parameter and its associated value. Set this property to the argument value required.

property sequence

seq file to sample (phylip)

This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property test

specify operation, default is bootstrap

This controls the addition of the -test parameter and its associated value. Set this property to the argument value required.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

property weights

weights file

This controls the addition of the -weights parameter and its associated value. Set this property to the argument value required.

class Bio.Emboss.Applications.**FDNAParsCommandline**(*cmd='fdnapars'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the fdnapars program from EMBOSS.

fdnapars is an EMBOSS version of the PHYLIP program dnapars, for estimating trees from DNA sequences using parsiomny. Calling this command without providing a value for the option "-intreefile" will invoke "interactive mode" (and as a result fail if called with subprocess) if "-auto" is not set to true.

__init__(*cmd='fdnapars'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property dotdiff

Use dot-differencing? [Y/n]

This controls the addition of the -dotdiff parameter and its associated value. Set this property to the argument value required.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property intreefile

Phylip tree file

This controls the addition of the -intreefile parameter and its associated value. Set this property to the argument value required.

property maxtrees

max trees to save during run

This controls the addition of the -maxtrees parameter and its associated value. Set this property to the argument value required.

property njumble

number of times to randomise input order (default is 0)

This controls the addition of the -njumble parameter and its associated value. Set this property to the argument value required.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property outgrno

Specify outgroup

This controls the addition of the -outgrno parameter and its associated value. Set this property to the argument value required.

property outtreefile

filename for output tree

This controls the addition of the -outtreefile parameter and its associated value. Set this property to the argument value required.

property rearrange

Rearrange on just 1 best tree (Y/n)

This controls the addition of the -rearrange parameter and its associated value. Set this property to the argument value required.

property seed

provide random seed

This controls the addition of the -seed parameter and its associated value. Set this property to the argument value required.

property sequence

seq file to use (phylip)

This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property thorough

more thorough search (Y/n)

This controls the addition of the -thorough parameter and its associated value. Set this property to the argument value required.

property thresh

Use threshold parsimony (y/N)

This controls the addition of the -thresh parameter and its associated value. Set this property to the argument value required.

property threshold

Threshold value

This controls the addition of the -threshold parameter and its associated value. Set this property to the argument value required.

property transversion

Use tranversion parsimony (y/N)

This controls the addition of the -transversion parameter and its associated value. Set this property to the argument value required.

property trout

Write trees to file (Y/n)

This controls the addition of the -trout parameter and its associated value. Set this property to the argument value required.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

property weights

weights file

This controls the addition of the -weights parameter and its associated value. Set this property to the argument value required.

class Bio.Emboss.Applications.**FProtParsCommandline**(*cmd='fprotpars'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the fdnapars program from EMBOSS.

fprotpars is an EMBOSS version of the PHYLIP program protpars, for estimating trees from protein sequences using parsiomny. Calling this command without providing a value for the option "-intreefile" will invoke "interactive mode" (and as a result fail if called with subprocess) if "-auto" is not set to true.

__init__(*cmd='fprotpars'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property dotdiff

Use dot-differencing? [Y/n]

This controls the addition of the -dotdiff parameter and its associated value. Set this property to the argument value required.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property intreefile

Phylip tree file to score

This controls the addition of the -intreefile parameter and its associated value. Set this property to the argument value required.

property njumble

number of times to randomise input order (default is 0)

This controls the addition of the -njumble parameter and its associated value. Set this property to the argument value required.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property outgrno

Specify outgroup

This controls the addition of the -outgrno parameter and its associated value. Set this property to the argument value required.

property outtreefile

phylip tree output file

This controls the addition of the -outtreefile parameter and its associated value. Set this property to the argument value required.

property seed

provide random seed

This controls the addition of the -seed parameter and its associated value. Set this property to the argument value required.

property sequence

seq file to use (phylip)

This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property thresh

Use threshold parsimony (y/N)

This controls the addition of the -thresh parameter and its associated value. Set this property to the argument value required.

property threshold

Threshold value

This controls the addition of the -threshold parameter and its associated value. Set this property to the argument value required.

property trout

Write trees to file (Y/n)

This controls the addition of the -trout parameter and its associated value. Set this property to the argument value required.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

property weights

weights file

This controls the addition of the -weights parameter and its associated value. Set this property to the argument value required.

property whichcode

which genetic code, [U,M,V,F,Y]]

This controls the addition of the -whichcode parameter and its associated value. Set this property to the argument value required.

class Bio.Emboss.Applications.**FProtDistCommandline**(*cmd='fprotdist'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the fprotdist program from EMBOSS.

fprotdist is an EMBOSS wrapper for the PHYLIP program protdist used to estimate trees from protein sequences using parsimony

__init__(*cmd='fprotdist'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property aacateg

Choose the category to use [G,C,H]

This controls the addition of the -aacateg parameter and its associated value. Set this property to the argument value required.

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property basefreq

DNA base frequencies (space separated list)

This controls the addition of the -basefreq parameter and its associated value. Set this property to the argument value required.

property catergories

file of rates

This controls the addition of the -catergories parameter and its associated value. Set this property to the argument value required.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property ease

Pob change category (float between -0 and 1)

This controls the addition of the -ease parameter and its associated value. Set this property to the argument value required.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property gamma

This controls the addition of the -gamma parameter and its associated value. Set this property to the argument value required.

property gammacoefficient

value for gamma (> 0.001)

This controls the addition of the -gammacoefficient parameter and its associated value. Set this property to the argument value required.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property invarcoefficient

float for variation of substitution rate among sites

This controls the addition of the -invarcoefficient parameter and its associated value. Set this property to the argument value required.

property method

sub. model [j,h,d,k,s,c]

This controls the addition of the -method parameter and its associated value. Set this property to the argument value required.

property ncategories

number of rate categories (1-9)

This controls the addition of the -ncategories parameter and its associated value. Set this property to the argument value required.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property rate

rate for each category

This controls the addition of the -rate parameter and its associated value. Set this property to the argument value required.

property sequence

seq file to use (phylip)

This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property ttratio

Transition/transversion ratio (0-1)

This controls the addition of the -ttratio parameter and its associated value. Set this property to the argument value required.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

property weights

weights file

This controls the addition of the -weights parameter and its associated value. Set this property to the argument value required.

property whichcode

genetic code [c,m,v,f,y]

This controls the addition of the -whichcode parameter and its associated value. Set this property to the argument value required.

class Bio.Emboss.Applications.FConsenseCommandline(*cmd='fconsense'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the fconsense program from EMBOSS.

fconsense is an EMBOSS wrapper for the PHYLIP program consense used to calculate consensus trees.

__init__(*cmd='fconsense'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property intreefile

file with phylip trees to make consensus from

This controls the addition of the -intreefile parameter and its associated value. Set this property to the argument value required.

property method

consensus method [s, mr, MRE, ml]

This controls the addition of the -method parameter and its associated value. Set this property to the argument value required.

property mlfrac

cut-off freq for branch to appear in consensus (0.5-1.0)

This controls the addition of the -mlfrac parameter and its associated value. Set this property to the argument value required.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property outgrno

OTU to use as outgroup (starts from 0)

This controls the addition of the -outgrno parameter and its associated value. Set this property to the argument value required.

property outtreefile

Phylip tree output file (optional)

This controls the addition of the -outtreefile parameter and its associated value. Set this property to the argument value required.

property root

treat trees as rooted (YES, no)

This controls the addition of the -root parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property trout

treat trees as rooted (YES, no)

This controls the addition of the -trout parameter and its associated value. Set this property to the argument value required.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.WaterCommandline(*cmd='water'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the water program from EMBOSS.

__init__(*cmd='water'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property aformat

Display output in a different specified output format

This controls the addition of the -aformat parameter and its associated value. Set this property to the argument value required.

property asequence

First sequence to align

This controls the addition of the -asequence parameter and its associated value. Set this property to the argument value required.

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property brief

Display brief identity and similarity

This property controls the addition of the -brief switch, treat this property as a boolean.

property bsequence

Second sequence to align

This controls the addition of the -bsequence parameter and its associated value. Set this property to the argument value required.

property datafile

Matrix file

This controls the addition of the -datafile parameter and its associated value. Set this property to the argument value required.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property gapextend

Gap extension penalty

This controls the addition of the -gapextend parameter and its associated value. Set this property to the argument value required.

property gapopen

Gap open penalty

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property nobrief

Display extended identity and similarity

This property controls the addition of the -nobrief switch, treat this property as a boolean.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property similarity

Display percent identity and similarity

This controls the addition of the -similarity parameter and its associated value. Set this property to the argument value required.

property snucleotide

Sequences are nucleotide (boolean)

This controls the addition of the -snucleotide parameter and its associated value. Set this property to the argument value required.

property sprotein

Sequences are protein (boolean)

This controls the addition of the -sprotein parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**NeedleCommandline**(*cmd='needle'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the needle program from EMBOSS.

__init__(*cmd='needle'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property aformat

Display output in a different specified output format

This controls the addition of the -aformat parameter and its associated value. Set this property to the argument value required.

property asequence

First sequence to align

This controls the addition of the -asequence parameter and its associated value. Set this property to the argument value required.

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property brief

Display brief identity and similarity

This property controls the addition of the -brief switch, treat this property as a boolean.

property bsequence

Second sequence to align

This controls the addition of the -bsequence parameter and its associated value. Set this property to the argument value required.

property datafile

Matrix file

This controls the addition of the -datafile parameter and its associated value. Set this property to the argument value required.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property endextend

The score added to the end gap penalty for each base or residue in the end gap.

This controls the addition of the -endextend parameter and its associated value. Set this property to the argument value required.

property endopen

The score taken away when an end gap is created.

This controls the addition of the -endopen parameter and its associated value. Set this property to the argument value required.

property endweight

Apply And gap penalties

This controls the addition of the -endweight parameter and its associated value. Set this property to the argument value required.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property gapextend

Gap extension penalty

This controls the addition of the -gapextend parameter and its associated value. Set this property to the argument value required.

property gapopen

Gap open penalty

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property nobrief

Display extended identity and similarity

This property controls the addition of the -nobrief switch, treat this property as a boolean.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property similarity

Display percent identity and similarity

This controls the addition of the -similarity parameter and its associated value. Set this property to the argument value required.

property snucleotide

Sequences are nucleotide (boolean)

This controls the addition of the -snucleotide parameter and its associated value. Set this property to the argument value required.

property sprotein

Sequences are protein (boolean)

This controls the addition of the -sprotein parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**NeedleallCommandline**(*cmd='needleall'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the needleall program from EMBOSS.

__init__(*cmd='needleall'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property aformat

Display output in a different specified output format

This controls the addition of the -aformat parameter and its associated value. Set this property to the argument value required.

property asequence

First sequence to align

This controls the addition of the -asequence parameter and its associated value. Set this property to the argument value required.

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property brief

Display brief identity and similarity

This property controls the addition of the -brief switch, treat this property as a boolean.

property bsequence

Second sequence to align

This controls the addition of the -bsequence parameter and its associated value. Set this property to the argument value required.

property datafile

Matrix file

This controls the addition of the -datafile parameter and its associated value. Set this property to the argument value required.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property endextend

The score added to the end gap penalty for each base or residue in the end gap.

This controls the addition of the -endextend parameter and its associated value. Set this property to the argument value required.

property endopen

The score taken away when an end gap is created.

This controls the addition of the -endopen parameter and its associated value. Set this property to the argument value required.

property endweight

Apply And gap penalties

This controls the addition of the -endweight parameter and its associated value. Set this property to the argument value required.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property errorfile

Error file to be written to.

This controls the addition of the -errorfile parameter and its associated value. Set this property to the argument value required.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property gapextend

Gap extension penalty

This controls the addition of the -gapextend parameter and its associated value. Set this property to the argument value required.

property gapopen

Gap open penalty

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property minscore

Exclude alignments with scores below this threshold score.

This controls the addition of the -minscore parameter and its associated value. Set this property to the argument value required.

property nobrief

Display extended identity and similarity

This property controls the addition of the -nobrief switch, treat this property as a boolean.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property similarity

Display percent identity and similarity

This controls the addition of the -similarity parameter and its associated value. Set this property to the argument value required.

property snucleotide

Sequences are nucleotide (boolean)

This controls the addition of the -snucleotide parameter and its associated value. Set this property to the argument value required.

property sprotein

Sequences are protein (boolean)

This controls the addition of the -sprotein parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**StretcherCommandline**(*cmd='stretcher'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the stretcher program from EMBOSS.

__init__(*cmd='stretcher'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property aformat

Display output in a different specified output format

This controls the addition of the -aformat parameter and its associated value. Set this property to the argument value required.

property asequence

First sequence to align

This controls the addition of the -asequence parameter and its associated value. Set this property to the argument value required.

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property bsequence

Second sequence to align

This controls the addition of the -bsequence parameter and its associated value. Set this property to the argument value required.

property datafile

Matrix file

This controls the addition of the -datafile parameter and its associated value. Set this property to the argument value required.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property gapextend

Gap extension penalty

This controls the addition of the -gapextend parameter and its associated value. Set this property to the argument value required.

property gapopen

Gap open penalty

This controls the addition of the -gapopen parameter and its associated value. Set this property to the argument value required.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property snucleotide

Sequences are nucleotide (boolean)

This controls the addition of the -snucleotide parameter and its associated value. Set this property to the argument value required.

property sprotein

Sequences are protein (boolean)

This controls the addition of the -sprotein parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**FuzznucCommandline**(*cmd='fuzznuc'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the fuzznuc program from EMBOSS.

__init__(*cmd='fuzznuc'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property complement

Search complementary strand

This controls the addition of the -complement parameter and its associated value. Set this property to the argument value required.

property debug
: Write debug output to program.dbg.

  This property controls the addition of the -debug switch, treat this property as a boolean.

property die
: Report dying program messages.

  This property controls the addition of the -die switch, treat this property as a boolean.

property error
: Report errors.

  This property controls the addition of the -error switch, treat this property as a boolean.

property filter
: Read standard input, write standard output.

  This property controls the addition of the -filter switch, treat this property as a boolean.

property help
: Report command line options.

  More information on associated and general qualifiers can be found with -help -verbose

  This property controls the addition of the -help switch, treat this property as a boolean.

property options
: Prompt for standard and additional values.

  If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

  This property controls the addition of the -options switch, treat this property as a boolean.

property outfile
: Output filename

  This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property pattern
: Search pattern, using standard IUPAC one-letter codes

  This controls the addition of the -pattern parameter and its associated value. Set this property to the argument value required.

property pmismatch
: Number of mismatches

  This controls the addition of the -pmismatch parameter and its associated value. Set this property to the argument value required.

property rformat
: Specify the report format to output in.

  This controls the addition of the -rformat parameter and its associated value. Set this property to the argument value required.

property sequence
: Sequence database USA

  This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**FuzzproCommandline**(*cmd='fuzzpro'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the fuzzpro program from EMBOSS.

__init__(*cmd='fuzzpro'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property pattern

Search pattern, using standard IUPAC one-letter codes

This controls the addition of the -pattern parameter and its associated value. Set this property to the argument value required.

property pmismatch

Number of mismatches

This controls the addition of the -pmismatch parameter and its associated value. Set this property to the argument value required.

property rformat

Specify the report format to output in.

This controls the addition of the -rformat parameter and its associated value. Set this property to the argument value required.

property sequence

Sequence database USA

This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**Est2GenomeCommandline**(*cmd='est2genome'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the est2genome program from EMBOSS.

__init__(*cmd='est2genome'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property align

Show the alignment.

This controls the addition of the -align parameter and its associated value. Set this property to the argument value required.

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property best

You can print out all comparisons instead of just the best

This controls the addition of the -best parameter and its associated value. Set this property to the argument value required.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property est

EST sequence(s)

This controls the addition of the -est parameter and its associated value. Set this property to the argument value required.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property gappenalty

Cost for deleting a single base in either sequence, excluding introns

This controls the addition of the -gappenalty parameter and its associated value. Set this property to the argument value required.

property genome

Genomic sequence

This controls the addition of the -genome parameter and its associated value. Set this property to the argument value required.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property intronpenalty

Cost for an intron, independent of length.

This controls the addition of the -intronpenalty parameter and its associated value. Set this property to the argument value required.

property match

Score for matching two bases

This controls the addition of the -match parameter and its associated value. Set this property to the argument value required.

property minscore

Exclude alignments with scores below this threshold score.

This controls the addition of the -minscore parameter and its associated value. Set this property to the argument value required.

property mismatch

Cost for mismatching two bases

This controls the addition of the -mismatch parameter and its associated value. Set this property to the argument value required.

property mode

This determines the comparison mode. 'both', 'forward', or 'reverse'

This controls the addition of the -mode parameter and its associated value. Set this property to the argument value required.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property reverse

Reverse the orientation of the EST sequence

This controls the addition of the -reverse parameter and its associated value. Set this property to the argument value required.

property seed

Random number seed

This controls the addition of the -seed parameter and its associated value. Set this property to the argument value required.

property shuffle

Shuffle

This controls the addition of the -shuffle parameter and its associated value. Set this property to the argument value required.

property space

for linear-space recursion.

This controls the addition of the -space parameter and its associated value. Set this property to the argument value required.

property splice

Use donor and acceptor splice sites.

This controls the addition of the -splice parameter and its associated value. Set this property to the argument value required.

property splicepenalty

Cost for an intron, independent of length and starting/ending on donor-acceptor sites

This controls the addition of the -splicepenalty parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

property width

Alignment width

This controls the addition of the -width parameter and its associated value. Set this property to the argument value required.

class Bio.Emboss.Applications.**ETandemCommandline**(*cmd='etandem'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the etandem program from EMBOSS.

__init__(*cmd='etandem'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property debug
: Write debug output to program.dbg.

  This property controls the addition of the -debug switch, treat this property as a boolean.

property die
: Report dying program messages.

  This property controls the addition of the -die switch, treat this property as a boolean.

property error
: Report errors.

  This property controls the addition of the -error switch, treat this property as a boolean.

property filter
: Read standard input, write standard output.

  This property controls the addition of the -filter switch, treat this property as a boolean.

property help
: Report command line options.

  More information on associated and general qualifiers can be found with -help -verbose

  This property controls the addition of the -help switch, treat this property as a boolean.

property maxrepeat
: Maximum repeat size

  This controls the addition of the -maxrepeat parameter and its associated value. Set this property to the argument value required.

property minrepeat
: Minimum repeat size

  This controls the addition of the -minrepeat parameter and its associated value. Set this property to the argument value required.

property mismatch
: Allow N as a mismatch

  This controls the addition of the -mismatch parameter and its associated value. Set this property to the argument value required.

property options
: Prompt for standard and additional values.

  If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

  This property controls the addition of the -options switch, treat this property as a boolean.

property outfile
: Output filename

  This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property rformat
: Output report format

  This controls the addition of the -rformat parameter and its associated value. Set this property to the argument value required.

property sequence

Sequence

This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property threshold

Threshold score

This controls the addition of the -threshold parameter and its associated value. Set this property to the argument value required.

property uniform

Allow uniform consensus

This controls the addition of the -uniform parameter and its associated value. Set this property to the argument value required.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**EInvertedCommandline**(*cmd='einverted'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the einverted program from EMBOSS.

__init__(*cmd='einverted'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property gap

Gap penalty

This controls the addition of the -gap parameter and its associated value. Set this property to the argument value required.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property match

Match score

This controls the addition of the -match parameter and its associated value. Set this property to the argument value required.

property maxrepeat

Maximum separation between the start and end of repeat

This controls the addition of the -maxrepeat parameter and its associated value. Set this property to the argument value required.

property mismatch

Mismatch score

This controls the addition of the -mismatch parameter and its associated value. Set this property to the argument value required.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property sequence

Sequence

This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property threshold

Minimum score threshold

This controls the addition of the -threshold parameter and its associated value. Set this property to the argument value required.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**PalindromeCommandline**(*cmd='palindrome'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the palindrome program from EMBOSS.

__init__(*cmd='palindrome'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property gaplimit

Maximum gap between repeats

This controls the addition of the -gaplimit parameter and its associated value. Set this property to the argument value required.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property maxpallen

Maximum palindrome length

This controls the addition of the -maxpallen parameter and its associated value. Set this property to the argument value required.

property minpallen

Minimum palindrome length

This controls the addition of the -minpallen parameter and its associated value. Set this property to the argument value required.

property nummismatches

Number of mismatches allowed

This controls the addition of the -nummismatches parameter and its associated value. Set this property to the argument value required.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property overlap

Report overlapping matches

This controls the addition of the -overlap parameter and its associated value. Set this property to the argument value required.

property sequence

Sequence

This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.TranalignCommandline(*cmd='tranalign'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the tranalign program from EMBOSS.

__init__(*cmd='tranalign'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property asequence

Nucleotide sequences to be aligned.

This controls the addition of the -asequence parameter and its associated value. Set this property to the argument value required.

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property bsequence

Protein sequence alignment

This controls the addition of the -bsequence parameter and its associated value. Set this property to the argument value required.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property outseq

Output sequence file.

This controls the addition of the -outseq parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property table

Code to use

This controls the addition of the -table parameter and its associated value. Set this property to the argument value required.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**DiffseqCommandline**(*cmd='diffseq'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the diffseq program from EMBOSS.

__init__(*cmd='diffseq'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property aoutfeat

File for output of first sequence's features

This controls the addition of the -aoutfeat parameter and its associated value. Set this property to the argument value required.

property asequence

First sequence to compare

This controls the addition of the -asequence parameter and its associated value. Set this property to the argument value required.

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property boutfeat

File for output of second sequence's features

This controls the addition of the -boutfeat parameter and its associated value. Set this property to the argument value required.

property bsequence

Second sequence to compare

This controls the addition of the -bsequence parameter and its associated value. Set this property to the argument value required.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property rformat

Output report file format

This controls the addition of the -rformat parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

property wordsize

Word size to use for comparisons (10 default)

This controls the addition of the -wordsize parameter and its associated value. Set this property to the argument value required.

class Bio.Emboss.Applications.**IepCommandline**(*cmd='iep'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline for EMBOSS iep: calculated isoelectric point and charge.

Examples

```
>>> from Bio.Emboss.Applications import IepCommandline
>>> iep_cline = IepCommandline(sequence="proteins.faa",
...                            outfile="proteins.txt")
>>> print(iep_cline)
iep -outfile=proteins.txt -sequence=proteins.faa
```

You would typically run the command line with iep_cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='iep'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property amino

Number of N-termini

Integer 0 (default) or more.

This controls the addition of the -amino parameter and its associated value. Set this property to the argument value required.

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property carboxyl

Number of C-termini

Integer 0 (default) or more.

This controls the addition of the -carboxyl parameter and its associated value. Set this property to the argument value required.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property disulphides

Number of disulphide bridges

Integer 0 (default) or more.

This controls the addition of the -disulphides parameter and its associated value. Set this property to the argument value required.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property lysinemodified

Number of modified lysines

Integer 0 (default) or more.

This controls the addition of the -lysinemodified parameter and its associated value. Set this property to the argument value required.

property notermini

Exclude (True) or include (False) charge at N and C terminus.

This controls the addition of the -notermini parameter and its associated value. Set this property to the argument value required.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property sequence

Protein sequence(s) filename

This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.**SeqretCommandline**(*cmd='seqret'*, ***kwargs*)

Bases: _EmbossMinimalCommandLine

Commandline object for the seqret program from EMBOSS.

This tool allows you to interconvert between different sequence file formats (e.g. GenBank to FASTA). Combining Biopython's Bio.SeqIO module with seqret using a suitable intermediate file format can allow you to read/write to an even wider range of file formats.

This wrapper currently only supports the core functionality, things like feature tables (in EMBOSS 6.1.0 onwards) are not yet included.

__init__(*cmd='seqret'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property osformat

Output sequence(s) format (e.g. fasta, genbank)

This controls the addition of the -osformat parameter and its associated value. Set this property to the argument value required.

property outseq

Output sequence file.

This controls the addition of the -outseq parameter and its associated value. Set this property to the argument value required.

property sequence

Input sequence(s) filename

This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property sformat

Input sequence(s) format (e.g. fasta, genbank)

This controls the addition of the -sformat parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

class Bio.Emboss.Applications.SeqmatchallCommandline(*cmd='seqmatchall'*, ***kwargs*)

Bases: _EmbossCommandLine

Commandline object for the seqmatchall program from EMBOSS.

e.g. >>> cline = SeqmatchallCommandline(sequence="opuntia.fasta", outfile="opuntia.txt") >>> cline.auto = True >>> cline.wordsize = 18 >>> cline.aformat = "pair" >>> print(cline) seqmatchall -auto -outfile=opuntia.txt -sequence=opuntia.fasta -wordsize=18 -aformat=pair

__init__(*cmd='seqmatchall'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property aformat

Display output in a different specified output format

This controls the addition of the -aformat parameter and its associated value. Set this property to the argument value required.

property auto

Turn off prompts.

Automatic mode disables prompting, so we recommend you set this argument all the time when calling an EMBOSS tool from Biopython.

This property controls the addition of the -auto switch, treat this property as a boolean.

property debug

Write debug output to program.dbg.

This property controls the addition of the -debug switch, treat this property as a boolean.

property die

Report dying program messages.

This property controls the addition of the -die switch, treat this property as a boolean.

property error

Report errors.

This property controls the addition of the -error switch, treat this property as a boolean.

property filter

Read standard input, write standard output.

This property controls the addition of the -filter switch, treat this property as a boolean.

property help

Report command line options.

More information on associated and general qualifiers can be found with -help -verbose

This property controls the addition of the -help switch, treat this property as a boolean.

property options

Prompt for standard and additional values.

If you are calling an EMBOSS tool from within Biopython, we DO NOT recommend using this option.

This property controls the addition of the -options switch, treat this property as a boolean.

property outfile

Output filename

This controls the addition of the -outfile parameter and its associated value. Set this property to the argument value required.

property sequence

Readable set of sequences

This controls the addition of the -sequence parameter and its associated value. Set this property to the argument value required.

property stdout

Write standard output.

This property controls the addition of the -stdout switch, treat this property as a boolean.

property verbose

Report some/full command line options

This property controls the addition of the -verbose switch, treat this property as a boolean.

property warning

Report warnings.

This property controls the addition of the -warning switch, treat this property as a boolean.

property wordsize

Word size (Integer 2 or more, default 4)

This controls the addition of the -wordsize parameter and its associated value. Set this property to the argument value required.

Bio.Emboss.Primer3 module

Code to parse output from the EMBOSS eprimer3 program.

As elsewhere in Biopython there are two input functions, read and parse, for single record output and multi-record output. For primer3, a single record object is created for each target sequence and may contain multiple primers.

i.e. If you ran eprimer3 with a single target sequence, use the read function. If you ran eprimer3 with multiple targets, use the parse function to iterate over the retsults.

class Bio.Emboss.Primer3.**Record**

Bases: object

Represent information from a primer3 run finding primers.

Members:

- primers - list of Primer objects describing primer pairs for this target sequence.
- comments - the comment line(s) for the record

__init__()

Initialize the class.

class Bio.Emboss.Primer3.**Primers**

Bases: object

A primer set designed by Primer3.

Members:

- size - length of product, note you can use len(primer) as an alternative to primer.size
- forward_seq
- forward_start
- forward_length
- forward_tm
- forward_gc
- reverse_seq
- reverse_start
- reverse_length
- reverse_tm
- reverse_gc
- internal_seq
- internal_start
- internal_length
- internal_tm
- internal_gc

__init__()

Initialize the class.

__len__()

Length of the primer product (i.e. product size).

Bio.Emboss.Primer3.**parse**(*handle*)

Iterate over primer3 output as Bio.Emboss.Primer3.Record objects.

Bio.Emboss.Primer3.**read**(*handle*)

Parse primer3 output into a Bio.Emboss.Primer3.Record object.

This is for when there is one and only one target sequence. If designing primers for multiple sequences, use the parse function.

Bio.Emboss.PrimerSearch module

Code to interact with the primersearch program from EMBOSS.

class Bio.Emboss.PrimerSearch.**InputRecord**

Bases: object

Represent the input file into the primersearch program.

This makes it easy to add primer information and write it out to the simple primer file format.

__init__()

Initialize the class.

__str__()

Summarize the primersearch input record as a string.

add_primer_set(*primer_name*, *first_primer_seq*, *second_primer_seq*)

Add primer information to the record.

class Bio.Emboss.PrimerSearch.**OutputRecord**

Bases: object

Represent the information from a primersearch job.

amplifiers is a dictionary where the keys are the primer names and the values are a list of PrimerSearchAmplifier objects.

__init__()

Initialize the class.

class Bio.Emboss.PrimerSearch.**Amplifier**

Bases: object

Represent a single amplification from a primer.

__init__()

Initialize the class.

Bio.Emboss.PrimerSearch.**read**(*handle*)

Get output from primersearch into a PrimerSearchOutputRecord.

Module contents

Code to interact with the ever-so-useful EMBOSS programs.

28.1.11 Bio.Entrez package

Submodules

Bio.Entrez.Parser module

Parser for XML results returned by NCBI's Entrez Utilities.

This parser is used by the read() function in Bio.Entrez, and is not intended be used directly.

The question is how to represent an XML file as Python objects. Some XML files returned by NCBI look like lists, others look like dictionaries, and others look like a mix of lists and dictionaries.

My approach is to classify each possible element in the XML as a plain string, an integer, a list, a dictionary, or a structure. The latter is a dictionary where the same key can occur multiple times; in Python, it is represented as a dictionary where that key occurs once, pointing to a list of values found in the XML file.

The parser then goes through the XML and creates the appropriate Python object for each element. The different levels encountered in the XML are preserved on the Python side. So a subelement of a subelement of an element is a value in a dictionary that is stored in a list which is a value in some other dictionary (or a value in a list which itself belongs to a list which is a value in a dictionary, and so on). Attributes encountered in the XML are stored as a dictionary in a member .attributes of each element, and the tag name is saved in a member .tag.

To decide which kind of Python object corresponds to each element in the XML, the parser analyzes the DTD referred at the top of (almost) every XML file returned by the Entrez Utilities. This is preferred over a hand- written solution, since the number of DTDs is rather large and their contents may change over time. About half the code in this parser deals with parsing the DTD, and the other half with the XML itself.

class Bio.Entrez.Parser.**NoneElement**(*tag*, *attributes*, *key*)

Bases: object

NCBI Entrez XML element mapped to None.

__init__(*tag*, *attributes*, *key*)

Create a NoneElement.

__eq__(*other*)

Define equality with other None objects.

__ne__(*other*)

Define non-equality.

__repr__()

Return a string representation of the object.

__hash__ = None

class Bio.Entrez.Parser.**IntegerElement**(*value*, **args*, ***kwargs*)

Bases: int

NCBI Entrez XML element mapped to an integer.

static __new__(*cls*, *value*, **args*, ***kwargs*)

Create an IntegerElement.

__init__(*value*, *tag*, *attributes*, *key*)

Initialize an IntegerElement.

__repr__()

Return a string representation of the object.

class Bio.Entrez.Parser.**StringElement**(*value*, **args*, ***kwargs*)

Bases: str

NCBI Entrez XML element mapped to a string.

static __new__(*cls*, *value*, **args*, ***kwargs*)

Create a StringElement.

__init__(*value*, *tag*, *attributes*, *key*)

Initialize a StringElement.

__repr__()
Return a string representation of the object.

class Bio.Entrez.Parser.ListElement(*tag*, *attributes*, *allowed_tags*, *key=None*)
Bases: list

NCBI Entrez XML element mapped to a list.

__init__(*tag*, *attributes*, *allowed_tags*, *key=None*)
Create a ListElement.

__repr__()
Return a string representation of the object.

store(*value*)
Append an element to the list, checking tags.

class Bio.Entrez.Parser.DictionaryElement(*tag*, *attrs*, *allowed_tags*, *repeated_tags=None*, *key=None*)
Bases: dict

NCBI Entrez XML element mapped to a dictionaray.

__init__(*tag*, *attrs*, *allowed_tags*, *repeated_tags=None*, *key=None*)
Create a DictionaryElement.

__repr__()
Return a string representation of the object.

store(*value*)
Add an entry to the dictionary, checking tags.

class Bio.Entrez.Parser.OrderedListElement(*tag*, *attributes*, *allowed_tags*, *first_tag*, *key=None*)
Bases: list

NCBI Entrez XML element mapped to a list of lists.

OrderedListElement is used to describe a list of repeating elements such as A, B, C, A, B, C, A, B, C ... where each set of A, B, C forms a group. This is then stored as [[A, B, C], [A, B, C], [A, B, C], ...]

__init__(*tag*, *attributes*, *allowed_tags*, *first_tag*, *key=None*)
Create an OrderedListElement.

__repr__()
Return a string representation of the object.

store(*value*)
Append an element to the list, checking tags.

class Bio.Entrez.Parser.ErrorElement(*value*, **args*, ***kwargs*)
Bases: str

NCBI Entrez XML element containing an error message.

static __new__(*cls*, *value*, **args*, ***kwargs*)
Create an ErrorElement.

__init__(*value*, *tag*)
Initialize an ErrorElement.

__repr__()
Return the error message as a string.

exception Bio.Entrez.Parser.NotXMLError(*message*)
Bases: ValueError

Failed to parse file as XML.

__init__(*message*)
Initialize the class.

__str__()
Return a string summary of the exception.

exception Bio.Entrez.Parser.CorruptedXMLError(*message*)
Bases: ValueError

Corrupted XML.

__init__(*message*)
Initialize the class.

__str__()
Return a string summary of the exception.

exception Bio.Entrez.Parser.ValidationError(*name*)
Bases: ValueError

XML tag found which was not defined in the DTD.

Validating parsers raise this error if the parser finds a tag in the XML that is not defined in the DTD. Non-validating parsers do not raise this error. The Bio.Entrez.read and Bio.Entrez.parse functions use validating parsers by default (see those functions for more information).

__init__(*name*)
Initialize the class.

__str__()
Return a string summary of the exception.

class Bio.Entrez.Parser.DataHandlerMeta(**args*, ***kwargs*)
Bases: type

A metaclass is needed until Python supports @classproperty.

__init__(**args*, ***kwargs*)
Initialize the class.

property directory
Directory for caching XSD and DTD files.

__annotations__ = {}

class Bio.Entrez.Parser.DataHandler(*validate*, *escape*, *ignore_errors*)
Bases: object

Data handler for parsing NCBI XML from Entrez.

global_dtd_dir = '/home/pcock/repositories/biopython/Bio/Entrez/DTDs'

global_xsd_dir = '/home/pcock/repositories/biopython/Bio/Entrez/XSDs'

local_dtd_dir = '/home/pcock/.config/biopython/Bio/Entrez/DTDs'

local_xsd_dir = '/home/pcock/.config/biopython/Bio/Entrez/XSDs'

__init__(*validate*, *escape*, *ignore_errors*)
Create a DataHandler object.

read(*source*)
Set up the parser and let it read the XML results.

parse(*source*)
Set up the parser and let it read the XML results.

xmlDeclHandler(*version*, *encoding*, *standalone*)
Set XML handlers when an XML declaration is found.

handleMissingDocumentDefinition(*tag*, *attrs*)
Raise an Exception if neither a DTD nor an XML Schema is found.

startNamespaceDeclHandler(*prefix*, *uri*)
Handle start of an XML namespace declaration.

endNamespaceDeclHandler(*prefix*)
Handle end of an XML namespace declaration.

schemaHandler(*name*, *attrs*)
Process the XML schema (before processing the element).

startElementHandler(*tag*, *attrs*)
Handle start of an XML element.

startRawElementHandler(*name*, *attrs*)
Handle start of an XML raw element.

startSkipElementHandler(*name*, *attrs*)
Handle start of an XML skip element.

endStringElementHandler(*tag*)
Handle end of an XML string element.

endRawElementHandler(*name*)
Handle end of an XML raw element.

endSkipElementHandler(*name*)
Handle end of an XML skip element.

endErrorElementHandler(*tag*)
Handle end of an XML error element.

endElementHandler(*name*)
Handle end of an XML element.

endIntegerElementHandler(*tag*)
Handle end of an XML integer element.

characterDataHandlerRaw(*content*)
Handle character data as-is (raw).

characterDataHandlerEscape(*content*)

Handle character data by encoding it.

skipCharacterDataHandler(*content*)

Handle character data by skipping it.

parse_xsd(*root*)

Parse an XSD file.

elementDecl(*name*, *model*)

Call a call-back function for each element declaration in a DTD.

This is used for each element declaration in a DTD like:

```
<!ELEMENT       name          (...)>
```

The purpose of this function is to determine whether this element should be regarded as a string, integer, list, dictionary, structure, or error.

open_dtd_file(*filename*)

Open specified DTD file.

open_xsd_file(*filename*)

Open specified XSD file.

save_dtd_file(*filename*, *text*)

Save DTD file to cache.

save_xsd_file(*filename*, *text*)

Save XSD file to cache.

externalEntityRefHandler(*context*, *base*, *systemId*, *publicId*)

Handle external entity reference in order to cache DTD locally.

The purpose of this function is to load the DTD locally, instead of downloading it from the URL specified in the XML. Using the local DTD results in much faster parsing. If the DTD is not found locally, we try to download it. If new DTDs become available from NCBI, putting them in Bio/Entrez/DTDs will allow the parser to see them.

Module contents

Provides code to access NCBI over the WWW.

The main Entrez web page is available at: http://www.ncbi.nlm.nih.gov/Entrez/

Entrez Programming Utilities web page is available at: http://www.ncbi.nlm.nih.gov/books/NBK25501/

This module provides a number of functions like `efetch` (short for Entrez Fetch) which will return the data as a handle object. This is a standad interface used in Python for reading data from a file, or in this case a remote network connection, and provides methods like `.read()` or offers iteration over the contents line by line. See also "What the heck is a handle?" in the Biopython Tutorial and Cookbook: http://biopython.org/DIST/docs/tutorial/Tutorial.html http://biopython.org/DIST/docs/tutorial/Tutorial.pdf The handle returned by these functions can be either in text mode or in binary mode, depending on the data requested and the results returned by NCBI Entrez. Typically, XML data will be in binary mode while other data will be in text mode, as required by the downstream parser to parse the data.

Unlike a handle to a file on disk from the `open(filename)` function, which has a `.name` attribute giving the filename, the handles from `Bio.Entrez` all have a `.url` attribute instead giving the URL used to connect to the NCBI Entrez API.

The `epost`, `efetch`, and `esummary` tools take an "id" parameter which corresponds to one or more database UIDs (or accession.version identifiers in the case of sequence databases such as "nuccore" or "protein"). The Python value of the "id" keyword passed to these functions may be either a single ID as a string or integer or multiple IDs as an iterable of strings/integers. You may also pass a single string containing multiple IDs delimited by commas. The `elink` tool also accepts multiple IDs but the argument is handled differently than the other three. See that function's docstring for more information.

All the functions that send requests to the NCBI Entrez API will automatically respect the NCBI rate limit (of 3 requests per second without an API key, or 10 requests per second with an API key) and will automatically retry when encountering transient failures (i.e. connection failures or HTTP 5XX codes). By default, Biopython does a maximum of three tries before giving up, and sleeps for 15 seconds between tries. You can tweak these parameters by setting `Bio.Entrez.max_tries` and `Bio.Entrez.sleep_between_tries`.

The Entrez module also provides an XML parser which takes a handle as input.

Variables:

- email Set the Entrez email parameter (default is not set).
- tool Set the Entrez tool parameter (default is `biopython`).
- api_key Personal API key from NCBI. If not set, only 3 queries per second are allowed. 10 queries per seconds otherwise with a valid API key.
- max_tries Configures how many times failed requests will be automatically retried on error (default is 3).
- sleep_between_tries The delay, in seconds, before retrying a request on error (default is 15).

Functions:

- efetch Retrieves records in the requested format from a list of one or more primary IDs or from the user's environment
- epost Posts a file containing a list of primary IDs for future use in the user's environment to use with subsequent search strategies
- esearch Searches and retrieves primary IDs (for use in EFetch, ELink, and ESummary) and term translations and optionally retains results for future use in the user's environment.
- elink Checks for the existence of an external or Related Articles link from a list of one or more primary IDs. Retrieves primary IDs and relevancy scores for links to Entrez databases or Related Articles; creates a hyperlink to the primary LinkOut provider for a specific ID and database, or lists LinkOut URLs and Attributes for multiple IDs.
- einfo Provides field index term counts, last update, and available links for each database.
- esummary Retrieves document summaries from a list of primary IDs or from the user's environment.
- egquery Provides Entrez database counts in XML for a single search using Global Query.
- espell Retrieves spelling suggestions.
- ecitmatch Retrieves PubMed IDs (PMIDs) that correspond to a set of input citation strings.
- read Parses the XML results returned by any of the above functions. Alternatively, the XML data can be read from a file opened in binary mode. Typical usage is:

```
>>> from Bio import Entrez
>>> Entrez.email = "Your.Name.Here@example.org"
>>> handle = Entrez.einfo() # or esearch, efetch, ...
>>> record = Entrez.read(handle)
>>> handle.close()
```

  where record is now a Python dictionary or list.

- parse Parses the XML results returned by those of the above functions which can return multiple records - such as efetch, esummary and elink. Typical usage is:

```
>>> handle = Entrez.esummary(db="pubmed", id="19304878,14630660", retmode="xml")
>>> records = Entrez.parse(handle)
>>> for record in records:
...     # each record is a Python dictionary or list.
...     print(record['Title'])
Biopython: freely available Python tools for computational molecular biology and␣
↪bioinformatics.
PDB file parser and structure class implemented in Python.
>>> handle.close()
```

  This function is appropriate only if the XML file contains multiple records, and is particular useful for large files.

- _open Internally used function.

Bio.Entrez.**epost**(*db*, ***keywds*)

Post a file of identifiers for future use.

Posts a file containing a list of UIs for future use in the user's environment to use with subsequent search strategies.

See the online documentation for an explanation of the parameters: http://www.ncbi.nlm.nih.gov/books/NBK25499/#chapter4.EPost

Returns
Handle to the results.

Raises
urllib.error.URLError – If there's a network error.

Bio.Entrez.**efetch**(*db*, ***keywords*)

Fetch Entrez results which are returned as a handle.

EFetch retrieves records in the requested format from a list or set of one or more UIs or from user's environment.

See the online documentation for an explanation of the parameters: http://www.ncbi.nlm.nih.gov/books/NBK25499/#chapter4.EFetch

Short example:

```
>>> from Bio import Entrez
>>> Entrez.email = "Your.Name.Here@example.org"
>>> handle = Entrez.efetch(db="nucleotide", id="AY851612", rettype="gb", retmode=
↪"text")
>>> print(handle.readline().strip())
LOCUS       AY851612                892 bp    DNA     linear   PLN 10-APR-2007
>>> handle.close()
```

This will automatically use an HTTP POST rather than HTTP GET if there are over 200 identifiers as recommended by the NCBI.

Warning: The NCBI changed the default retmode in Feb 2012, so many databases which previously returned text output now give XML.

Returns
Handle to the results.

Raises
urllib.error.URLError – If there's a network error.

Bio.Entrez.**esearch**(*db*, *term*, ***keywds*)

Run an Entrez search and return a handle to the results.

ESearch searches and retrieves primary IDs (for use in EFetch, ELink and ESummary) and term translations, and optionally retains results for future use in the user's environment.

See the online documentation for an explanation of the parameters: http://www.ncbi.nlm.nih.gov/books/NBK25499/#chapter4.ESearch

Short example:

```
>>> from Bio import Entrez
>>> Entrez.email = "Your.Name.Here@example.org"
>>> handle = Entrez.esearch(
...     db="nucleotide", retmax=10, idtype="acc",
...     term="opuntia[ORGN] accD 2007[Publication Date]"
... )
...
>>> record = Entrez.read(handle)
>>> handle.close()
>>> int(record["Count"]) >= 2
True
>>> "EF590893.1" in record["IdList"]
True
>>> "EF590892.1" in record["IdList"]
True
```

Returns
Handle to the results, which are always in XML format.

Raises
urllib.error.URLError – If there's a network error.

Bio.Entrez.**elink**(***keywds*)

Check for linked external articles and return a handle.

ELink checks for the existence of an external or Related Articles link from a list of one or more primary IDs; retrieves IDs and relevancy scores for links to Entrez databases or Related Articles; creates a hyperlink to the primary LinkOut provider for a specific ID and database, or lists LinkOut URLs and attributes for multiple IDs.

See the online documentation for an explanation of the parameters: http://www.ncbi.nlm.nih.gov/books/NBK25499/#chapter4.ELink

Note that ELink treats the "id" parameter differently than the other tools when multiple values are given. You should generally pass multiple UIDs as a list of strings or integers. This will provide a "one-to-one" mapping from source database UIDs to destination database UIDs in the result. If multiple source UIDs are passed as a single comma-delimited string all destination UIDs will be mixed together in the result.

This example finds articles related to the Biopython application note's entry in the PubMed database:

```
>>> from Bio import Entrez
>>> Entrez.email = "Your.Name.Here@example.org"
>>> pmid = "19304878"
>>> handle = Entrez.elink(dbfrom="pubmed", id=pmid, linkname="pubmed_pubmed")
>>> record = Entrez.read(handle)
>>> handle.close()
>>> print(record[0]["LinkSetDb"][0]["LinkName"])
```

(continues on next page)

(continued from previous page)

```
pubmed_pubmed
>>> linked = [link["Id"] for link in record[0]["LinkSetDb"][0]["Link"]]
>>> "14630660" in linked
True
```

This is explained in much more detail in the Biopython Tutorial.

Returns
Handle to the results, by default in XML format.

Raises
urllib.error.URLError – If there's a network error.

Bio.Entrez.**einfo**(***keywds*)

Return a summary of the Entrez databases as a results handle.

EInfo provides field names, index term counts, last update, and available links for each Entrez database.

See the online documentation for an explanation of the parameters: http://www.ncbi.nlm.nih.gov/books/NBK25499/#chapter4.EInfo

Short example:

```
>>> from Bio import Entrez
>>> Entrez.email = "Your.Name.Here@example.org"
>>> record = Entrez.read(Entrez.einfo())
>>> 'pubmed' in record['DbList']
True
```

Returns
Handle to the results, by default in XML format.

Raises
urllib.error.URLError – If there's a network error.

Bio.Entrez.**esummary**(***keywds*)

Retrieve document summaries as a results handle.

ESummary retrieves document summaries from a list of primary IDs or from the user's environment.

See the online documentation for an explanation of the parameters: http://www.ncbi.nlm.nih.gov/books/NBK25499/#chapter4.ESummary

This example discovers more about entry 19923 in the structure database:

```
>>> from Bio import Entrez
>>> Entrez.email = "Your.Name.Here@example.org"
>>> handle = Entrez.esummary(db="structure", id="19923")
>>> record = Entrez.read(handle)
>>> handle.close()
>>> print(record[0]["Id"])
19923
>>> print(record[0]["PdbDescr"])
CRYSTAL STRUCTURE OF E. COLI ACONITASE B
```

Returns
Handle to the results, by default in XML format.

Raises
urllib.error.URLError – If there's a network error.

Bio.Entrez.**egquery**(***keywds*)

Provide Entrez database counts for a global search (DEPRECATED).

EGQuery provided Entrez database counts in XML for a single search using Global Query. However, the NCBI are no longer maintaining this function and suggest using esearch on each database of interest.

See the online documentation for an explanation of the parameters: http://www.ncbi.nlm.nih.gov/books/NBK25499/#chapter4.EGQuery

This quick example based on a longer version from the Biopython Tutorial just checks there are over 60 matches for 'Biopython' in PubMedCentral:

```
>>> from Bio import Entrez
>>> Entrez.email = "Your.Name.Here@example.org"
>>> handle = Entrez.egquery(term="biopython")
>>> record = Entrez.read(handle)
>>> handle.close()
>>> for row in record["eGQueryResult"]:
...     if "pmc" in row["DbName"]:
...         print(int(row["Count"]) > 60)
True
```

Returns
Handle to the results, by default in XML format.

Raises
urllib.error.URLError – If there's a network error.

Bio.Entrez.**espell**(***keywds*)

Retrieve spelling suggestions as a results handle.

ESpell retrieves spelling suggestions, if available.

See the online documentation for an explanation of the parameters: http://www.ncbi.nlm.nih.gov/books/NBK25499/#chapter4.ESpell

Short example:

```
>>> from Bio import Entrez
>>> Entrez.email = "Your.Name.Here@example.org"
>>> record = Entrez.read(Entrez.espell(term="biopythooon"))
>>> print(record["Query"])
biopythooon
>>> print(record["CorrectedQuery"])
biopython
```

Returns
Handle to the results, by default in XML format.

Raises
urllib.error.URLError – If there's a network error.

Bio.Entrez.**ecitmatch**(***keywds*)

Retrieve PMIDs for input citation strings, returned as a handle.

ECitMatch retrieves PubMed IDs (PMIDs) that correspond to a set of input citation strings.

See the online documentation for an explanation of the parameters: http://www.ncbi.nlm.nih.gov/books/NBK25499/#chapter4.ECitMatch

Short example:

```
>>> from Bio import Entrez
>>> Entrez.email = "Your.Name.Here@example.org"
>>> citation_1 = {"journal_title": "proc natl acad sci u s a",
...               "year": "1991", "volume": "88", "first_page": "3248",
...               "author_name": "mann bj", "key": "citation_1"}
>>> handle = Entrez.ecitmatch(db="pubmed", bdata=[citation_1])
>>> print(handle.read().strip().split("|"))
['proc natl acad sci u s a', '1991', '88', '3248', 'mann bj', 'citation_1', '2014248
↪']
>>> handle.close()
```

Returns
Handle to the results, by default in plain text.

Raises
urllib.error.URLError – If there's a network error.

Bio.Entrez.**read**(*source*, *validate=True*, *escape=False*, *ignore_errors=False*)

Parse an XML file from the NCBI Entrez Utilities into python objects.

This function parses an XML file created by NCBI's Entrez Utilities, returning a multilevel data structure of Python lists and dictionaries. Most XML files returned by NCBI's Entrez Utilities can be parsed by this function, provided its DTD is available. Biopython includes the DTDs for most commonly used Entrez Utilities.

The argument `source` must be a file or file-like object opened in binary mode, or a filename. The parser detects the encoding from the XML file, and uses it to convert all text in the XML to the correct Unicode string. The functions in Bio.Entrez to access NCBI Entrez will automatically return XML data in binary mode. For files, use mode "rb" when opening the file, as in

```
>>> from Bio import Entrez
>>> path = "Entrez/esearch1.xml"
>>> stream = open(path, "rb")  # opened in binary mode
>>> record = Entrez.read(stream)
>>> print(record['QueryTranslation'])
biopython[All Fields]
>>> stream.close()
```

Alternatively, you can use the filename directly, as in

```
>>> record = Entrez.read(path)
>>> print(record['QueryTranslation'])
biopython[All Fields]
```

which is safer, as the file stream will automatically be closed after the record has been read, or if an error occurs.

If validate is True (default), the parser will validate the XML file against the DTD, and raise an error if the XML file contains tags that are not represented in the DTD. If validate is False, the parser will simply skip such tags.

If escape is True, all characters that are not valid HTML are replaced by HTML escape characters to guarantee that the returned strings are valid HTML fragments. For example, a less-than sign (<) is replaced by <. If escape is False (default), the string is returned as is.

If ignore_errors is False (default), any error messages in the XML file will raise a RuntimeError. If ignore_errors is True, error messages will be stored as ErrorElement items, without raising an exception.

Whereas the data structure seems to consist of generic Python lists, dictionaries, strings, and so on, each of these is actually a class derived from the base type. This allows us to store the attributes (if any) of each element in a dictionary my_element.attributes, and the tag name in my_element.tag.

Bio.Entrez.**parse**(*source*, *validate=True*, *escape=False*, *ignore_errors=False*)

Parse an XML file from the NCBI Entrez Utilities into python objects.

This function parses an XML file created by NCBI's Entrez Utilities, returning a multilevel data structure of Python lists and dictionaries. This function is suitable for XML files that (in Python) can be represented as a list of individual records. Whereas 'read' reads the complete file and returns a single Python list, 'parse' is a generator function that returns the records one by one. This function is therefore particularly useful for parsing large files.

Most XML files returned by NCBI's Entrez Utilities can be parsed by this function, provided its DTD is available. Biopython includes the DTDs for most commonly used Entrez Utilities.

The argument `source` must be a file or file-like object opened in binary mode, or a filename. The parser detects the encoding from the XML file, and uses it to convert all text in the XML to the correct Unicode string. The functions in Bio.Entrez to access NCBI Entrez will automatically return XML data in binary mode. For files, use mode "rb" when opening the file, as in

```
>>> from Bio import Entrez
>>> path = "Entrez/pubmed1.xml"
>>> stream = open(path, "rb")  # opened in binary mode
>>> records = Entrez.parse(stream)
>>> for record in records:
...     print(record['MedlineCitation']['Article']['Journal']['Title'])
...
Social justice (San Francisco, Calif.)
Biochimica et biophysica acta
>>> stream.close()
```

Alternatively, you can use the filename directly, as in

```
>>> records = Entrez.parse(path)
>>> for record in records:
...     print(record['MedlineCitation']['Article']['Journal']['Title'])
...
Social justice (San Francisco, Calif.)
Biochimica et biophysica acta
```

which is safer, as the file stream will automatically be closed after all the records have been read, or if an error occurs.

If validate is True (default), the parser will validate the XML file against the DTD, and raise an error if the XML file contains tags that are not represented in the DTD. If validate is False, the parser will simply skip such tags.

If escape is True, all characters that are not valid HTML are replaced by HTML escape characters to guarantee that the returned strings are valid HTML fragments. For example, a less-than sign (<) is replaced by <. If escape is False (default), the string is returned as is.

If ignore_errors is False (default), any error messages in the XML file will raise a RuntimeError. If ignore_errors is True, error messages will be stored as ErrorElement items, without raising an exception.

Whereas the data structure seems to consist of generic Python lists, dictionaries, strings, and so on, each of these is actually a class derived from the base type. This allows us to store the attributes (if any) of each element in a dictionary my_element.attributes, and the tag name in my_element.tag.

28.1.12 Bio.ExPASy package

Submodules

Bio.ExPASy.Enzyme module

Parse the enzyme.dat file from Enzyme at ExPASy.

See https://www.expasy.org/enzyme/

Tested with the release of 03-Mar-2009.

Functions:

- read Reads a file containing one ENZYME entry
- parse Reads a file containing multiple ENZYME entries

Classes:

- Record Holds ENZYME data.

`Bio.ExPASy.Enzyme.parse(handle)`

Parse ENZYME records.

This function is for parsing ENZYME files containing multiple records.

Arguments:

- handle - handle to the file.

`Bio.ExPASy.Enzyme.read(handle)`

Read one ENZYME record.

This function is for parsing ENZYME files containing exactly one record.

Arguments:

- handle - handle to the file.

`class Bio.ExPASy.Enzyme.Record`

Bases: `dict`

Holds information from an ExPASy ENZYME record as a Python dictionary.

Each record contains the following keys:

- ID: EC number
- DE: Recommended name
- AN: Alternative names (if any)
- CA: Catalytic activity
- CF: Cofactors (if any)

- PR: Pointers to any Prosite documentation entries that correspond to the enzyme
- DR: Pointers to any Swiss-Prot protein sequence entries that correspond to the enzyme
- CC: Comments

`__init__`()
Initialize the class.

`__repr__`()
Return the canonical string representation of the Record object.

`__str__`()
Return a readable string representation of the Record object.

Bio.ExPASy.Prodoc module

Code to work with the prosite.doc file from Prosite.

See https://www.expasy.org/prosite/

Tested with:

- Release 15.0, July 1998
- Release 16.0, July 1999
- Release 20.22, 13 November 2007
- Release 20.43, 10 February 2009

Functions:

- read Read a Prodoc file containing exactly one Prodoc entry.
- parse Iterates over entries in a Prodoc file.

Classes:

- Record Holds Prodoc data.
- Reference Holds data from a Prodoc reference.

Bio.ExPASy.Prodoc.**read**(*handle*)
Read in a record from a file with exactly one Prodoc record.

Bio.ExPASy.Prodoc.**parse**(*handle*)
Iterate over the records in a Prodoc file.

class Bio.ExPASy.Prodoc.**Record**
Bases: `object`

Holds information from a Prodoc record.

Attributes:

- accession Accession number of the record.
- prosite_refs List of tuples (prosite accession, prosite name).
- text Free format text.
- references List of reference objects.

__init__()
Initialize the class.

class Bio.ExPASy.Prodoc.Reference
Bases: object

Holds information from a Prodoc citation.

Attributes:

- number Number of the reference. (string)
- authors Names of the authors.
- citation Describes the citation.

__init__()
Initialize the class.

Bio.ExPASy.Prosite module

Parser for the prosite dat file from Prosite at ExPASy.

See https://www.expasy.org/prosite/

Tested with:

- Release 20.43, 10-Feb-2009
- Release 2017_03 of 15-Mar-2017.

Functions:

- read Reads a Prosite file containing one Prosite record
- parse Iterates over records in a Prosite file.

Classes:

- Record Holds Prosite data.

Bio.ExPASy.Prosite.parse(*handle*)
Parse Prosite records.

This function is for parsing Prosite files containing multiple records.

Arguments:

- handle - handle to the file.

Bio.ExPASy.Prosite.read(*handle*)
Read one Prosite record.

This function is for parsing Prosite files containing exactly one record.

Arguments:

- handle - handle to the file.

class Bio.ExPASy.Prosite.Record
Bases: object

Holds information from a Prosite record.

Main attributes:

- name ID of the record. e.g. ADH_ZINC
- type Type of entry. e.g. PATTERN, MATRIX, or RULE
- accession e.g. PS00387
- created Date the entry was created. (MMM-YYYY for releases before January 2017, DD-MMM-YYYY since January 2017)
- data_update Date the 'primary' data was last updated.
- info_update Date data other than 'primary' data was last updated.
- pdoc ID of the PROSITE DOCumentation.
- description Free-format description.
- pattern The PROSITE pattern. See docs.
- matrix List of strings that describes a matrix entry.
- rules List of rule definitions (from RU lines). (strings)
- prorules List of prorules (from PR lines). (strings)

NUMERICAL RESULTS:

- nr_sp_release SwissProt release.
- nr_sp_seqs Number of seqs in that release of Swiss-Prot. (int)
- nr_total Number of hits in Swiss-Prot. tuple of (hits, seqs)
- nr_positive True positives. tuple of (hits, seqs)
- nr_unknown Could be positives. tuple of (hits, seqs)
- nr_false_pos False positives. tuple of (hits, seqs)
- nr_false_neg False negatives. (int)
- nr_partial False negatives, because they are fragments. (int)

COMMENTS:

- cc_taxo_range Taxonomic range. See docs for format
- cc_max_repeat Maximum number of repetitions in a protein
- cc_site Interesting site. list of tuples (pattern pos, desc.)
- cc_skip_flag Can this entry be ignored?
- cc_matrix_type
- cc_scaling_db
- cc_author
- cc_ft_key
- cc_ft_desc
- cc_version version number (introduced in release 19.0)

The following are all lists if tuples (swiss-prot accession, swiss-prot name).

DATA BANK REFERENCES:

- dr_positive

- dr_false_neg
- dr_false_pos
- dr_potential Potential hits, but fingerprint region not yet available.
- dr_unknown Could possibly belong
- pdb_structs List of PDB entries.

__init__()

Initialize the class.

Bio.ExPASy.ScanProsite module

Code for calling and parsing ScanProsite from ExPASy.

class Bio.ExPASy.ScanProsite.**Record**

Bases: list

Represents search results returned by ScanProsite.

This record is a list containing the search results returned by ScanProsite. The record also contains the data members n_match, n_seq, capped, and warning.

__init__()

Initialize the class.

Bio.ExPASy.ScanProsite.**scan**(*seq=''*, *mirror='https://prosite.expasy.org'*, *output='xml'*, ***keywords*)

Execute a ScanProsite search.

Arguments:

- **mirror: The ScanProsite mirror to be used**
  (default: https://prosite.expasy.org).
- **seq: The query sequence, or UniProtKB (Swiss-Prot,**
  TrEMBL) accession
- **output: Format of the search results**
  (default: xml)

Further search parameters can be passed as keywords; see the documentation for programmatic access to ScanProsite at https://prosite.expasy.org/scanprosite/scanprosite_doc.html for a description of such parameters.

This function returns a handle to the search results returned by ScanProsite. Search results in the XML format can be parsed into a Python object, by using the Bio.ExPASy.ScanProsite.read function.

Bio.ExPASy.ScanProsite.**read**(*handle*)

Parse search results returned by ScanProsite into a Python object.

class Bio.ExPASy.ScanProsite.**Parser**

Bases: ExpatParser

Process the result from a ScanProsite search (PRIVATE).

__init__()

Initialize the class.

feed(*data*, *isFinal=0*)

Raise an Error if plain text is received in the data.

This is to show the Error messages returned by ScanProsite.

class Bio.ExPASy.ScanProsite.**ContentHandler**

Bases: ContentHandler

Process and fill in the records, results of the search (PRIVATE).

integers = ('start', 'stop')

strings = ('sequence_ac', 'sequence_id', 'sequence_db', 'signature_ac', 'level', 'level_tag')

__init__()

Initialize the class.

startElement(*name*, *attrs*)

Define the beginning of a record and stores the search record.

endElement(*name*)

Define the end of the search record.

characters(*content*)

Store the record content.

__annotations__ = {}

Bio.ExPASy.cellosaurus module

Parser for the cellosaurus.txt file from ExPASy.

See https://web.expasy.org/cellosaurus/

Tested with the release of Version 18 (July 2016).

Functions:

- read Reads a file containing one cell line entry
- parse Reads a file containing multiple cell line entries

Classes:

- Record Holds cell line data.

Examples

This example downloads the Cellosaurus database and parses it. Note that urlopen returns a stream of bytes, while the parser expects a stream of plain string, so we use TextIOWrapper to convert bytes to string using the UTF-8 encoding. This is not needed if you download the cellosaurus.txt file in advance and open it (see the comment below).

```
>>> from urllib.request import urlopen
>>> from io import TextIOWrapper
>>> from Bio.ExPASy import cellosaurus
>>> url = "ftp://ftp.expasy.org/databases/cellosaurus/cellosaurus.txt"
>>> bytestream = urlopen(url)
```

(continues on next page)

(continued from previous page)

```
>>> textstream = TextIOWrapper(bytestream, "UTF-8")
>>> # alternatively, use
>>> # textstream = open("cellosaurus.txt")
>>> # if you downloaded the cellosaurus.txt file in advance.
>>> records = cellosaurus.parse(textstream)
>>> for record in records:
...     if 'Homo sapiens' in record['OX'][0]:
...         print(record['ID'])
...
#15310-LN
#W7079
(L)PC6
0.5alpha
...
```

Bio.ExPASy.cellosaurus.**parse**(*handle*)

Parse cell line records.

This function is for parsing cell line files containing multiple records.

Arguments:

- handle - handle to the file.

Bio.ExPASy.cellosaurus.**read**(*handle*)

Read one cell line record.

This function is for parsing cell line files containing exactly one record.

Arguments:

- handle - handle to the file.

class Bio.ExPASy.cellosaurus.**Record**

Bases: `dict`

Holds information from an ExPASy Cellosaurus record as a Python dictionary.

Each record contains the following keys:

Line code	Content	Occurrence in an entry
ID	Identifier (cell line name)	Once; starts an entry
AC	Accession (CVCL_xxxx)	Once
AS	Secondary accession number(s)	Optional; once
SY	Synonyms	Optional; once
DR	Cross-references	Optional; once or more
RX	References identifiers	Optional: once or more
WW	Web pages	Optional; once or more
CC	Comments	Optional; once or more
ST	STR profile data	Optional; twice or more
DI	Diseases	Optional; once or more
OX	Species of origin	Once or more
HI	Hierarchy	Optional; once or more
OI	Originate from same individual	Optional; once or more
SX	Sex of cell	Optional; once
AG	Age of donor at sampling	Optional; once
CA	Category	Once
DT	Date (entry history)	Once
//	Terminator	Once; ends an entry

__init__()

Initialize the class.

__repr__()

Return the canonical string representation of the Record object.

__str__()

Return a readable string representation of the Record object.

Module contents

Code to access resources at ExPASy over the WWW.

See https://www.expasy.org/

Functions:

- get_prodoc_entry Interface to the get-prodoc-entry CGI script.
- get_prosite_entry Interface to the get-prosite-entry CGI script.
- get_prosite_raw Interface to the get-prosite-raw CGI script.
- get_sprot_raw Interface to the get-sprot-raw CGI script.

Bio.ExPASy.**get_prodoc_entry**(*id*, *cgi='https://prosite.expasy.org/cgi-bin/prosite/get-prodoc-entry'*)

Get a text handle to a PRODOC entry at ExPASy in HTML format.

```
>>> from Bio import ExPASy
>>> import os
>>> with ExPASy.get_prodoc_entry('PDOC00001') as in_handle:
...     html = in_handle.read()
...
>>> with open("myprodocrecord.html", "w") as out_handle:
```

(continues on next page)

(continued from previous page)

```
...     length = out_handle.write(html)
...
>>> os.remove("myprodocrecord.html")  # tidy up
```

For a non-existing key XXX, ExPASy returns an HTML-formatted page containing this text: 'There is currently no PROSITE entry for'

Bio.ExPASy.**get_prosite_entry**(*id*, *cgi='https://prosite.expasy.org/cgi-bin/prosite/get-prosite-entry'*)

Get a text handle to a PROSITE entry at ExPASy in HTML format.

```
>>> from Bio import ExPASy
>>> import os
>>> with ExPASy.get_prosite_entry('PS00001') as in_handle:
...     html = in_handle.read()
...
>>> with open("myprositerecord.html", "w") as out_handle:
...     length = out_handle.write(html)
...
>>> os.remove("myprositerecord.html")  # tidy up
```

For a non-existing key XXX, ExPASy returns an HTML-formatted page containing this text: 'There is currently no PROSITE entry for'

Bio.ExPASy.**get_prosite_raw**(*id*, *cgi=None*)

Get a text handle to a raw PROSITE or PRODOC record at ExPASy.

The cgi argument is deprecated due to changes in the ExPASy website.

```
>>> from Bio import ExPASy
>>> from Bio.ExPASy import Prosite
>>> with ExPASy.get_prosite_raw('PS00001') as handle:
...     record = Prosite.read(handle)
...
>>> print(record.accession)
PS00001
```

This function raises a ValueError if the identifier does not exist:

```
>>> handle = ExPASy.get_prosite_raw("DOES_NOT_EXIST")
Traceback (most recent call last):
    ...
ValueError: Failed to find entry 'DOES_NOT_EXIST' on ExPASy
```

Bio.ExPASy.**get_sprot_raw**(*id*)

Get a text handle to a raw SwissProt entry at ExPASy.

For an ID of XXX, fetches http://www.uniprot.org/uniprot/XXX.txt (as per the https://www.expasy.org/expasy_urls.html documentation).

```
>>> from Bio import ExPASy
>>> from Bio import SwissProt
>>> with ExPASy.get_sprot_raw("O23729") as handle:
...     record = SwissProt.read(handle)
...
```

(continues on next page)

(continued from previous page)

```
>>> print(record.entry_name)
CHS3_BROFI
```

This function raises a ValueError if the identifier does not exist:

```
>>> ExPASy.get_sprot_raw("DOES_NOT_EXIST")
Traceback (most recent call last):
    ...
ValueError: Failed to find SwissProt entry 'DOES_NOT_EXIST'
```

28.1.13 Bio.GenBank package

Submodules

Bio.GenBank.Record module

Hold GenBank data in a straightforward format.

Classes:

- Record - All of the information in a GenBank record.
- Reference - hold reference data for a record.
- Feature - Hold the information in a Feature Table.
- Qualifier - Qualifiers on a Feature.

class Bio.GenBank.Record.**Record**

Bases: object

Hold GenBank information in a format similar to the original record.

The Record class is meant to make data easy to get to when you are just interested in looking at GenBank data.

Attributes:

- locus - The name specified after the LOCUS keyword in the GenBank record. This may be the accession number, or a clone id or something else.
- size - The size of the record.
- residue_type - The type of residues making up the sequence in this record. Normally something like RNA, DNA or PROTEIN, but may be as esoteric as 'ss-RNA circular'.
- data_file_division - The division this record is stored under in GenBank (ie. PLN -> plants; PRI -> humans, primates; BCT -> bacteria...)
- date - The date of submission of the record, in a form like '28-JUL-1998'
- accession - list of all accession numbers for the sequence.
- nid - Nucleotide identifier number.
- pid - Proteint identifier number
- version - The accession number + version (ie. AB01234.2)
- db_source - Information about the database the record came from
- gi - The NCBI gi identifier for the record.

- keywords - A list of keywords related to the record.
- segment - If the record is one of a series, this is info about which segment this record is (something like '1 of 6').
- source - The source of material where the sequence came from.
- organism - The genus and species of the organism (ie. 'Homo sapiens')
- taxonomy - A listing of the taxonomic classification of the organism, starting general and getting more specific.
- references - A list of Reference objects.
- comment - Text with any kind of comment about the record.
- features - A listing of Features making up the feature table.
- base_counts - A string with the counts of bases for the sequence.
- origin - A string specifying info about the origin of the sequence.
- sequence - A string with the sequence itself.
- contig - A string of location information for a CONTIG in a RefSeq file
- project - The genome sequencing project numbers (will be replaced by the dblink cross-references in 2009).
- dblinks - The genome sequencing project number(s) and other links. (will replace the project information in 2009).

GB_LINE_LENGTH = 79

GB_BASE_INDENT = 12

GB_FEATURE_INDENT = 21

GB_INTERNAL_INDENT = 2

GB_OTHER_INTERNAL_INDENT = 3

GB_FEATURE_INTERNAL_INDENT = 5

GB_SEQUENCE_INDENT = 9

BASE_FORMAT = '%-12s'

INTERNAL_FORMAT = ' %-10s'

OTHER_INTERNAL_FORMAT = ' %-9s'

BASE_FEATURE_FORMAT = '%-21s'

INTERNAL_FEATURE_FORMAT = ' %-16s'

SEQUENCE_FORMAT = '%9s'

__init__()

Initialize the class.

__str__()

Provide a GenBank formatted output option for a Record.

The objective of this is to provide an easy way to read in a GenBank record, modify it somehow, and then output it in 'GenBank format.' We are striving to make this work so that a parsed Record that is output using this function will look exactly like the original record.

Much of the output is based on format description info at:

ftp://ncbi.nlm.nih.gov/genbank/gbrel.txt

class Bio.GenBank.Record.**Reference**

Bases: object

Hold information from a GenBank reference.

Attributes:

- number - The number of the reference in the listing of references.
- bases - The bases in the sequence the reference refers to.
- authors - String with all of the authors.
- consrtm - Consortium the authors belong to.
- title - The title of the reference.
- journal - Information about the journal where the reference appeared.
- medline_id - The medline id for the reference.
- pubmed_id - The pubmed_id for the reference.
- remark - Free-form remarks about the reference.

__init__()

Initialize the class.

__str__()

Convert the reference to a GenBank format string.

class Bio.GenBank.Record.**Feature**(*key=''*, *location=''*)

Bases: object

Hold information about a Feature in the Feature Table of GenBank record.

Attributes:

- key - The key name of the feature (ie. source)
- location - The string specifying the location of the feature.
- qualifiers - A list of Qualifier objects in the feature.

__init__(*key=''*, *location=''*)

Initialize the class.

__repr__()

Representation of the object for debugging or logging.

__str__()

Return feature as a GenBank format string.

class Bio.GenBank.Record.**Qualifier**(*key=''*, *value=''*)

Bases: object

Hold information about a qualifier in a GenBank feature.

Attributes:

- key - The key name of the qualifier (ie. /organism=)
- value - The value of the qualifier ("Dictyostelium discoideum").

__init__(*key=''*, *value=''*)

Initialize the class.

__repr__()

Representation of the object for debugging or logging.

__str__()

Return feature qualifier as a GenBank format string.

Bio.GenBank.Scanner module

Internal code for parsing GenBank and EMBL files (PRIVATE).

This code is NOT intended for direct use. It provides a basic scanner (for use with a event consumer such as Bio.GenBank._FeatureConsumer) to parse a GenBank or EMBL file (with their shared INSDC feature table).

It is used by Bio.GenBank to parse GenBank files It is also used by Bio.SeqIO to parse GenBank and EMBL files

Feature Table Documentation:

- http://www.insdc.org/files/feature_table.html
- http://www.ncbi.nlm.nih.gov/projects/collab/FT/index.html
- ftp://ftp.ncbi.nih.gov/genbank/docs/

class Bio.GenBank.Scanner.**InsdcScanner**(*debug=0*)

Bases: object

Basic functions for breaking up a GenBank/EMBL file into sub sections.

The International Nucleotide Sequence Database Collaboration (INSDC) between the DDBJ, EMBL, and GenBank. These organisations all use the same "Feature Table" layout in their plain text flat file formats.

However, the header and sequence sections of an EMBL file are very different in layout to those produced by GenBank/DDBJ.

RECORD_START = 'XXX'

HEADER_WIDTH = 3

FEATURE_START_MARKERS = ['XXX*FEATURES***XXX']**

FEATURE_END_MARKERS = ['XXX*END FEATURES***XXX']**

FEATURE_QUALIFIER_INDENT = 0

FEATURE_QUALIFIER_SPACER = ''

SEQUENCE_HEADERS = ['XXX']

__init__(*debug=0*)

Initialize the class.

set_handle(*handle*)

Set the handle attribute.

find_start()

Read in lines until find the ID/LOCUS line, which is returned.

Any preamble (such as the header used by the NCBI on `*.seq.gz` archives) will we ignored.

parse_header()

Return list of strings making up the header.

New line characters are removed.

Assumes you have just read in the ID/LOCUS line.

parse_features(*skip=False*)

Return list of tuples for the features (if present).

Each feature is returned as a tuple (key, location, qualifiers) where key and location are strings (e.g. "CDS" and "complement(join(490883..490885,1..879))") while qualifiers is a list of two string tuples (feature qualifier keys and values).

Assumes you have already read to the start of the features table.

parse_feature(*feature_key*, *lines*)

Parse a feature given as a list of strings into a tuple.

Expects a feature as a list of strings, returns a tuple (key, location, qualifiers)

For example given this GenBank feature:

```
CDS             complement(join(490883..490885,1..879))
                /locus_tag="NEQ001"
                /note="conserved hypothetical [Methanococcus jannaschii];
                COG1583:Uncharacterized ACR; IPR001472:Bipartite nuclear
                localization signal; IPR002743: Protein of unknown
                function DUF57"
                /codon_start=1
                /transl_table=11
                /product="hypothetical protein"
                /protein_id="NP_963295.1"
                /db_xref="GI:41614797"
                /db_xref="GeneID:2732620"
                /translation="MRLLLELKALNSIDKKQLSNYLIQGFIYNILKNTEYSWLHNWKK
                EKYFNFTLIPKKDIIENKRYYLIISSPDKRFIEVLHNKIKDLDIITIGLAQFQLRKTK
                KFDPKLRFPWVTITPIVLREGKIVILKGDKYYKVFVKRLEELKKYNLIKKKEPILEEP
                IEISLNQIKDGWKIIDVKDRYYDFRNKSFSAFSNWLRDLKEQSLRKYNNFCGKNFYFE
                EAIFEGFTFYKTVSIRIRINRGEAVYIGTLWKELNVYRKLDKEEREFYKFLYDCGLGS
                LNSMGFGFVNTKKNSAR"
```

Then should give input key="CDS" and the rest of the data as a list of strings lines=["complement(join(490883..490885,1..879))", ..., "LNSMGFGFVNTKKNSAR"] where the leading spaces and trailing newlines have been removed.

Returns tuple containing: (key as string, location string, qualifiers as list) as follows for this example:

key = "CDS", string location = "complement(join(490883..490885,1..879))", string qualifiers = list of string tuples:

[('locus_tag', '"NEQ001"'),
('note', '"conserved hypothetical [Methanococcus jannaschii];nCOG1583:…"'), ('codon_start', '1'), ('transl_table', '11'), ('product', '"hypothetical protein"'), ('protein_id', '"NP_963295.1"'), ('db_xref', '"GI:41614797"'), ('db_xref', '"GeneID:2732620"'), ('translation', '"MRLLLELKALNSIDKKQLSNYLIQGFIYNILKNTEYSWLHNWKKnEKYFNFT…"')]

In the above example, the "note" and "translation" were edited for compactness, and they would contain multiple new line characters (displayed above as n)

If a qualifier is quoted (in this case, everything except codon_start and transl_table) then the quotes are NOT removed.

Note that no whitespace is removed.

`parse_footer()`

Return a tuple containing a list of any misc strings, and the sequence.

`feed`(*handle*, *consumer*, *do_features=True*)

Feed a set of data into the consumer.

This method is intended for use with the "old" code in Bio.GenBank

Arguments:

- handle - A handle with the information to parse.
- consumer - The consumer that should be informed of events.
- do_features - Boolean, should the features be parsed? Skipping the features can be much faster.

Return values:

- true - Passed a record
- false - Did not find a record

`parse`(*handle*, *do_features=True*)

Return a SeqRecord (with SeqFeatures if do_features=True).

See also the method parse_records() for use on multi-record files.

`parse_records`(*handle*, *do_features=True*)

Parse records, return a SeqRecord object iterator.

Each record (from the ID/LOCUS line to the // line) becomes a SeqRecord

The SeqRecord objects include SeqFeatures if do_features=True

This method is intended for use in Bio.SeqIO

`parse_cds_features`(*handle*, *alphabet=None*, *tags2id=('protein_id', 'locus_tag', 'product')*)

Parse CDS features, return SeqRecord object iterator.

Each CDS feature becomes a SeqRecord.

Arguments:

- alphabet - Obsolete, should be left as None.
- tags2id - Tuple of three strings, the feature keys to use for the record id, name and description,

This method is intended for use in Bio.SeqIO

class Bio.GenBank.Scanner.**EmblScanner**(*debug=0*)

Bases: *InsdcScanner*

For extracting chunks of information in EMBL files.

RECORD_START = 'ID '

HEADER_WIDTH = 5

FEATURE_START_MARKERS = ['FH Key Location/Qualifiers', 'FH']

FEATURE_END_MARKERS = ['XX']

FEATURE_QUALIFIER_INDENT = 21

FEATURE_QUALIFIER_SPACER = 'FT '

SEQUENCE_HEADERS = ['SQ', 'CO']

EMBL_INDENT = 5

EMBL_SPACER = ' '

parse_footer()

Return a tuple containing a list of any misc strings, and the sequence.

__annotations__ = {}

class Bio.GenBank.Scanner.**GenBankScanner**(*debug=0*)

Bases: *InsdcScanner*

For extracting chunks of information in GenBank files.

RECORD_START = 'LOCUS '

HEADER_WIDTH = 12

FEATURE_START_MARKERS = ['FEATURES Location/Qualifiers', 'FEATURES']

FEATURE_END_MARKERS: list[str] = []

FEATURE_QUALIFIER_INDENT = 21

FEATURE_QUALIFIER_SPACER = ' '

SEQUENCE_HEADERS = ['CONTIG', 'ORIGIN', 'BASE COUNT', 'WGS', 'TSA', 'TLS']

GENBANK_INDENT = 12

GENBANK_SPACER = ' '

STRUCTURED_COMMENT_START = '-START##'

STRUCTURED_COMMENT_END = '-END##'

STRUCTURED_COMMENT_DELIM = ' :: '

parse_footer()

Return a tuple containing a list of any misc strings, and the sequence.

__annotations__ = {'FEATURE_END_MARKERS': list[str]}

Bio.GenBank.utils module

Useful utilities for helping in parsing GenBank files.

class Bio.GenBank.utils.**FeatureValueCleaner**(*to_process=keys_to_process*)

Bases: object

Provide specialized capabilities for cleaning up values in features.

This class is designed to provide a mechanism to clean up and process values in the key/value pairs of GenBank features. This is useful because in cases like:

```
/translation="MED
YDPWNLRFQSKYKSRDA"
```

you'll otherwise end up with white space in it.

This cleaning needs to be done on a case by case basis since it is impossible to interpret whether you should be concatenating everything (as in translations), or combining things with spaces (as might be the case with /notes).

```
>>> cleaner = FeatureValueCleaner(["translation"])
>>> cleaner
FeatureValueCleaner(['translation'])
>>> cleaner.clean_value("translation", "MED\nYDPWNLRFQSKYKSRDA")
'MEDYDPWNLRFQSKYKSRDA'
```

keys_to_process = ['translation']

__init__(*to_process=keys_to_process*)

Initialize with the keys we should deal with.

__repr__()

Return a string representation of the class.

clean_value(*key_name*, *value*)

Clean the specified value and return it.

If the value is not specified to be dealt with, the original value will be returned.

Module contents

Code to work with GenBank formatted files.

Rather than using Bio.GenBank, you are now encouraged to use Bio.SeqIO with the "genbank" or "embl" format names to parse GenBank or EMBL files into SeqRecord and SeqFeature objects (see the Biopython tutorial for details).

Using Bio.GenBank directly to parse GenBank files is only useful if you want to obtain GenBank-specific Record objects, which is a much closer representation to the raw file contents than the SeqRecord alternative from the FeatureParser (used in Bio.SeqIO).

To use the Bio.GenBank parser, there are two helper functions:

- read Parse a handle containing a single GenBank record as Bio.GenBank specific Record objects.
- parse Iterate over a handle containing multiple GenBank records as Bio.GenBank specific Record objects.

The following internal classes are not intended for direct use and may be deprecated in a future release.

Classes:

- Iterator Iterate through a file of GenBank entries
- FeatureParser Parse GenBank data in SeqRecord and SeqFeature objects.
- RecordParser Parse GenBank data into a Record object.

Exceptions:

- ParserFailureError Exception indicating a failure in the parser (ie. scanner or consumer)

class Bio.GenBank.**Iterator**(*handle*, *parser=None*)

Bases: object

Iterator interface to move over a file of GenBank entries one at a time (OBSOLETE).

This class is likely to be deprecated in a future release of Biopython. Please use Bio.SeqIO.parse(..., format="gb") or Bio.GenBank.parse(...) for SeqRecord and GenBank specific Record objects respectively instead.

__init__(*handle*, *parser=None*)

Initialize the iterator.

Arguments:

- handle - A handle with GenBank entries to iterate through.
- parser - An optional parser to pass the entries through before returning them. If None, then the raw entry will be returned.

__next__()

Return the next GenBank record from the handle.

Will return None if we ran out of records.

__iter__()

Iterate over the records.

exception Bio.GenBank.**ParserFailureError**

Bases: ValueError

Failure caused by some kind of problem in the parser.

class Bio.GenBank.**FeatureParser**(*debug_level=0*, *use_fuzziness=1*, *feature_cleaner=None*)

Bases: object

Parse GenBank files into Seq + Feature objects (OBSOLETE).

Direct use of this class is discouraged, and may be deprecated in a future release of Biopython.

Please use Bio.SeqIO.parse(...) or Bio.SeqIO.read(...) instead.

__init__(*debug_level=0*, *use_fuzziness=1*, *feature_cleaner=None*)

Initialize a GenBank parser and Feature consumer.

Arguments:

- debug_level - An optional argument that species the amount of debugging information the parser should spit out. By default we have no debugging info (the fastest way to do things), but if you want you can set this as high as two and see exactly where a parse fails.
- use_fuzziness - Specify whether or not to use fuzzy representations. The default is 1 (use fuzziness).

- feature_cleaner - A class which will be used to clean out the values of features. This class must implement the function clean_value. GenBank.utils has a "standard" cleaner class, which is used by default.

parse(*handle*)

Parse the specified handle.

class Bio.GenBank.**RecordParser**(*debug_level=0*)

Bases: object

Parse GenBank files into Record objects (OBSOLETE).

Direct use of this class is discouraged, and may be deprecated in a future release of Biopython.

Please use the Bio.GenBank.parse(...) or Bio.GenBank.read(...) functions instead.

__init__(*debug_level=0*)

Initialize the parser.

Arguments:

- debug_level - An optional argument that species the amount of debugging information the parser should spit out. By default we have no debugging info (the fastest way to do things), but if you want you can set this as high as two and see exactly where a parse fails.

parse(*handle*)

Parse the specified handle into a GenBank record.

Bio.GenBank.**parse**(*handle*)

Iterate over GenBank formatted entries as Record objects.

```
>>> from Bio import GenBank
>>> with open("GenBank/NC_000932.gb") as handle:
...     for record in GenBank.parse(handle):
...         print(record.accession)
['NC_000932']
```

To get SeqRecord objects use Bio.SeqIO.parse(..., format="gb") instead.

Bio.GenBank.**read**(*handle*)

Read a handle containing a single GenBank entry as a Record object.

```
>>> from Bio import GenBank
>>> with open("GenBank/NC_000932.gb") as handle:
...     record = GenBank.read(handle)
...     print(record.accession)
['NC_000932']
```

To get a SeqRecord object use Bio.SeqIO.read(..., format="gb") instead.

28.1.14 Bio.Geo package

Submodules

Bio.Geo.Record module

Hold GEO data in a straightforward format.

classes: o Record - All of the information in an GEO record.

See http://www.ncbi.nlm.nih.gov/geo/

class Bio.Geo.Record.**Record**

Bases: object

Hold GEO information in a format similar to the original record.

The Record class is meant to make data easy to get to when you are just interested in looking at GEO data.

Attributes: entity_type entity_id entity_attributes col_defs table_rows

__init__()

Initialize the class.

__str__()

Return the GEO record as a string.

Bio.Geo.Record.**out_block**(*text*, *prefix=''*)

Format text in blocks of 80 chars with an additional optional prefix.

Module contents

Parser for files from NCBI's Gene Expression Omnibus (GEO).

http://www.ncbi.nlm.nih.gov/geo/

Bio.Geo.**parse**(*handle*)

Read Gene Expression Omnibus records from file handle.

Returns a generator object which yields Bio.Geo.Record() objects.

28.1.15 Bio.Graphics package

Subpackages

Bio.Graphics.GenomeDiagram package

Module contents

GenomeDiagram module integrated into Biopython.

class Bio.Graphics.GenomeDiagram.**Diagram**(*name=None*, *format='circular'*, *pagesize='A3'*, *orientation='landscape'*, *x=0.05*, *y=0.05*, *xl=None*, *xr=None*, *yt=None*, *yb=None*, *start=None*, *end=None*, *tracklines=False*, *fragments=10*, *fragment_size=None*, *track_size=0.75*, *circular=True*, *circle_core=0.0*)

Bases: `object`

Diagram container.

Arguments:

- name - a string, identifier for the diagram.
- tracks - a list of Track objects comprising the diagram.
- format - a string, format of the diagram 'circular' or 'linear', depending on the sort of diagram required.
- pagesize - a string, the pagesize of output describing the ISO size of the image, or a tuple of pixels.
- orientation - a string describing the required orientation of the final drawing ('landscape' or 'portrait').
- x - a float (0->1), the proportion of the page to take up with even X margins t the page.
- y - a float (0->1), the proportion of the page to take up with even Y margins to the page.
- xl - a float (0->1), the proportion of the page to take up with the left X margin to the page (overrides x).
- xr - a float (0->1), the proportion of the page to take up with the right X margin to the page (overrides x).
- yt - a float (0->1), the proportion of the page to take up with the top Y margin to the page (overrides y).
- yb - a float (0->1), the proportion of the page to take up with the bottom Y margin to the page (overrides y).
- circle_core - a float, the proportion of the available radius to leave empty at the center of a circular diagram (0 to 1).
- start - an integer, the base/aa position to start the diagram at.
- end - an integer, the base/aa position to end the diagram at.
- tracklines - a boolean, True if track guidelines are to be drawn.
- fragments - and integer, for a linear diagram, the number of equal divisions into which the sequence is divided.
- fragment_size - a float (0->1), the proportion of the space available to each fragment that should be used in drawing.
- track_size - a float (0->1), the proportion of the space available to each track that should be used in drawing with sigils.
- circular - a boolean, True if the genome/sequence to be drawn is, in reality, circular.

__init__(*name=None*, *format='circular'*, *pagesize='A3'*, *orientation='landscape'*, *x=0.05*, *y=0.05*, *xl=None*, *xr=None*, *yt=None*, *yb=None*, *start=None*, *end=None*, *tracklines=False*, *fragments=10*, *fragment_size=None*, *track_size=0.75*, *circular=True*, *circle_core=0.0*)

Initialize.

gdd = Diagram(name=None)

set_all_tracks(*attr*, *value*)

Set the passed attribute of all tracks in the set to the passed value.

Arguments:

- attr - An attribute of the Track class.

- value - The value to set that attribute.

set_all_tracks(self, attr, value)

draw(*format=None*, *pagesize=None*, *orientation=None*, *x=None*, *y=None*, *xl=None*, *xr=None*, *yt=None*, *yb=None*, *start=None*, *end=None*, *tracklines=None*, *fragments=None*, *fragment_size=None*, *track_size=None*, *circular=None*, *circle_core=None*, *cross_track_links=None*)

Draw the diagram, with passed parameters overriding existing attributes.

gdd.draw(format='circular')

write(*filename='test1.ps'*, *output='PS'*, *dpi=72*)

Write the drawn diagram to a specified file, in a specified format.

Arguments:

- filename - a string indicating the name of the output file, or a handle to write to.
- output - a string indicating output format, one of PS, PDF, SVG, or provided the ReportLab renderPM module is installed, one of the bitmap formats JPG, BMP, GIF, PNG, TIFF or TIFF. The format can be given in upper or lower case.
- dpi - an integer. Resolution (dots per inch) for bitmap formats.

Returns:
No return value.

write(self, filename='test1.ps', output='PS', dpi=72)

write_to_string(*output='PS'*, *dpi=72*)

Return a byte string containing the diagram in the requested format.

Arguments:

- output - a string indicating output format, one of PS, PDF, SVG, JPG, BMP, GIF, PNG, TIFF or TIFF (as specified for the write method).
- dpi - Resolution (dots per inch) for bitmap formats.

Returns:
Return the completed drawing as a bytes string in a prescribed format.

add_track(*track*, *track_level*)

Add a Track object to the diagram.

It also accepts instructions to place it at a particular level on the diagram.

Arguments:

- track - Track object to draw.
- track_level - an integer. The level at which the track will be drawn (above an arbitrary baseline).

add_track(self, track, track_level)

new_track(*track_level*, ***args*)

Add a new Track to the diagram at a given level.

The track is returned for further user manipulation.

Arguments:

- track_level - an integer. The level at which the track will be drawn (above an arbitrary baseline).

new_track(self, track_level)

del_track(*track_level*)

Remove the track to be drawn at a particular level on the diagram.

Arguments:

- track_level - an integer. The level of the track on the diagram to delete.

del_track(self, track_level)

get_tracks()

Return a list of the tracks contained in the diagram.

move_track(*from_level*, *to_level*)

Move a track from one level on the diagram to another.

Arguments:

- from_level - an integer. The level at which the track to be moved is found.
- to_level - an integer. The level to move the track to.

renumber_tracks(*low=1*, *step=1*)

Renumber all tracks consecutively.

Optionally from a passed lowest number.

Arguments:

- low - an integer. The track number to start from.
- step - an integer. The track interval for separation of tracks.

get_levels()

Return a sorted list of levels occupied by tracks in the diagram.

get_drawn_levels()

Return a sorted list of levels occupied by tracks.

These tracks are not explicitly hidden.

range()

Return lowest and highest base numbers from track features.

Returned type is a tuple.

__getitem__(*key*)

Return the track contained at the level of the passed key.

__str__()

Return a formatted string describing the diagram.

class Bio.Graphics.GenomeDiagram.**Track**(*name=None*, *height=1*, *hide=0*, *greytrack=0*, *greytrack_labels=5*, *greytrack_fontsize=8*, *greytrack_font='Helvetica'*, *greytrack_font_rotation=0*, *greytrack_font_color=_grey*, *scale=1*, *scale_format=None*, *scale_color=colors.black*, *scale_font='Helvetica'*, *scale_fontsize=6*, *scale_fontangle=45*, *scale_largeticks=0.5*, *scale_ticks=1*, *scale_smallticks=0.3*, *scale_largetick_interval=1e6*, *scale_smalltick_interval=1e4*, *scale_largetick_labels=1*, *scale_smalltick_labels=0*, *axis_labels=1*, *start=None*, *end=None*, *greytrack_font_colour=None*, *scale_colour=None*)

Bases: `object`

Track.

Attributes:

- height Int describing the relative height to other trackscale_fontsizes in the diagram
- name String describing the track
- hide Boolean, 0 if the track is not to be drawn
- start, end Integers (or None) specifying start/end to draw just a partial track.
- greytrack Boolean, 1 if a grey background to the track is to be drawn
- greytrack_labels Int describing how many track-identifying labels should be placed on the track at regular intervals
- greytrack_font String describing the font to use for the greytrack labels
- greytrack_fontsize Int describing the font size to display the labels on the grey track
- greytrack_font_rotation Int describing the angle through which to rotate the grey track labels (Linear only)
- greytrack_font_color colors.Color describing the color to draw the grey track labels
- scale Boolean, 1 if a scale is to be drawn on the track
- scale_format String, defaults to None, when scale values are written as numerals. Setting this to 'SInt' invokes SI unit-like multiples, such as Mbp, Kbp and so on.
- scale_color colors.Color to draw the elements of the scale
- scale_font String describing the font to use for the scale labels
- scale_fontsize Int describing the size of the scale label font
- scale_fontangle Int describing the angle at which to draw the scale labels (linear only)
- scale_ticks Boolean, 1 if ticks should be drawn at all on the scale
- scale_largeticks Float (0->1) describing the height of large scale ticks relative to the track height.
- scale_smallticks Float (0->1) describing the height of large scale ticks relative to the track height.
- scale_largetick_interval Int, describing the number of bases that should separate large ticks
- scale_smalltick_interval Int, describing the number of bases that should separate small ticks
- scale_largetick_labels Boolean describing whether position labels should be written over large ticks
- scale_smalltick_labels Boolean describing whether position labels should be written over small ticks
- axis_labels Boolean describing whether the value labels should be placed on the Y axes

__init__(*name=None*, *height=1*, *hide=0*, *greytrack=0*, *greytrack_labels=5*, *greytrack_fontsize=8*, *greytrack_font='Helvetica'*, *greytrack_font_rotation=0*, *greytrack_font_color=_grey*, *scale=1*, *scale_format=None*, *scale_color=colors.black*, *scale_font='Helvetica'*, *scale_fontsize=6*, *scale_fontangle=45*, *scale_largeticks=0.5*, *scale_ticks=1*, *scale_smallticks=0.3*, *scale_largetick_interval=1e6*, *scale_smalltick_interval=1e4*, *scale_largetick_labels=1*, *scale_smalltick_labels=0*, *axis_labels=1*, *start=None*, *end=None*, *greytrack_font_colour=None*, *scale_colour=None*)

Initialize.

Arguments:

- height Int describing the relative height to other tracks in the diagram
- name String describing the track
- hide Boolean, 0 if the track is not to be drawn
- greytrack Boolean, 1 if a grey background to the track is to be drawn
- greytrack_labels Int describing how many track-identifying labels should be placed on the track at regular intervals
- greytrack_font String describing the font to use for the greytrack labels
- greytrack_fontsize Int describing the font size to display the labels on the grey track
- greytrack_font_rotation Int describing the angle through which to rotate the grey track labels (Linear only)
- greytrack_font_color colors.Color describing the color to draw the grey track labels (overridden by backwards compatible argument with UK spelling, colour).
- scale Boolean, 1 if a scale is to be drawn on the track
- scale_color colors.Color to draw the elements of the scale (overridden by backwards compatible argument with UK spelling, colour).
- scale_font String describing the font to use for the scale labels
- scale_fontsize Int describing the size of the scale label font
- scale_fontangle Int describing the angle at which to draw the scale labels (linear only)
- scale_ticks Boolean, 1 if ticks should be drawn at all on the scale
- scale_largeticks Float (0->1) describing the height of large scale ticks relative to the track height.
- scale_smallticks Float (0->1) describing the height of large scale ticks relative to the track height.
- scale_largetick_interval Int, describing the number of bases that should separate large ticks
- scale_smalltick_interval Int, describing the number of bases that should separate small ticks
- scale_largetick_labels Boolean describing whether position labels should be written over large ticks
- scale_smalltick_labels Boolean describing whether position labels should be written over small ticks
- name String to help identify the track
- height Relative height to draw the track
- axis_labels Boolean describing whether the value labels should be placed on the Y axes

add_set(*set*)

Add a preexisting FeatureSet or GraphSet object to the track.

new_set(*type='feature'*, ***args*)

Create a new FeatureSet or GraphSet object.

Create a new FeatureSet or GraphSet object, add it to the track, and return for user manipulation

del_set(*set_id*)

Remove the set with the passed id from the track.

get_sets()

Return the sets contained in this track.

get_ids()

Return the ids of all sets contained in this track.

range()

Return the lowest and highest base (or mark) numbers as a tuple.

to_string(*verbose=0*)

Return a formatted string with information about the track.

Arguments:

- verbose - Boolean indicating whether a short or complete account of the track is required

__getitem__(*key*)

Return the set with the passed id.

__str__()

Return a formatted string with information about the Track.

class Bio.Graphics.GenomeDiagram.**FeatureSet**(*set_id=None*, *name=None*, *parent=None*)

Bases: object

FeatureSet object.

__init__(*set_id=None*, *name=None*, *parent=None*)

Create the object.

Arguments:

- set_id: Unique id for the set
- name: String identifying the feature set

add_feature(*feature*, ***kwargs*)

Add a new feature.

Arguments:

- feature: Bio.SeqFeature object
- kwargs: Keyword arguments for Feature. Named attributes of the Feature

Add a Bio.SeqFeature object to the diagram (will be stored internally in a Feature wrapper).

del_feature(*feature_id*)

Delete a feature.

Arguments:

- feature_id: Unique id of the feature to delete

Remove a feature from the set, indicated by its id.

set_all_features(*attr*, *value*)

Set an attribute of all the features.

Arguments:

- attr: An attribute of the Feature class
- value: The value to set that attribute to

Set the passed attribute of all features in the set to the passed value.

get_features(*attribute=None*, *value=None*, *comparator=None*)

Retrieve features.

Arguments:

- attribute: String, attribute of a Feature object
- value: The value desired of the attribute
- comparator: String, how to compare the Feature attribute to the passed value

If no attribute or value is given, return a list of all features in the feature set. If both an attribute and value are given, then depending on the comparator, then a list of all features in the FeatureSet matching (or not) the passed value will be returned. Allowed comparators are: 'startswith', 'not', 'like'.

The user is expected to make a responsible decision about which feature attributes to use with which passed values and comparator settings.

get_ids()

Return a list of all ids for the feature set.

range()

Return the lowest and highest base (or mark) numbers as a tuple.

to_string(*verbose=0*)

Return a formatted string with information about the set.

Arguments:

- verbose: Boolean indicating whether a short (default) or complete account of the set is required

__len__()

Return the number of features in the set.

__getitem__(*key*)

Return a feature, keyed by id.

__str__()

Return a formatted string with information about the feature set.

class Bio.Graphics.GenomeDiagram.**Feature**(*parent=None*, *feature_id=None*, *feature=None*, *color=colors.lightgreen*, *label=0*, *border=None*, *colour=None*)

Bases: object

Class to wrap Bio.SeqFeature objects for GenomeDiagram.

Attributes:

- parent FeatureSet, container for the object
- id Unique id
- color color.Color, color to draw the feature
- hide Boolean for whether the feature will be drawn or not
- sigil String denoting the type of sigil to use for the feature. Currently either "BOX" or "ARROW" are supported.
- arrowhead_length Float denoting length of the arrow head to be drawn, relative to the bounding box height. The arrow shaft takes up the remainder of the bounding box's length.

- arrowshaft_height Float denoting length of the representative arrow shaft to be drawn, relative to the bounding box height. The arrow head takes the full height of the bound box.
- name_qualifiers List of Strings, describes the qualifiers that may contain feature names in the wrapped Bio.SeqFeature object
- label Boolean, 1 if the label should be shown
- label_font String describing the font to use for the feature label
- label_size Int describing the feature label font size
- label_color color.Color describing the feature label color
- label_angle Float describing the angle through which to rotate the feature label in degrees (default = 45, linear only)
- label_position String, 'start', 'end' or 'middle' denoting where to place the feature label. Leave as None for the default which is 'start' for linear diagrams, and at the bottom of the feature as drawn on circular diagrams.
- label_strand Integer -1 or +1 to explicitly place the label on the forward or reverse strand. Default (None) follows th feature's strand. Use -1 to put labels under (linear) or inside (circular) the track, +1 to put them above (linear) or outside (circular) the track.
- locations List of tuples of (start, end) ints describing where the feature and any subfeatures start and end
- type String denoting the feature type
- name String denoting the feature name
- strand Int describing the strand on which the feature is found

__init__(*parent=None*, *feature_id=None*, *feature=None*, *color=colors.lightgreen*, *label=0*, *border=None*, *colour=None*)

Initialize.

Arguments:

- parent FeatureSet containing the feature
- feature_id Unique id for the feature
- feature Bio.SeqFeature object to be wrapped
- color color.Color Color to draw the feature (overridden by backwards compatible argument with UK spelling, colour). Either argument is overridden if 'color' is found in feature qualifiers
- border color.Color Color to draw the feature border, use None for the same as the fill color, False for no border.
- label Boolean, 1 if the label should be shown

set_feature(*feature*)

Define the Bio.SeqFeature object to be wrapped.

get_feature()

Return the unwrapped Bio.SeqFeature object.

set_colour(*colour*)

Backwards compatible variant of set_color(self, color) using UK spelling.

set_color(*color*)

Set the color in which the feature will be drawn.

Arguments:

- color The color to draw the feature - either a colors.Color object, an RGB tuple of floats, or an integer corresponding a colors in colors.txt

__getattr__(*name*)

Get attribute by name.

If the Feature class doesn't have the attribute called for, check in self._feature for it.

class Bio.Graphics.GenomeDiagram.**GraphSet**(*name=None*)

Bases: object

Graph Set.

Attributes:

- id Unique identifier for the set
- name String describing the set

__init__(*name=None*)

Initialize.

Arguments:

- name String identifying the graph set sensibly

new_graph(*data*, *name=None*, *style='bar'*, *color=colors.lightgreen*, *altcolor=colors.darkseagreen*, *linewidth=1*, *center=None*, *colour=None*, *altcolour=None*, *centre=None*)

Add a GraphData object to the diagram.

Arguments:

- data List of (position, value) int tuples
- name String, description of the graph
- style String ('bar', 'heat', 'line') describing how the graph will be drawn
- color colors.Color describing the color to draw all or 'high' (some styles) data (overridden by backwards compatible argument with UK spelling, colour).
- altcolor colors.Color describing the color to draw 'low' (some styles) data (overridden by backwards compatible argument with UK spelling, colour).
- linewidth Float describing linewidth for graph
- center Float setting the value at which the x-axis crosses the y-axis (overridden by backwards compatible argument with UK spelling, centre)

Add a GraphData object to the diagram (will be stored internally).

del_graph(*graph_id*)

Remove a graph from the set, indicated by its id.

get_graphs()

Return list of all graphs in the graph set, sorted by id.

Sorting is to ensure reliable stacking.

get_ids()

Return a list of all ids for the graph set.

range()

Return the lowest and highest base (or mark) numbers as a tuple.

data_quartiles()

Return (minimum, lowerQ, medianQ, upperQ, maximum) values as a tuple.

to_string(*verbose=0*)

Return a formatted string with information about the set.

Arguments:

- verbose - Flag indicating whether a short or complete account of the set is required

__len__()

Return the number of graphs in the set.

__getitem__(*key*)

Return a graph, keyed by id.

__str__()

Return a formatted string with information about the feature set.

class Bio.Graphics.GenomeDiagram.**GraphData**(*id=None*, *data=None*, *name=None*, *style='bar'*, *color=colors.lightgreen*, *altcolor=colors.darkseagreen*, *center=None*, *colour=None*, *altcolour=None*)

Bases: object

Graph Data.

Attributes:

- id Unique identifier for the data
- data Dictionary of describing the data, keyed by position
- name String describing the data
- style String ('bar', 'heat', 'line') describing how to draw the data
- poscolor colors.Color for drawing high (some styles) or all values
- negcolor colors.Color for drawing low values (some styles)
- linewidth Int, thickness to draw the line in 'line' styles

__init__(*id=None*, *data=None*, *name=None*, *style='bar'*, *color=colors.lightgreen*, *altcolor=colors.darkseagreen*, *center=None*, *colour=None*, *altcolour=None*)

Initialize.

Arguments:

- id Unique ID for the graph
- data List of (position, value) tuples
- name String describing the graph
- style String describing the presentation style ('bar', 'line', 'heat')
- color colors.Color describing the color to draw all or the 'high' (some styles) values (overridden by backwards compatible argument with UK spelling, colour).

- altcolor colors.Color describing the color to draw the 'low' values (some styles only) (overridden by backwards compatible argument with UK spelling, colour).
- center Value at which x-axis crosses y-axis.

set_data(*data*)

Add data as a list of (position, value) tuples.

get_data()

Return data as a list of sorted (position, value) tuples.

add_point(*point*)

Add a single point to the set of data as a (position, value) tuple.

quartiles()

Return (minimum, lowerQ, medianQ, upperQ, maximum) values as tuple.

range()

Return range of data as (start, end) tuple.

Returns the range of the data, i.e. its start and end points on the genome as a (start, end) tuple.

mean()

Return the mean value for the data points (float).

stdev()

Return the sample standard deviation for the data (float).

__len__()

Return the number of points in the data set.

__getitem__(*index*)

Return data value(s) at the given position.

Given an integer representing position on the sequence returns a float - the data value at the passed position.

If a slice, returns graph data from the region as a list or (position, value) tuples. Slices with step are not supported.

__str__()

Return a string describing the graph data.

class Bio.Graphics.GenomeDiagram.**CrossLink**(*featureA*, *featureB*, *color=colors.lightgreen*, *border=None*, *flip=False*)

Bases: object

Hold information for drawing a cross link between features.

__init__(*featureA*, *featureB*, *color=colors.lightgreen*, *border=None*, *flip=False*)

Create a new cross link.

Arguments featureA and featureB should GenomeDiagram feature objects, or 3-tuples (track object, start, end), and currently must be on different tracks.

The color and border arguments should be ReportLab colour objects, or for border use a boolean False for no border, otherwise it defaults to the same as the main colour.

The flip argument draws an inverted cross link, useful for showing a mapping where one sequence has been reversed. It is conventional to also use a different colour (e.g. red for simple links, blue for any flipped links).

property startA

Start position of Feature A.

property endA

End position of Feature A.

property startB

Start position of Feature B.

property endB

End position of Feature B.

class Bio.Graphics.GenomeDiagram.**ColorTranslator**(*filename=None*)

Bases: object

Class providing methods for translating representations of color into.

Examples

```
>>> from Bio.Graphics import GenomeDiagram
>>> gdct=GenomeDiagram._Colors.ColorTranslator()
>>> print(gdct.float1_color((0.5, 0.5, 0.5)))
Color(.5,.5,.5,1)
>>> print(gdct.int255_color((1, 75, 240)))
Color(.003922,.294118,.941176,1)
>>> print(gdct.artemis_color(7))
Color(1,1,0,1)
>>> print(gdct.scheme_color(2))
Color(1,0,0,1)
>>> gdct.get_artemis_colorscheme()
{0: (Color(1,1,1,1), 'pathogenicity, adaptation, chaperones'), 1: (Color(.39,.39,.
↪39,1), 'energy metabolism'), 2: (Color(1,0,0,1), 'information transfer'), 3:␣
↪(Color(0,1,0,1), 'surface'), 4: (Color(0,0,1,1), 'stable RNA'), 5: (Color(0,1,1,
↪1), 'degradation of large molecules'), 6: (Color(1,0,1,1), 'degradation of small␣
↪molecules'), 7: (Color(1,1,0,1), 'central/intermediary/miscellaneous metabolism'),
↪ 8: (Color(.6,.98,.6,1), 'unknown'), 9: (Color(.53,.81,.98,1), 'regulators'), 10:␣
↪(Color(1,.65,0,1), 'conserved hypotheticals'), 11: (Color(.78,.59,.39,1),
↪'pseudogenes and partial genes'), 12: (Color(1,.78,.78,1), 'phage/IS elements'),␣
↪13: (Color(.7,.7,.7,1), 'some miscellaneous information'), 14: (Color(0,0,0,1), '
↪'), 15: (Color(1,.25,.25,1), 'secondary metabolism'), 16: (Color(1,.5,.5,1), ''),␣
↪17: (Color(1,.75,.75,1), '')}
```

```
>>> print(gdct.translate((0.5, 0.5, 0.5)))
Color(.5,.5,.5,1)
>>> print(gdct.translate((1, 75, 240)))
Color(.003922,.294118,.941176,1)
>>> print(gdct.translate(7))
Color(1,1,0,1)
>>> print(gdct.translate(2))
Color(1,0,0,1)
```

__init__(*filename=None*)

Initialize.

Argument filename is the location of a file containing colorscheme information.

translate(*color=None*, *colour=None*)

Translate a color into a ReportLab Color object.

Arguments:

- color - Color defined as an int, a tuple of three ints 0->255 or a tuple of three floats 0 -> 1, or a string giving one of the named colors defined by ReportLab, or a ReportLab color object (returned as is).
- colour - Backwards compatible alias using UK spelling (which will over-ride any color argument).

Returns a colors.Color object, determined semi-intelligently depending on the input values

read_colorscheme(*filename*)

Load colour scheme from file.

Reads information from a file containing color information and stores it internally.

Argument filename is the location of a file defining colors in tab-separated format plaintext as:

```
INT \t RED \t GREEN \t BLUE \t Comment
```

Where RED, GREEN and BLUE are intensities in the range 0 -> 255, e.g.:

```
2 \t 255 \t 0 \t 0 \t Red: Information transfer
```

get_artemis_colorscheme()

Return the Artemis color scheme as a dictionary.

artemis_color(*value*)

Artemis color (integer) to ReportLab Color object.

Arguments:

- value: An int representing a functional class in the Artemis color scheme (see www.sanger.ac.uk for a description), or a string from a GenBank feature annotation for the color which may be dot delimited (in which case the first value is used).

Takes an int representing a functional class in the Artemis color scheme, and returns the appropriate colors.Color object

get_colorscheme()

Return the user-defined color scheme as a dictionary.

scheme_color(*value*)

Map a user-defined color integer to a ReportLab Color object.

- value: An int representing a single color in the user-defined color scheme

Takes an int representing a user-defined color and returns the appropriate colors.Color object.

int255_color(*values*)

Map integer (red, green, blue) tuple to a ReportLab Color object.

- values: A tuple of (red, green, blue) intensities as integers in the range 0->255

Takes a tuple of (red, green, blue) intensity values in the range 0 -> 255 and returns an appropriate colors.Color object.

float1_color(*values*)

Map float (red, green, blue) tuple to a ReportLab Color object.

- values: A tuple of (red, green, blue) intensities as floats in the range 0 -> 1

Takes a tuple of (red, green, blue) intensity values in the range 0 -> 1 and returns an appropriate colors.Color object.

Submodules

Bio.Graphics.BasicChromosome module

Draw representations of organism chromosomes with added information.

These classes are meant to model the drawing of pictures of chromosomes. This can be useful for lots of things, including displaying markers on a chromosome (ie. for genetic mapping) and showing syteny between two chromosomes.

The structure of these classes is intended to be a Composite, so that it will be easy to plug in and switch different parts without breaking the general drawing capabilities of the system. The relationship between classes is that everything derives from _ChromosomeComponent, which specifies the overall interface. The parts then are related so that an Organism contains Chromosomes, and these Chromosomes contain ChromosomeSegments. This representation differs from the canonical composite structure in that we don't really have 'leaf' nodes here – all components can potentially hold sub-components.

Most of the time the ChromosomeSegment class is what you'll want to customize for specific drawing tasks.

For providing drawing capabilities, these classes use reportlab:

http://www.reportlab.com

This provides nice output in PDF, SVG and postscript. If you have reportlab's renderPM module installed you can also use PNG etc.

class Bio.Graphics.BasicChromosome.**Organism**(*output_format='pdf'*)

Bases: _ChromosomeComponent

Top level class for drawing chromosomes.

This class holds information about an organism and all of its chromosomes, and provides the top level object which could be used for drawing a chromosome representation of an organism.

Chromosomes should be added and removed from the Organism via the add and remove functions.

__init__(*output_format='pdf'*)

Initialize the class.

draw(*output_file*, *title*)

Draw out the information for the Organism.

Arguments:

- output_file – The name of a file specifying where the document should be saved, or a handle to be written to. The output format is set when creating the Organism object. Alternatively, output_file=None will return the drawing using the low-level ReportLab objects (for further processing, such as adding additional graphics, before writing).
- title – The output title of the produced document.

class Bio.Graphics.BasicChromosome.**Chromosome**(*chromosome_name*)

Bases: _ChromosomeComponent

Class for drawing a chromosome of an organism.

This organizes the drawing of a single organisms chromosome. This class can be instantiated directly, but the draw method makes the most sense to be called in the context of an organism.

__init__(*chromosome_name*)

Initialize a Chromosome for drawing.

Arguments:

- chromosome_name - The label for the chromosome.

Attributes:

- start_x_position, end_x_position - The x positions on the page where the chromosome should be drawn. This allows multiple chromosomes to be drawn on a single page.
- start_y_position, end_y_position - The y positions on the page where the chromosome should be contained.

Configuration Attributes:

- title_size - The size of the chromosome title.
- scale_num - A number of scale the drawing by. This is useful if you want to draw multiple chromosomes of different sizes at the same scale. If this is not set, then the chromosome drawing will be scaled by the number of segments in the chromosome (so each chromosome will be the exact same final size).

subcomponent_size()

Return the scaled size of all subcomponents of this component.

draw(*cur_drawing*)

Draw a chromosome on the specified template.

Ideally, the x_position and y_*_position attributes should be set prior to drawing – otherwise we're going to have some problems.

__annotations__ = {}

class Bio.Graphics.BasicChromosome.**ChromosomeSegment**

Bases: _ChromosomeComponent

Draw a segment of a chromosome.

This class provides the important configurable functionality of drawing a Chromosome. Each segment has some customization available here, or can be subclassed to define additional functionality. Most of the interesting drawing stuff is likely to happen at the ChromosomeSegment level.

__init__()

Initialize a ChromosomeSegment.

Attributes:

- start_x_position, end_x_position - Defines the x range we have to draw things in.
- start_y_position, end_y_position - Defines the y range we have to draw things in.

Configuration Attributes:

- scale - A scaling value for the component. By default this is set at 1 (ie – has the same scale as everything else). Higher values give more size to the component, smaller values give less.
- fill_color - A color to fill in the segment with. Colors are available in reportlab.lib.colors
- label - A label to place on the chromosome segment. This should be a text string specifying what is to be included in the label.
- label_size - The size of the label.
- chr_percent - The percentage of area that the chromosome segment takes up.

draw(*cur_drawing*)

Draw a chromosome segment.

Before drawing, the range we are drawing in needs to be set.

__annotations__ = {}

class Bio.Graphics.BasicChromosome.**AnnotatedChromosomeSegment**(*bp_length*, *features*, *default_feature_color=colors.blue*, *name_qualifiers=('gene', 'label', 'name', 'locus_tag', 'product')*)

Bases: *ChromosomeSegment*

Annotated chromosome segment.

This is like the ChromosomeSegment, but accepts a list of features.

__init__(*bp_length*, *features*, *default_feature_color=colors.blue*, *name_qualifiers=('gene', 'label', 'name', 'locus_tag', 'product')*)

Initialize.

The features can either be SeqFeature objects, or tuples of values: start (int), end (int), strand (+1, -1, O or None), label (string), ReportLab color (string or object), and optional ReportLab fill color.

Note we require 0 <= start <= end <= bp_length, and within the vertical space allocated to this segment lines will be places according to the start/end coordinates (starting from the top).

Positive stand features are drawn on the right, negative on the left, otherwise all the way across.

We recommend using consistent units for all the segment's scale values (e.g. their length in base pairs).

When providing features as SeqFeature objects, the default color is used, unless the feature's qualifiers include an Artemis colour string (functionality also in GenomeDiagram). The caption also follows the GenomeDiagram approach and takes the first qualifier from the list or tuple specified in name_qualifiers.

Note additional attribute label_sep_percent controls the percentage of area that the chromosome segment takes up, by default half of the chr_percent attribute (half of 25%, thus 12.5%)

__annotations__ = {}

class Bio.Graphics.BasicChromosome.**TelomereSegment**(*inverted=0*)

Bases: *ChromosomeSegment*

A segment that is located at the end of a linear chromosome.

This is just like a regular segment, but it draws the end of a chromosome which is represented by a half circle. This just overrides the _draw_segment class of ChromosomeSegment to provide that specialized drawing.

__init__(*inverted=0*)

Initialize a segment at the end of a chromosome.

See ChromosomeSegment for all of the attributes that can be customized in a TelomereSegments.

Arguments:

- inverted – Whether or not the telomere should be inverted (ie. drawn on the bottom of a chromosome)

`__annotations__ = {}`

`class Bio.Graphics.BasicChromosome.SpacerSegment`

Bases: *ChromosomeSegment*

A segment that is located at the end of a linear chromosome.

Doesn't draw anything, just empty space which can be helpful for layout purposes (e.g. making room for feature labels).

`draw(cur_diagram)`

Draw nothing to the current diagram (dummy method).

The segment spacer has no actual image in the diagram, so this method therefore does nothing, but is defined to match the expected API of the other segment objects.

`__annotations__ = {}`

Bio.Graphics.ColorSpiral module

Generate RGB colours suitable for distinguishing categorical data.

This module provides a class that implements a spiral 'path' through HSV colour space, permitting the selection of a number of points along that path, and returning the output in RGB colour space, suitable for use with ReportLab and other graphics packages.

This approach to colour choice was inspired by Bang Wong's Points of View article: Color Coding, in Nature Methods _7_ 573 (https://doi.org/10.1038/nmeth0810-573).

The module also provides helper functions that return a list for colours, or a dictionary of colours (if passed an iterable containing the names of categories to be coloured).

`class Bio.Graphics.ColorSpiral.ColorSpiral(a=1, b=0.33, v_init=0.85, v_final=0.5, jitter=0.05)`

Bases: `object`

Implement a spiral path through HSV colour space.

This class provides functions for sampling points along a logarithmic spiral path through HSV colour space.

The spiral is described by r = a * exp(b * t) where r is the distance from the axis of the HSV cylinder to the current point in the spiral, and t is the angle through which the spiral has turned to reach the current point. a and b are (positive, real) parameters that control the shape of the spiral.

- a: the starting direction of the spiral
- b: the number of revolutions about the axis made by the spiral

We permit the spiral to move along the cylinder ('in V-space') between v_init and v_final, to give a gradation in V (essentially, brightness), along the path, where v_init, v_final are in [0,1].

A brightness 'jitter' may also be provided as an absolute value in V-space, to aid in distinguishing consecutive colour points on the path.

`__init__(a=1, b=0.33, v_init=0.85, v_final=0.5, jitter=0.05)`

Initialize a logarithmic spiral path through HSV colour space.

Arguments:

- a - Parameter a for the spiral, controls the initial spiral direction. a > 0
- b - parameter b for the spiral, controls the rate at which the spiral revolves around the axis. b > 0
- v_init - initial value of V (brightness) for the spiral. v_init in [0,1]
- v_final - final value of V (brightness) for the spiral v_final in [0,1]
- jitter - the degree of V (brightness) jitter to add to each selected colour. The amount of jitter will be selected from a uniform random distribution [-jitter, jitter], and V will be maintained in [0,1].

get_colors(*k*, *offset=0.1*)

Generate k different RBG colours evenly-space on the spiral.

A generator returning the RGB colour space values for k evenly-spaced points along the defined spiral in HSV space.

Arguments:

- k - the number of points to return
- offset - how far along the spiral path to start.

property a

Parameter controlling initial spiral direction (a > 0)

property b

Parameter controlling rate spiral revolves around axis (b > 0)

property v_init

Initial value of V (brightness) for the spiral (range 0 to 1)

property v_final

Final value of V (brightness) for the spiral (range 0 to 1)

property jitter

Degree of V (brightness) jitter to add to each color (range 0 to 1)

Bio.Graphics.ColorSpiral.**get_colors**(*k*, ***kwargs*)

Return k colours selected by the ColorSpiral object, as a generator.

Arguments:

- k - the number of colours to return
- kwargs - pass-through arguments to the ColorSpiral object

Bio.Graphics.ColorSpiral.**get_color_dict**(*l*, ***kwargs*)

Return a dictionary of colours using the provided values as keys.

Returns a dictionary, keyed by the members of iterable l, with a colour assigned to each member.

Arguments:

- l - an iterable representing classes to be coloured
- kwargs - pass-through arguments to the ColorSpiral object

Bio.Graphics.Comparative module

Plots to compare information between different sources.

This file contains high level plots which are designed to be used to compare different types of information. The most basic example is comparing two variables in a traditional scatter plot.

class Bio.Graphics.Comparative.**ComparativeScatterPlot**(*output_format='pdf'*)

Bases: object

Display a scatter-type plot comparing two different kinds of info.

Attributes;

- display_info - a 2D list of the information we'll be outputting. Each top level list is a different data type, and each data point is a two-tuple of the coordinates of a point.

So if you had two distributions of points, it should look like:

```
display_info = [[(1, 2), (3, 4)],
                [(5, 6), (7, 8)]]
```

If everything is just one set of points, display_info can look like:

```
display_info = [[(1, 2), (3, 4), (5, 6)]]
```

__init__(*output_format='pdf'*)

Initialize the class.

draw_to_file(*output_file*, *title*)

Write the comparative plot to a file.

Arguments:

- output_file - The name of the file to output the information to, or a handle to write to.
- title - A title to display on the graphic.

Bio.Graphics.DisplayRepresentation module

Represent information for graphical display.

Classes in this module are designed to hold information in a way that makes it easy to draw graphical figures.

class Bio.Graphics.DisplayRepresentation.**ChromosomeCounts**(*segment_names*, *color_scheme=RAINBOW_COLORS*)

Bases: object

Represent a chromosome with count information.

This is used to display information about counts along a chromosome. The segments are expected to have different count information, which will be displayed using a color scheme.

I envision using this class when you think that certain regions of the chromosome will be especially abundant in the counts, and you want to pick those out.

__init__(*segment_names*, *color_scheme=RAINBOW_COLORS*)

Initialize a representation of chromosome counts.

Arguments:

- segment_names - An ordered list of all segment names along the chromosome. The count and other information will be added to these.
- color_scheme - A coloring scheme to use in the counts. This should be a dictionary mapping count ranges to colors (specified in reportlab.lib.colors).

add_count(*segment_name*, *count=1*)

Add counts to the given segment name.

Arguments:

- segment_name - The name of the segment we should add counts to. If the name is not present, a KeyError will be raised.
- count - The counts to add the current segment. This defaults to a single count.

scale_segment_value(*segment_name*, *scale_value=None*)

Divide the counts for a segment by some kind of scale value.

This is useful if segments aren't represented by raw counts, but are instead counts divided by some number.

add_label(*segment_name*, *label*)

Add a label to a specific segment.

Raises a KeyError is the specified segment name is not found.

set_scale(*segment_name*, *scale*)

Set the scale for a specific chromosome segment.

By default all segments have the same scale – this allows scaling by the size of the segment.

Raises a KeyError is the specified segment name is not found.

get_segment_info()

Retrieve the color and label info about the segments.

Returns a list consisting of two tuples specifying the counts and label name for each segment. The list is ordered according to the original listing of names. Labels are set as None if no label was specified.

fill_chromosome(*chromosome*)

Add the collected segment information to a chromosome for drawing.

Arguments:

- chromosome - A Chromosome graphics object that we can add chromosome segments to.

This creates ChromosomeSegment (and TelomereSegment) objects to fill in the chromosome. The information is derived from the label and count information, with counts transformed to the specified color map.

Returns the chromosome with all of the segments added.

Bio.Graphics.Distribution module

Display information distributed across a Chromosome-like object.

These classes are meant to show the distribution of some kind of information as it changes across any kind of segment. It was designed with chromosome distributions in mind, but could also work for chromosome regions, BAC clones or anything similar.

Reportlab is used for producing the graphical output.

class Bio.Graphics.Distribution.**DistributionPage**(*output_format='pdf'*)

Bases: object

Display a grouping of distributions on a page.

This organizes Distributions, and will display them nicely on a single page.

__init__(*output_format='pdf'*)

Initialize the class.

draw(*output_file*, *title*)

Draw out the distribution information.

Arguments:

- output_file - The name of the file to output the information to, or a handle to write to.
- title - A title to display on the graphic.

class Bio.Graphics.Distribution.**BarChartDistribution**(*display_info=None*)

Bases: object

Display the distribution of values as a bunch of bars.

__init__(*display_info=None*)

Initialize a Bar Chart display of distribution info.

Attributes:

- display_info - the information to be displayed in the distribution. This should be ordered as a list of lists, where each internal list is a data set to display in the bar chart.

draw(*cur_drawing*, *start_x*, *start_y*, *end_x*, *end_y*)

Draw a bar chart with the info in the specified range.

class Bio.Graphics.Distribution.**LineDistribution**

Bases: object

Display the distribution of values as connected lines.

This distribution displays the change in values across the object as lines. This also allows multiple distributions to be displayed on a single graph.

__init__()

Initialize the class.

draw(*cur_drawing*, *start_x*, *start_y*, *end_x*, *end_y*)

Draw a line distribution into the current drawing.

Bio.Graphics.KGML_vis module

Classes and functions to visualise a KGML Pathway Map.

The KGML definition is as of release KGML v0.7.1 (http://www.kegg.jp/kegg/xml/docs/)

Classes:

Bio.Graphics.KGML_vis.**darken**(*color*, *factor=0.7*)

Return darkened color as a ReportLab RGB color.

Take a passed color and returns a Reportlab color that is darker by the factor indicated in the parameter.

Bio.Graphics.KGML_vis.**color_to_reportlab**(*color*)

Return the passed color in Reportlab Color format.

We allow colors to be specified as hex values, tuples, or Reportlab Color objects, and with or without an alpha channel. This function acts as a Rosetta stone for conversion of those formats to a Reportlab Color object, with alpha value.

Any other color specification is returned directly

Bio.Graphics.KGML_vis.**get_temp_imagefilename**(*url*)

Return filename of temporary file containing downloaded image.

Create a new temporary file to hold the image file at the passed URL and return the filename.

class Bio.Graphics.KGML_vis.**KGMLCanvas**(*pathway*, *import_imagemap=False*, *label_compounds=True*, *label_orthologs=True*, *label_reaction_entries=True*, *label_maps=True*, *show_maps=False*, *fontname='Helvetica'*, *fontsize=6*, *draw_relations=True*, *show_orthologs=True*, *show_compounds=True*, *show_genes=True*, *show_reaction_entries=True*, *margins=(0.02, 0.02)*)

Bases: `object`

Reportlab Canvas-based representation of a KGML pathway map.

__init__(*pathway*, *import_imagemap=False*, *label_compounds=True*, *label_orthologs=True*, *label_reaction_entries=True*, *label_maps=True*, *show_maps=False*, *fontname='Helvetica'*, *fontsize=6*, *draw_relations=True*, *show_orthologs=True*, *show_compounds=True*, *show_genes=True*, *show_reaction_entries=True*, *margins=(0.02, 0.02)*)

Initialize the class.

draw(*filename*)

Add the map elements to the drawing.

Module contents

Bio.Graphics offers several graphical outputs, all using ReportLab.

28.1.16 Bio.HMM package

Submodules

Bio.HMM.DynamicProgramming module

Dynamic Programming algorithms for general usage.

This module contains classes which implement Dynamic Programming algorithms that can be used generally.

class Bio.HMM.DynamicProgramming.**AbstractDPAlgorithms**(*markov_model*, *sequence*)

Bases: object

An abstract class to calculate forward and backward probabilities.

This class should not be instantiated directly, but should be used through a derived class which implements proper scaling of variables.

This class is just meant to encapsulate the basic forward and backward algorithms, and allow derived classes to deal with the problems of multiplying probabilities.

Derived class of this must implement:

- _forward_recursion – Calculate the forward values in the recursion using some kind of technique for preventing underflow errors.
- _backward_recursion – Calculate the backward values in the recursion step using some technique to prevent underflow errors.

__init__(*markov_model*, *sequence*)

Initialize to calculate forward and backward probabilities.

Arguments:

- markov_model – The current Markov model we are working with.
- sequence – A training sequence containing a set of emissions.

forward_algorithm()

Calculate sequence probability using the forward algorithm.

This implements the forward algorithm, as described on p57-58 of Durbin et al.

Returns:

- A dictionary containing the forward variables. This has keys of the form (state letter, position in the training sequence), and values containing the calculated forward variable.
- The calculated probability of the sequence.

backward_algorithm()

Calculate sequence probability using the backward algorithm.

This implements the backward algorithm, as described on p58-59 of Durbin et al.

Returns:

- A dictionary containing the backwards variables. This has keys of the form (state letter, position in the training sequence), and values containing the calculated backward variable.

class Bio.HMM.DynamicProgramming.**ScaledDPAlgorithms**(*markov_model*, *sequence*)

Bases: *AbstractDPAlgorithms*

Implement forward and backward algorithms using a rescaling approach.

This scales the f and b variables, so that they remain within a manageable numerical interval during calculations. This approach is described in Durbin et al. on p 78.

This approach is a little more straightforward then log transformation but may still give underflow errors for some types of models. In these cases, the LogDPAlgorithms class should be used.

__init__(*markov_model*, *sequence*)

Initialize the scaled approach to calculating probabilities.

Arguments:

- markov_model – The current Markov model we are working with.
- sequence – A TrainingSequence object that must have a set of emissions to work with.

__annotations__ = {}

class Bio.HMM.DynamicProgramming.**LogDPAlgorithms**(*markov_model*, *sequence*)

Bases: *AbstractDPAlgorithms*

Implement forward and backward algorithms using a log approach.

This uses the approach of calculating the sum of log probabilities using a lookup table for common values.

XXX This is not implemented yet!

__init__(*markov_model*, *sequence*)

Initialize the class.

__annotations__ = {}

Bio.HMM.MarkovModel module

Deal with representations of Markov Models.

class Bio.HMM.MarkovModel.**MarkovModelBuilder**(*state_alphabet*, *emission_alphabet*)

Bases: `object`

Interface to build up a Markov Model.

This class is designed to try to separate the task of specifying the Markov Model from the actual model itself. This is in hopes of making the actual Markov Model classes smaller.

So, this builder class should be used to create Markov models instead of trying to initiate a Markov Model directly.

DEFAULT_PSEUDO = 1

__init__(*state_alphabet*, *emission_alphabet*)

Initialize a builder to create Markov Models.

Arguments:

- state_alphabet – An iterable (e.g., tuple or list) containing all of the letters that can appear in the states

- emission_alphabet – An iterable (e.g., tuple or list) containing all of the letters for states that can be emitted by the HMM.

get_markov_model()

Return the markov model corresponding with the current parameters.

Each markov model returned by a call to this function is unique (ie. they don't influence each other).

set_initial_probabilities(*initial_prob*)

Set initial state probabilities.

initial_prob is a dictionary mapping states to probabilities. Suppose, for example, that the state alphabet is ('A', 'B'). Call set_initial_prob({'A': 1}) to guarantee that the initial state will be 'A'. Call set_initial_prob({'A': 0.5, 'B': 0.5}) to make each initial state equally probable.

This method must now be called in order to use the Markov model because the calculation of initial probabilities has changed incompatibly; the previous calculation was incorrect.

If initial probabilities are set for all states, then they should add up to 1. Otherwise the sum should be <= 1. The residual probability is divided up evenly between all the states for which the initial probability has not been set. For example, calling set_initial_prob({}) results in P('A') = 0.5 and P('B') = 0.5, for the above example.

set_equal_probabilities()

Reset all probabilities to be an average value.

Resets the values of all initial probabilities and all allowed transitions and all allowed emissions to be equal to 1 divided by the number of possible elements.

This is useful if you just want to initialize a Markov Model to starting values (ie. if you have no prior notions of what the probabilities should be – or if you are just feeling too lazy to calculate them :-).

Warning 1 – this will reset all currently set probabilities.

Warning 2 – This just sets all probabilities for transitions and emissions to total up to 1, so it doesn't ensure that the sum of each set of transitions adds up to 1.

set_random_initial_probabilities()

Set all initial state probabilities to a randomly generated distribution.

Returns the dictionary containing the initial probabilities.

set_random_transition_probabilities()

Set all allowed transition probabilities to a randomly generated distribution.

Returns the dictionary containing the transition probabilities.

set_random_emission_probabilities()

Set all allowed emission probabilities to a randomly generated distribution.

Returns the dictionary containing the emission probabilities.

set_random_probabilities()

Set all probabilities to randomly generated numbers.

Resets probabilities of all initial states, transitions, and emissions to random values.

allow_all_transitions()

Create transitions between all states.

By default all transitions within the alphabet are disallowed; this is a convenience function to change this to allow all possible transitions.

allow_transition(*from_state*, *to_state*, *probability=None*, *pseudocount=None*)

Set a transition as being possible between the two states.

probability and pseudocount are optional arguments specifying the probabilities and pseudo counts for the transition. If these are not supplied, then the values are set to the default values.

Raises: KeyError – if the two states already have an allowed transition.

destroy_transition(*from_state*, *to_state*)

Restrict transitions between the two states.

Raises: KeyError if the transition is not currently allowed.

set_transition_score(*from_state*, *to_state*, *probability*)

Set the probability of a transition between two states.

Raises: KeyError if the transition is not allowed.

set_transition_pseudocount(*from_state*, *to_state*, *count*)

Set the default pseudocount for a transition.

To avoid computational problems, it is helpful to be able to set a 'default' pseudocount to start with for estimating transition and emission probabilities (see p62 in Durbin et al for more discussion on this. By default, all transitions have a pseudocount of 1.

Raises: KeyError if the transition is not allowed.

set_emission_score(*seq_state*, *emission_state*, *probability*)

Set the probability of a emission from a particular state.

Raises: KeyError if the emission from the given state is not allowed.

set_emission_pseudocount(*seq_state*, *emission_state*, *count*)

Set the default pseudocount for an emission.

To avoid computational problems, it is helpful to be able to set a 'default' pseudocount to start with for estimating transition and emission probabilities (see p62 in Durbin et al for more discussion on this. By default, all emissions have a pseudocount of 1.

Raises: KeyError if the emission from the given state is not allowed.

class Bio.HMM.MarkovModel.**HiddenMarkovModel**(*state_alphabet*, *emission_alphabet*, *initial_prob*, *transition_prob*, *emission_prob*, *transition_pseudo*, *emission_pseudo*)

Bases: object

Represent a hidden markov model that can be used for state estimation.

__init__(*state_alphabet*, *emission_alphabet*, *initial_prob*, *transition_prob*, *emission_prob*, *transition_pseudo*, *emission_pseudo*)

Initialize a Markov Model.

Note: You should use the MarkovModelBuilder class instead of initiating this class directly.

Arguments:

- state_alphabet – A tuple containing all of the letters that can appear in the states.
- emission_alphabet – A tuple containing all of the letters for states that can be emitted by the HMM.
- initial_prob - A dictionary of initial probabilities for all states.
- transition_prob – A dictionary of transition probabilities for all possible transitions in the sequence.

- emission_prob – A dictionary of emission probabilities for all possible emissions from the sequence states.
- transition_pseudo – Pseudo-counts to be used for the transitions, when counting for purposes of estimating transition probabilities.
- emission_pseudo – Pseudo-counts to be used for the emissions, when counting for purposes of estimating emission probabilities.

`get_blank_transitions`()

Get the default transitions for the model.

Returns a dictionary of all of the default transitions between any two letters in the sequence alphabet. The dictionary is structured with keys as (letter1, letter2) and values as the starting number of transitions.

`get_blank_emissions`()

Get the starting default emissions for each sequence.

This returns a dictionary of the default emissions for each letter. The dictionary is structured with keys as (seq_letter, emission_letter) and values as the starting number of emissions.

`transitions_from`(*state_letter*)

Get all destination states which can transition from source state_letter.

This returns all letters which the given state_letter can transition to, i.e. all the destination states reachable from state_letter.

An empty list is returned if state_letter has no outgoing transitions.

`transitions_to`(*state_letter*)

Get all source states which can transition to destination state_letter.

This returns all letters which the given state_letter is reachable from, i.e. all the source states which can reach state_later

An empty list is returned if state_letter is unreachable.

`viterbi`(*sequence*, *state_alphabet*)

Calculate the most probable state path using the Viterbi algorithm.

This implements the Viterbi algorithm (see pgs 55-57 in Durbin et al for a full explanation – this is where I took my implementation ideas from), to allow decoding of the state path, given a sequence of emissions.

Arguments:

- sequence – A Seq object with the emission sequence that we want to decode.
- state_alphabet – An iterable (e.g., tuple or list) containing all of the letters that can appear in the states

Bio.HMM.Trainer module

Provide trainers which estimate parameters based on training sequences.

These should be used to 'train' a Markov Model prior to actually using it to decode state paths. When supplied training sequences and a model to work from, these classes will estimate parameters of the model.

This aims to estimate two parameters:

- a_{kl} – the number of times there is a transition from k to l in the training data.
- e_{k}(b) – the number of emissions of the state b from the letter k in the training data.

class Bio.HMM.Trainer.**TrainingSequence**(*emissions*, *state_path*)

Bases: object

Hold a training sequence with emissions and optionally, a state path.

__init__(*emissions*, *state_path*)

Initialize a training sequence.

Arguments:

- emissions - An iterable (e.g., a tuple, list, or Seq object) containing the sequence of emissions in the training sequence.
- state_path - An iterable (e.g., a tuple or list) containing the sequence of states. If there is no known state path, then the sequence of states should be an empty iterable.

class Bio.HMM.Trainer.**AbstractTrainer**(*markov_model*)

Bases: object

Provide generic functionality needed in all trainers.

__init__(*markov_model*)

Initialize the class.

log_likelihood(*probabilities*)

Calculate the log likelihood of the training seqs.

Arguments:

- probabilities – A list of the probabilities of each training sequence under the current parameters, calculated using the forward algorithm.

estimate_params(*transition_counts*, *emission_counts*)

Get a maximum likelihood estimation of transition and emission.

Arguments:

- transition_counts – A dictionary with the total number of counts of transitions between two states.
- emissions_counts – A dictionary with the total number of counts of emissions of a particular emission letter by a state letter.

This then returns the maximum likelihood estimators for the transitions and emissions, estimated by formulas 3.18 in Durbin et al:

```
a_{kl} = A_{kl} / sum(A_{kl'})
e_{k}(b) = E_{k}(b) / sum(E_{k}(b'))
```

Returns: Transition and emission dictionaries containing the maximum likelihood estimators.

ml_estimator(*counts*)

Calculate the maximum likelihood estimator.

This can calculate maximum likelihoods for both transitions and emissions.

Arguments:

- counts – A dictionary of the counts for each item.

See estimate_params for a description of the formula used for calculation.

class Bio.HMM.Trainer.**BaumWelchTrainer**(*markov_model*)

Bases: *AbstractTrainer*

Trainer that uses the Baum-Welch algorithm to estimate parameters.

These should be used when a training sequence for an HMM has unknown paths for the actual states, and you need to make an estimation of the model parameters from the observed emissions.

This uses the Baum-Welch algorithm, first described in Baum, L.E. 1972. Inequalities. 3:1-8 This is based on the description in 'Biological Sequence Analysis' by Durbin et al. in section 3.3

This algorithm is guaranteed to converge to a local maximum, but not necessarily to the global maxima, so use with care!

__init__(*markov_model*)

Initialize the trainer.

Arguments:

- markov_model - The model we are going to estimate parameters for. This should have the parameters with some initial estimates, that we can build from.

train(*training_seqs*, *stopping_criteria*, *dp_method=ScaledDPAlgorithms*)

Estimate the parameters using training sequences.

The algorithm for this is taken from Durbin et al. p64, so this is a good place to go for a reference on what is going on.

Arguments:

- training_seqs – A list of TrainingSequence objects to be used for estimating the parameters.
- stopping_criteria – A function, that when passed the change in log likelihood and threshold, will indicate if we should stop the estimation iterations.
- dp_method – A class instance specifying the dynamic programming implementation we should use to calculate the forward and backward variables. By default, we use the scaling method.

update_transitions(*transition_counts*, *training_seq*, *forward_vars*, *backward_vars*, *training_seq_prob*)

Add the contribution of a new training sequence to the transitions.

Arguments:

- transition_counts – A dictionary of the current counts for the transitions
- training_seq – The training sequence we are working with
- forward_vars – Probabilities calculated using the forward algorithm.
- backward_vars – Probabilities calculated using the backwards algorithm.
- training_seq_prob - The probability of the current sequence.

This calculates A_{kl} (the estimated transition counts from state k to state l) using formula 3.20 in Durbin et al.

update_emissions(*emission_counts*, *training_seq*, *forward_vars*, *backward_vars*, *training_seq_prob*)

Add the contribution of a new training sequence to the emissions.

Arguments:

- emission_counts – A dictionary of the current counts for the emissions
- training_seq – The training sequence we are working with
- forward_vars – Probabilities calculated using the forward algorithm.

- backward_vars – Probabilities calculated using the backwards algorithm.
- training_seq_prob - The probability of the current sequence.

This calculates E_{k}(b) (the estimated emission probability for emission letter b from state k) using formula 3.21 in Durbin et al.

__annotations__ = {}

class Bio.HMM.Trainer.**KnownStateTrainer**(*markov_model*)

Bases: *AbstractTrainer*

Estimate probabilities with known state sequences.

This should be used for direct estimation of emission and transition probabilities when both the state path and emission sequence are known for the training examples.

__init__(*markov_model*)

Initialize the class.

train(*training_seqs*)

Estimate the Markov Model parameters with known state paths.

This trainer requires that both the state and the emissions are known for all of the training sequences in the list of TrainingSequence objects. This training will then count all of the transitions and emissions, and use this to estimate the parameters of the model.

__annotations__ = {}

Bio.HMM.Utilities module

Generic functions which are useful for working with HMMs.

This just collects general functions which you might like to use in dealing with HMMs.

Bio.HMM.Utilities.**pretty_print_prediction**(*emissions*, *real_state*, *predicted_state*, *emission_title='Emissions'*, *real_title='Real State'*, *predicted_title='Predicted State'*, *line_width=75*)

Print out a state sequence prediction in a nice manner.

Arguments:

- emissions – The sequence of emissions of the sequence you are dealing with.
- real_state – The actual state path that generated the emissions.
- predicted_state – A state path predicted by some kind of HMM model.

Module contents

A selection of Hidden Markov Model code.

28.1.17 Bio.KEGG package

Subpackages

Bio.KEGG.Compound package

Module contents

Code to work with the KEGG Ligand/Compound database.

Functions:

- parse - Returns an iterator giving Record objects.

Classes:

- Record - A representation of a KEGG Ligand/Compound.

`class Bio.KEGG.Compound.Record`

Bases: `object`

Holds info from a KEGG Ligand/Compound record.

Attributes:

- entry The entry identifier.
- name A list of the compound names.
- formula The chemical formula for the compound
- mass The molecular weight for the compound
- pathway A list of 3-tuples: ('PATH', pathway id, pathway)
- enzyme A list of the EC numbers.
- structures A list of 2-tuples: (database, list of struct ids)
- dblinks A list of 2-tuples: (database, list of link ids)

`__init__()`

Initialize as new record.

`__str__()`

Return a string representation of this Record.

`Bio.KEGG.Compound.parse(`*handle*`)`

Parse a KEGG Ligan/Compound file, returning Record objects.

This is an iterator function, typically used in a for loop. For example, using one of the example KEGG files in the Biopython test suite,

```
>>> with open("KEGG/compound.sample") as handle:
...     for record in parse(handle):
...         print("%s %s" % (record.entry, record.name[0]))
...
C00023 Iron
C00017 Protein
C00099 beta-Alanine
C00294 Inosine
```

(continues on next page)

(continued from previous page)

```
C00298 Trypsin
C00348 all-trans-Undecaprenyl phosphate
C00349 2-Methyl-3-oxopropanoate
C01386 NH2Mec
```

Bio.KEGG.Enzyme package

Module contents

Code to work with the KEGG Enzyme database.

Functions:

- parse - Returns an iterator giving Record objects.

Classes:

- Record - Holds the information from a KEGG Enzyme record.

class Bio.KEGG.Enzyme.**Record**

Bases: object

Holds info from a KEGG Enzyme record.

Attributes:

- entry The EC number (without the 'EC ').
- name A list of the enzyme names.
- classname A list of the classification terms.
- sysname The systematic name of the enzyme.
- reaction A list of the reaction description strings.
- substrate A list of the substrates.
- product A list of the products.
- inhibitor A list of the inhibitors.
- cofactor A list of the cofactors.
- effector A list of the effectors.
- comment A list of the comment strings.
- pathway A list of 3-tuples: (database, id, pathway)
- genes A list of 2-tuples: (organism, list of gene ids)
- disease A list of 3-tuples: (database, id, disease)
- structures A list of 2-tuples: (database, list of struct ids)
- dblinks A list of 2-tuples: (database, list of db ids)

__init__()

Initialize a new Record.

__str__()

Return a string representation of this Record.

Bio.KEGG.Enzyme.**parse**(*handle*)

Parse a KEGG Enzyme file, returning Record objects.

This is an iterator function, typically used in a for loop. For example, using one of the example KEGG files in the Biopython test suite,

```
>>> with open("KEGG/enzyme.sample") as handle:
...     for record in parse(handle):
...         print("%s %s" % (record.entry, record.name[0]))
...
1.1.1.1 alcohol dehydrogenase
1.1.1.62 17beta-estradiol 17-dehydrogenase
1.1.1.68 Transferred to 1.5.1.20
1.6.5.3 NADH:ubiquinone reductase (H+-translocating)
1.14.13.28 3,9-dihydroxypterocarpan 6a-monooxygenase
2.4.1.68 glycoprotein 6-alpha-L-fucosyltransferase
3.1.1.6 acetylesterase
2.7.2.1 acetate kinase
```

Bio.KEGG.Enzyme.**read**(*handle*)

Parse a KEGG Enzyme file with exactly one entry.

If the handle contains no records, or more than one record, an exception is raised. For example:

```
>>> with open("KEGG/enzyme.new") as handle:
...     record = read(handle)
...     print("%s %s" % (record.entry, record.name[0]))
...
6.2.1.25 benzoate---CoA ligase
```

Bio.KEGG.Gene package

Module contents

Code to work with the KEGG Gene database.

Functions: - parse - Returns an iterator giving Record objects.

Classes: - Record - A representation of a KEGG Gene.

class Bio.KEGG.Gene.**Record**

Bases: `object`

Holds info from a KEGG Gene record.

Attributes:

- entry The entry identifier.
- name A list of the gene names.
- definition The definition for the gene.
- orthology A list of 2-tuples: (orthology id, role)

- organism A tuple: (organism id, organism)
- position The position for the gene
- motif A list of 2-tuples: (database, list of link ids)
- dblinks A list of 2-tuples: (database, list of link ids)

__init__()

Initialize new record.

__str__()

Return a string representation of this Record.

Bio.KEGG.Gene.**parse**(*handle*)

Parse a KEGG Gene file, returning Record objects.

This is an iterator function, typically used in a for loop. For example, using one of the example KEGG files in the Biopython test suite,

```
>>> with open("KEGG/gene.sample") as handle:
...     for record in parse(handle):
...         print("%s %s" % (record.entry, record.name[0]))
...
b1174 minE
b1175 minD
```

Bio.KEGG.KGML package

Submodules

Bio.KEGG.KGML.KGML_parser module

Classes and functions to parse a KGML pathway map.

The KGML pathway map is parsed into the object structure defined in KGML_Pathway.py in this module.

Classes:

- KGMLParser - Parses KGML file

Functions:

- read - Returns a single Pathway object, using KGMLParser internally

Bio.KEGG.KGML.KGML_parser.**read**(*handle*)

Parse a single KEGG Pathway from given file handle.

Returns a single Pathway object. There should be one and only one pathway in each file, but there may well be pathological examples out there.

Bio.KEGG.KGML.KGML_parser.**parse**(*handle*)

Return an iterator over Pathway elements.

Arguments:

- handle - file handle to a KGML file for parsing, or a KGML string

This is a generator for the return of multiple Pathway objects.

class Bio.KEGG.KGML.KGML_parser.**KGMLParser**(*elem*)

Bases: object

Parses a KGML XML Pathway entry into a Pathway object.

Example: Read and parse large metabolism file

```
>>> from Bio.KEGG.KGML.KGML_parser import read
>>> pathway = read(open('KEGG/ko01100.xml', 'r'))
>>> print(len(pathway.entries))
3628
>>> print(len(pathway.reactions))
1672
>>> print(len(pathway.maps))
149
```

```
>>> pathway = read(open('KEGG/ko00010.xml', 'r'))
>>> print(pathway)
Pathway: Glycolysis / Gluconeogenesis
KEGG ID: path:ko00010
Image file: http://www.kegg.jp/kegg/pathway/ko/ko00010.png
Organism: ko
Entries: 99
Entry types:
    ortholog: 61
    compound: 31
    map: 7
```

__init__(*elem*)

Initialize the class.

parse()

Parse the input elements.

Bio.KEGG.KGML.KGML_pathway module

Classes to represent a KGML Pathway Map.

The KGML definition is as of release KGML v0.7.2 (http://www.kegg.jp/kegg/xml/docs/)

Classes:

- Pathway - Specifies graph information for the pathway map
- Relation - Specifies a relationship between two proteins or KOs, or protein and compound. There is an implied direction to the relationship in some cases.
- Reaction - A specific chemical reaction between a substrate and a product.
- Entry - A node in the pathway graph
- Graphics - Entry subelement describing its visual representation

class Bio.KEGG.KGML.KGML_pathway.**Pathway**

Bases: object

Represents a KGML pathway from KEGG.

Specifies graph information for the pathway map, as described in release KGML v0.7.2 (http://www.kegg.jp/kegg/xml/docs/)

Attributes:

- name - KEGGID of the pathway map
- org - ko/ec/[org prefix]
- number - map number (integer)
- title - the map title
- image - URL of the image map for the pathway
- link - URL of information about the pathway
- entries - Dictionary of entries in the pathway, keyed by node ID
- reactions - Set of reactions in the pathway

The name attribute has a restricted format, so we make it a property and enforce the formatting.

The Pathway object is the only allowed route for adding/removing Entry, Reaction, or Relation elements.

Entries are held in a dictionary and keyed by the node ID for the pathway graph - this allows for ready access via the Reaction/Relation etc. elements. Entries must be added before reference by any other element.

Reactions are held in a dictionary, keyed by node ID for the path. The elements referred to in the reaction must be added before the reaction itself.

__init__()
Initialize the class.

get_KGML()
Return the pathway as a string in prettified KGML format.

add_entry(*entry*)
Add an Entry element to the pathway.

remove_entry(*entry*)
Remove an Entry element from the pathway.

add_reaction(*reaction*)
Add a Reaction element to the pathway.

remove_reaction(*reaction*)
Remove a Reaction element from the pathway.

add_relation(*relation*)
Add a Relation element to the pathway.

remove_relation(*relation*)
Remove a Relation element from the pathway.

__str__()
Return a readable summary description string.

property name
The KEGGID for the pathway map.

property number
The KEGG map number.

property compounds

Get a list of entries of type compound.

property maps

Get a list of entries of type map.

property orthologs

Get a list of entries of type ortholog.

property genes

Get a list of entries of type gene.

property reactions

Get a list of reactions in the pathway.

property reaction_entries

List of entries corresponding to each reaction in the pathway.

property relations

Get a list of relations in the pathway.

property element

Return the Pathway as a valid KGML element.

property bounds

Coordinate bounds for all Graphics elements in the Pathway.

Returns the [(xmin, ymin), (xmax, ymax)] coordinates for all Graphics elements in the Pathway

class Bio.KEGG.KGML.KGML_pathway.**Entry**

Bases: object

Represent an Entry from KGML.

Each Entry element is a node in the pathway graph, as described in release KGML v0.7.2 (http://www.kegg.jp/kegg/xml/docs/)

Attributes:

- id - The ID of the entry in the pathway map (integer)
- names - List of KEGG IDs for the entry
- type - The type of the entry
- link - URL of information about the entry
- reaction - List of KEGG IDs of the corresponding reactions (integer)
- graphics - List of Graphics objects describing the Entry's visual representation
- components - List of component node ID for this Entry ('group')
- alt - List of alternate names for the Entry

NOTE: The alt attribute represents a subelement of the substrate and product elements in the KGML file

__init__()

Initialize the class.

__str__()

Return readable descriptive string.

add_component(*element*)

Add an element to the entry.

If the Entry is already part of a pathway, make sure the component already exists.

remove_component(*value*)

Remove the entry with the passed ID from the group.

add_graphics(*entry*)

Add the Graphics entry.

remove_graphics(*entry*)

Remove the Graphics entry with the passed ID from the group.

property name

List of KEGG identifiers for the Entry.

property reaction

List of reaction KEGG IDs for this Entry.

property id

The pathway graph node ID for the Entry.

property element

Return the Entry as a valid KGML element.

property bounds

Coordinate bounds for all Graphics elements in the Entry.

Return the [(xmin, ymin), (xmax, ymax)] coordinates for the Entry Graphics elements.

property is_reactant

Return true if this Entry participates in any reaction in its parent pathway.

class Bio.KEGG.KGML.KGML_pathway.**Component**(*parent*)

Bases: `object`

An Entry subelement used to represents a complex node.

A subelement of the Entry element, used when the Entry is a complex node, as described in release KGML v0.7.2 (http://www.kegg.jp/kegg/xml/docs/)

The Component acts as a collection (with type 'group', and typically its own Graphics subelement), having only an ID.

__init__(*parent*)

Initialize the class.

property id

The pathway graph node ID for the Entry

property element

Return the Component as a valid KGML element.

class Bio.KEGG.KGML.KGML_pathway.**Graphics**(*parent*)

Bases: `object`

An Entry subelement used to represents the visual representation.

A subelement of Entry, specifying its visual representation, as described in release KGML v0.7.2 (http://www.kegg.jp/kegg/xml/docs/)

Attributes:

- name Label for the graphics object
- x X-axis position of the object (int)
- y Y-axis position of the object (int)
- coords polyline coordinates, list of (int, int) tuples
- type object shape
- width object width (int)
- height object height (int)
- fgcolor object foreground color (hex RGB)
- bgcolor object background color (hex RGB)

Some attributes are present only for specific graphics types. For example, line types do not (typically) have a width. We permit non-DTD attributes and attribute settings, such as

dash List of ints, describing an on/off pattern for dashes

__init__(*parent*)

Initialize the class.

property x

The X coordinate for the graphics element.

property y

The Y coordinate for the graphics element.

property width

The width of the graphics element.

property height

The height of the graphics element.

property coords

Polyline coordinates for the graphics element.

property fgcolor

Foreground color.

property bgcolor

Background color.

property element

Return the Graphics as a valid KGML element.

property bounds

Coordinate bounds for the Graphics element.

Return the bounds of the Graphics object as an [(xmin, ymin), (xmax, ymax)] tuple. Coordinates give the centre of the circle, rectangle, roundrectangle elements, so we have to adjust for the relevant width/height.

property centre

Return the centre of the Graphics object as an (x, y) tuple.

class Bio.KEGG.KGML.KGML_pathway.**Reaction**

Bases: object

A specific chemical reaction with substrates and products.

This describes a specific chemical reaction between one or more substrates and one or more products.

Attributes:

- id Pathway graph node ID of the entry
- names List of KEGG identifier(s) from the REACTION database
- type String: reversible or irreversible
- substrate Entry object of the substrate
- product Entry object of the product

__init__()

Initialize the class.

__str__()

Return an informative human-readable string.

add_substrate(*substrate_id*)

Add a substrate, identified by its node ID, to the reaction.

add_product(*product_id*)

Add a product, identified by its node ID, to the reaction.

property id

Node ID for the reaction.

property name

List of KEGG identifiers for the reaction.

property substrates

Return list of substrate Entry elements.

property products

Return list of product Entry elements.

property entry

Return the Entry corresponding to this reaction.

property reactant_ids

Return a list of substrate and product reactant IDs.

property element

Return KGML element describing the Reaction.

class Bio.KEGG.KGML.KGML_pathway.**Relation**

Bases: object

A relationship between to products, KOs, or protein and compound.

This describes a relationship between two products, KOs, or protein and compound, as described in release KGML v0.7.2 (http://www.kegg.jp/kegg/xml/docs/)

Attributes:

- entry1 - The first Entry object node ID defining the relation (int)

- entry2 - The second Entry object node ID defining the relation (int)
- type - The relation type
- subtypes - List of subtypes for the relation, as a list of (name, value) tuples

__init__()
Initialize the class.

__str__()
Return a useful human-readable string.

property entry1
Entry1 of the relation.

property entry2
Entry2 of the relation.

property element
Return KGML element describing the Relation.

Module contents

Code to work with data from the KEGG database.

References: Kanehisa, M. and Goto, S.; KEGG: Kyoto Encyclopedia of Genes and Genomes. Nucleic Acids Res. 28, 29-34 (2000).

URL: http://www.genome.ad.jp/kegg/

Bio.KEGG.Map package

Module contents

Load KEGG Pathway maps for use with the Biopython Pathway module.

The pathway maps are in the format:

```
RXXXXX:[X.X.X.X:] A + 2 B <=> C
RXXXXX:[X.X.X.X:] 3C <=> 2 D + E
...
```

where RXXXXX is a five-digit reaction id, and X.X.X.X is the optional EC number of the enzyme that catalyze the reaction.

Bio.KEGG.Map.**parse**(*handle*)
Parse a KEGG pathway map.

Submodules

Bio.KEGG.REST module

Provides code to access the REST-style KEGG online API.

This module aims to make the KEGG online REST-style API easier to use. See: https://www.kegg.jp/kegg/rest/keggapi.html

The KEGG REST-style API provides simple access to a range of KEGG databases. This works using simple URLs (which this module will construct for you), with any errors indicated via HTTP error levels.

The functionality is somewhat similar to Biopython's Bio.TogoWS and Bio.Entrez modules.

Currently KEGG does not provide any usage guidelines (unlike the NCBI whose requirements are reasonably clear). To avoid risking overloading the service, Biopython will only allow three calls per second.

References: Kanehisa, M. and Goto, S.; KEGG: Kyoto Encyclopedia of Genes and Genomes. Nucleic Acids Res. 28, 29-34 (2000).

Bio.KEGG.REST.**kegg_info**(*database*)

KEGG info - Displays the current statistics of a given database.

db - database or organism (string)

The argument db can be a KEGG database name (e.g. 'pathway' or its official abbreviation, 'path'), or a KEGG organism code or T number (e.g. 'hsa' or 'T01001' for human).

A valid list of organism codes and their T numbers can be obtained via kegg_info('organism') or https://rest.kegg.jp/list/organism

Bio.KEGG.REST.**kegg_list**(*database*, *org=None*)

KEGG list - Entry list for database, or specified database entries.

db - database or organism (string) org - optional organism (string), see below.

For the pathway and module databases the optional organism can be used to restrict the results.

Bio.KEGG.REST.**kegg_find**(*database*, *query*, *option=None*)

KEGG find - Data search.

Finds entries with matching query keywords or other query data in a given database.

db - database or organism (string) query - search terms (string) option - search option (string), see below.

For the compound and drug database, set option to the string 'formula', 'exact_mass' or 'mol_weight' to search on that field only. The chemical formula search is a partial match irrespective of the order of atoms given. The exact mass (or molecular weight) is checked by rounding off to the same decimal place as the query data. A range of values may also be specified with the minus(-) sign.

Bio.KEGG.REST.**kegg_get**(*dbentries*, *option=None*)

KEGG get - Data retrieval.

dbentries - Identifiers (single string, or list of strings), see below. option - One of "aaseq", "ntseq", "mol", "kcf", "image", "kgml" (string)

The input is limited up to 10 entries. The input is limited to one pathway entry with the image or kgml option. The input is limited to one compound/glycan/drug entry with the image option.

Returns a handle.

Bio.KEGG.REST.**kegg_conv**(*target_db*, *source_db*, *option=None*)

KEGG conv - convert KEGG identifiers to/from outside identifiers.

Arguments:

- target_db - Target database
- source_db_or_dbentries - source database or database entries
- option - Can be "turtle" or "n-triple" (string).

Bio.KEGG.REST.**kegg_link**(*target_db*, *source_db*, *option=None*)

KEGG link - find related entries by using database cross-references.

target_db - Target database source_db_or_dbentries - source database option - Can be "turtle" or "n-triple" (string).

Module contents

Code to work with data from the KEGG database.

References: Kanehisa, M. and Goto, S.; KEGG: Kyoto Encyclopedia of Genes and Genomes. Nucleic Acids Res. 28, 29-34 (2000).

URL: http://www.genome.ad.jp/kegg/

28.1.18 Bio.Medline package

Module contents

Code to work with Medline from the NCBI.

Classes:

- Record A dictionary holding Medline data.

Functions:

- read Reads one Medline record
- parse Allows you to iterate over a bunch of Medline records

class Bio.Medline.**Record**

Bases: dict

A dictionary holding information from a Medline record.

All data are stored under the mnemonic appearing in the Medline file. These mnemonics have the following interpretations:

Mnemonic	Description
AB	Abstract
CI	Copyright Information
AD	Affiliation
IRAD	Investigator Affiliation
AID	Article Identifier
AU	Author
FAU	Full Author

continues on next page

Table 1 – continued from previous page

CN	Corporate Author
DCOM	Date Completed
DA	Date Created
LR	Date Last Revised
DEP	Date of Electronic Publication
DP	Date of Publication
EDAT	Entrez Date
GS	Gene Symbol
GN	General Note
GR	Grant Number
IR	Investigator Name
FIR	Full Investigator Name
IS	ISSN
IP	Issue
TA	Journal Title Abbreviation
JT	Journal Title
LA	Language
LID	Location Identifier
MID	Manuscript Identifier
MHDA	MeSH Date
MH	MeSH Terms
JID	NLM Unique ID
RF	Number of References
OAB	Other Abstract
OCI	Other Copyright Information
OID	Other ID
OT	Other Term
OTO	Other Term Owner
OWN	Owner
PG	Pagination
PS	Personal Name as Subject
FPS	Full Personal Name as Subject
PL	Place of Publication
PHST	Publication History Status
PST	Publication Status
PT	Publication Type
PUBM	Publishing Model
PMC	PubMed Central Identifier
PMID	PubMed Unique Identifier
RN	Registry Number/EC Number
NM	Substance Name
SI	Secondary Source ID
SO	Source
SFM	Space Flight Mission
STAT	Status
SB	Subset
TI	Title
TT	Transliterated Title
VI	Volume
CON	Comment on
CIN	Comment in
EIN	Erratum in

continues on next page

Table 1 – continued from previous page

EFR	Erratum for
CRI	Corrected and Republished in
CRF	Corrected and Republished from
PRIN	Partial retraction in
PROF	Partial retraction of
RPI	Republished in
RPF	Republished from
RIN	Retraction in
ROF	Retraction of
UIN	Update in
UOF	Update of
SPIN	Summary for patients in
ORI	Original report in

Bio.Medline.**parse**(*handle*)

Read Medline records one by one from the handle.

The handle is either is a Medline file, a file-like object, or a list of lines describing one or more Medline records.

Typical usage:

```
>>> from Bio import Medline
>>> with open("Medline/pubmed_result2.txt") as handle:
...     records = Medline.parse(handle)
...     for record in records:
...         print(record['TI'])
...
A high level interface to SCOP and ASTRAL ...
GenomeDiagram: a python package for the visualization of ...
Open source clustering software.
PDB file parser and structure class implemented in Python.
```

Bio.Medline.**read**(*handle*)

Read a single Medline record from the handle.

The handle is either is a Medline file, a file-like object, or a list of lines describing a Medline record.

Typical usage:

```
>>> from Bio import Medline
>>> with open("Medline/pubmed_result1.txt") as handle:
...     record = Medline.read(handle)
...     print(record['TI'])
...
The Bio* toolkits--a brief overview.
```

28.1.19 Bio.NMR package

Submodules

Bio.NMR.NOEtools module

NOEtools: For predicting NOE coordinates from assignment data.

The input and output are modelled on nmrview peaklists. This modules is suitable for directly generating an nmrview peaklist with predicted crosspeaks directly from the input assignment peaklist.

`Bio.NMR.NOEtools.`**`predictNOE`**(*peaklist*, *originNuc*, *detectedNuc*, *originResNum*, *toResNum*)

Predict the i->j NOE position based on self peak (diagonal) assignments.

Parameters

peaklist
[xprtools.Peaklist] List of peaks from which to derive predictions

originNuc
[str] Name of originating nucleus.

originResNum
[int] Index of originating residue.

detectedNuc
[str] Name of detected nucleus.

toResNum
[int] Index of detected residue.

Returns

returnLine
[str] The .xpk file entry for the predicted crosspeak.

Notes

The initial peaklist is assumed to be diagonal (self peaks only) and currently there is no checking done to insure that this assumption holds true. Check your peaklist for errors and off diagonal peaks before attempting to use predictNOE.

Examples

Using predictNOE(peaklist,"N15","H1",10,12) where peaklist is of the type xpktools.peaklist would generate a .xpk file entry for a crosspeak that originated on N15 of residue 10 and ended up as magnetization detected on the H1 nucleus of residue 12

Bio.NMR.xpktools module

Tools to manipulate data from nmrview .xpk peaklist files.

class Bio.NMR.xpktools.**XpkEntry**(*entry*, *headline*)

Bases: object

Provide dictionary access to single entry from nmrview .xpk file.

This class is suited for handling single lines of non-header data from an nmrview .xpk file. This class provides methods for extracting data by the field name which is listed in the last line of the peaklist header.

Parameters

xpkentry
[str] The line from an nmrview .xpk file.

xpkheadline
[str] The line from the header file that gives the names of the entries. This is typically the sixth line of the header, 1-origin.

Attributes

fields
[dict] Dictionary of fields where key is in header line, value is an entry. Variables are accessed by either their name in the header line as in self.field["H1.P"] will return the H1.P entry for example. self.field["entrynum"] returns the line number (1st field of line)

__init__(*entry*, *headline*)

Initialize the class.

class Bio.NMR.xpktools.**Peaklist**(*infn*)

Bases: object

Provide access to header lines and data from a nmrview xpk file.

Header file lines and file data are available as attributes.

Parameters

infn
[str] The input nmrview filename.

Examples

```
>>> from Bio.NMR.xpktools import Peaklist
>>> peaklist = Peaklist('../Doc/examples/nmr/noed.xpk')
>>> peaklist.firstline
'label dataset sw sf '
>>> peaklist.dataset
'test.nv'
>>> peaklist.sf
'{599.8230 } { 60.7860 } { 60.7860 }'
>>> peaklist.datalabels
' H1.L  H1.P  H1.W  H1.B  H1.E  H1.J  15N2.L  15N2.P  15N2.W  15N2.B  15N2.E  15N2.
↪J  N15.L  N15.P  N15.W  N15.B  N15.E  N15.J  vol  int  stat '
```

Attributes

firstline
[str] The first line in the header.

axislabels
[str] The axis labels.

dataset
[str] The label of the dataset.

sw
[str] The sw coordinates.

sf
[str] The sf coordinates.

datalabels
[str] The labels of the entries.

data
[list] File data after header lines.

__init__(*infn*)

Initialize the class.

residue_dict(*index*)

Return a dict of lines in 'data' indexed by residue number or a nucleus.

The nucleus should be given as the input argument in the same form as it appears in the xpk label line (H1, 15N for example)

Parameters

index
[str] The nucleus to index data by.

Returns

resdict
[dict] Mappings of index nucleus to data line.

Examples

```
>>> from Bio.NMR.xpktools import Peaklist
>>> peaklist = Peaklist('../Doc/examples/nmr/noed.xpk')
>>> residue_d = peaklist.residue_dict('H1')
>>> sorted(residue_d.keys())
['10', '3', '4', '5', '6', '7', '8', '9', 'maxres', 'minres']
>>> residue_d['10']
['8  10.hn   7.663   0.021   0.010   ++   0.000   10.n   118.341   0.324   0.
↪010   +E   0.000   10.n   118.476   0.324   0.010   +E   0.000  0.49840 0.
↪49840 0']
```

write_header(*outfn*)

Write header lines from input file to handle `outfn`.

Bio.NMR.xpktools.**replace_entry**(*line*, *fieldn*, *newentry*)

Replace an entry in a string by the field number.

No padding is implemented currently. Spacing will change if the original field entry and the new field entry are of different lengths.

Bio.NMR.xpktools.**data_table**(*fn_list*, *datalabel*, *keyatom*)

Generate a data table from a list of input xpk files.

Parameters

fn_list
[list] List of .xpk file names.

datalabel
[str] The data element reported.

keyatom
[str] The name of the nucleus used as an index for the data table.

Returns

outlist
[list] List of table rows indexed by `keyatom`.

Module contents

Code for working with NMR data.

This directory currently contains contributions from:

- Bob Bussell <rgb2003@med.cornell.edu> – NOEtools and xpktools

28.1.20 Bio.Nexus package

Submodules

Bio.Nexus.Nexus module

Nexus class. Parse the contents of a NEXUS file.

Based upon 'NEXUS: An extensible file format for systematic information' Maddison, Swofford, Maddison. 1997. Syst. Biol. 46(4):590-621

exception Bio.Nexus.Nexus.**NexusError**

Bases: `Exception`

Provision for the management of Nexus exceptions.

class Bio.Nexus.Nexus.**CharBuffer**(*string*)

Bases: `object`

Helps reading NEXUS-words and characters from a buffer (semi-PRIVATE).

This class is not intended for public use (any more).

__init__(*string*)

Initialize the class.

peek()

Return the first character from the buffer.

peek_nonwhitespace()

Return the first character from the buffer, do not include spaces.

__next__()

Iterate over NEXUS characters in the file.

next_nonwhitespace()

Check for next non whitespace character in NEXUS file.

skip_whitespace()

Skip whitespace characters in NEXUS file.

next_until(*target*)

Iterate over the NEXUS file until a target character is reached.

peek_word(*word*)

Return a word stored in the buffer.

next_word()

Return the next NEXUS word from a string.

This deals with single and double quotes, whitespace and punctuation.

rest()

Return the rest of the string without parsing.

class Bio.Nexus.Nexus.**StepMatrix**(*symbols*, *gap*)

Bases: object

Calculate a stepmatrix for weighted parsimony.

See : COMBINATORIAL WEIGHTS IN PHYLOGENETIC ANALYSIS - A STATISTICAL PARSIMONY PROCEDURE Wheeler (1990), Cladistics 6:269-275.

__init__(*symbols*, *gap*)

Initialize the class.

set(*x*, *y*, *value*)

Set a given value in the matrix's position.

add(*x*, *y*, *value*)

Add the given value to existing, in matrix's position.

sum()

Calculate the associations, makes matrix of associations.

transformation()

Calculate the transformation matrix.

Normalizes the columns of the matrix of associations.

weighting()

Calculate the Phylogenetic weight matrix.

Constructed from the logarithmic transformation of the transformation matrix.

smprint(*name='your_name_here'*)

Print a stepmatrix.

Bio.Nexus.Nexus.**safename**(*name*, *mrbayes=False*)

Return a taxon identifier according to NEXUS standard.

Wrap quotes around names with punctuation or whitespace, and double single quotes.

mrbayes=True: write names without quotes, whitespace or punctuation for the mrbayes software package.

Bio.Nexus.Nexus.**quotestrip**(*word*)

Remove quotes and/or double quotes around identifiers.

Bio.Nexus.Nexus.**get_start_end**(*sequence*, *skiplist=('-', '?')*)

Return position of first and last character which is not in skiplist.

Skiplist defaults to ['-','?'].

Bio.Nexus.Nexus.**combine**(*matrices*)

Combine matrices in [(name,nexus-instance),...] and return new nexus instance.

combined_matrix=combine([(name1,nexus_instance1),(name2,nexus_instance2),...] Character sets, character partitions and taxon sets are prefixed, readjusted and present in the combined matrix.

class Bio.Nexus.Nexus.**Commandline**(*line*, *title*)

Bases: object

Represent a commandline as command and options.

__init__(*line*, *title*)

Initialize the class.

class Bio.Nexus.Nexus.**Block**(*title=None*)

Bases: object

Represent a NEXUS block with block name and list of commandlines.

__init__(*title=None*)

Initialize the class.

class Bio.Nexus.Nexus.**Nexus**(*input=None*)

Bases: object

Create the Nexus class, main class for the management of Nexus files.

__init__(*input=None*)

Initialize the class.

get_original_taxon_order()

Included for backwards compatibility (DEPRECATED).

set_original_taxon_order(*value*)

Included for backwards compatibility (DEPRECATED).

property original_taxon_order

Included for backwards compatibility (DEPRECATED).

read(*input*)

Read and parse NEXUS input (a filename, file-handle, or string).

write_nexus_data_partitions(*matrix=None*, *filename=None*, *blocksize=None*, *interleave=False*, *exclude=()*, *delete=()*, *charpartition=None*, *comment=''*, *mrbayes=False*)

Write a nexus file for each partition in charpartition.

Only non-excluded characters and non-deleted taxa are included, just the data block is written.

write_nexus_data(*filename=None*, *matrix=None*, *exclude=()*, *delete=()*, *blocksize=None*, *interleave=False*, *interleave_by_partition=False*, *comment=None*, *omit_NEXUS=False*, *append_sets=True*, *mrbayes=False*, *codons_block=True*)

Write a nexus file with data and sets block to a file or handle.

Character sets and partitions are appended by default, and are adjusted according to excluded characters (i.e. character sets still point to the same sites (not necessarily same positions), without including the deleted characters.

- filename - Either a filename as a string (which will be opened, written to and closed), or a handle object (which will be written to but NOT closed).
- interleave_by_partition - Optional name of partition (string)
- omit_NEXUS - Boolean. If true, the '#NEXUS' line normally at the start of the file is omitted.

Returns the filename/handle used to write the data.

append_sets(*exclude=()*, *delete=()*, *mrbayes=False*, *include_codons=True*, *codons_only=False*)

Return a sets block.

export_fasta(*filename=None*, *width=70*)

Write matrix into a fasta file.

export_phylip(*filename=None*)

Write matrix into a PHYLIP file.

Note that this writes a relaxed PHYLIP format file, where the names are not truncated, nor checked for invalid characters.

constant(*matrix=None*, *delete=()*, *exclude=()*)

Return a list with all constant characters.

cstatus(*site*, *delete=()*, *narrow=True*)

Summarize character.

narrow=True: paup-mode (a c ? –> ac; ? ? ? –> ?) narrow=false: (a c ? –> a c g t -; ? ? ? –> a c g t -)

weighted_stepmatrix(*name='your_name_here'*, *exclude=()*, *delete=()*)

Calculate a stepmatrix for weighted parsimony.

See Wheeler (1990), Cladistics 6:269-275 and Felsenstein (1981), Biol. J. Linn. Soc. 16:183-196

crop_matrix(*matrix=None*, *delete=()*, *exclude=()*)

Return a matrix without deleted taxa and excluded characters.

bootstrap(*matrix=None*, *delete=()*, *exclude=()*)

Return a bootstrapped matrix.

add_sequence(*name*, *sequence*)

Add a sequence (string) to the matrix.

insert_gap(*pos*, *n=1*, *leftgreedy=False*)

Add a gap into the matrix and adjust charsets and partitions.

pos=0: first position pos=nchar: last position

invert(*charlist*)

Return all character indices that are not in charlist.

gaponly(*include_missing=False*)

Return gap-only sites.

terminal_gap_to_missing(*missing=None*, *skip_n=True*)

Replace all terminal gaps with missing character.

Mixtures like ???——??——- are properly resolved.

Bio.Nexus.Nodes module

Linked list functionality for use in Bio.Nexus.

Provides functionality of a linked list. Each node has one (or none) predecessor, and an arbitrary number of successors. Nodes can store arbitrary data in a NodeData class.

Subclassed by Nexus.Trees to store phylogenetic trees.

Bug reports to Frank Kauff (fkauff@biologie.uni-kl.de)

exception Bio.Nexus.Nodes.**ChainException**

Bases: Exception

Provision for the management of Chain exceptions.

exception Bio.Nexus.Nodes.**NodeException**

Bases: Exception

Provision for the management of Node exceptions.

class Bio.Nexus.Nodes.**Chain**

Bases: object

Stores a list of nodes that are linked together.

__init__() → None

Initialize a node chain.

all_ids() → list[int]

Return a list of all node ids.

add(*node: Node*, *prev: int | None = None*) → int

Attach node to another.

collapse(*id*)

Delete node from chain and relinks successors to predecessor.

kill(*id*)

Kill a node from chain without caring to what it is connected.

unlink(*id*)

Disconnect node from his predecessor.

link(*parent*, *child*)

Connect son to parent.

is_parent_of(*parent*, *grandchild*)

Check if grandchild is a subnode of parent.

trace(*start*, *finish*)

Return a list of all node_ids between two nodes (excluding start, including end).

class Bio.Nexus.Nodes.**Node**(*data=None*)

Bases: object

A single node.

__init__(*data=None*)

Represent a node with one predecessor and multiple successors.

set_id(*id*)

Set the id of a node, if not set yet.

get_id()

Return the node's id.

get_succ()

Return a list of the node's successors.

get_prev()

Return the id of the node's predecessor.

add_succ(*id*)

Add a node id to the node's successors.

remove_succ(*id*)

Remove a node id from the node's successors.

set_succ(*new_succ*)

Set the node's successors.

set_prev(*id*)

Set the node's predecessor.

get_data()

Return a node's data.

set_data(*data*)

Set a node's data.

Bio.Nexus.StandardData module

Objects to represent NEXUS standard data type matrix coding.

exception Bio.Nexus.StandardData.**NexusError**

Bases: Exception

Provision for the management of Nexus exceptions.

class Bio.Nexus.StandardData.**StandardData**(*data*)

Bases: object

Create a StandardData iterable object.

Each coding specifies t [type] => (std [standard], multi [multistate] or uncer [uncertain]) and d [data]

__init__(*data*)

Initialize the class.

__len__()

Return the length of the coding, use len(my_coding).

__getitem__(*arg*)

Pull out child by index.

__iter__()

Iterate over the items.

__next__()

Return next item.

raw()

Return the full coding as a python list.

__str__()

Return the full coding as a python string, use str(my_coding).

Bio.Nexus.Trees module

Tree class to handle phylogenetic trees.

Provides a set of methods to read and write newick-format tree descriptions, get information about trees (monphyly of taxon sets, congruence between trees, common ancestors,...) and to manipulate trees (re-root trees, split terminal nodes).

exception Bio.Nexus.Trees.**TreeError**

Bases: Exception

Provision for the management of Tree exceptions.

class Bio.Nexus.Trees.**NodeData**(*taxon=None*, *branchlength=0.0*, *support=None*, *comment=None*)

Bases: object

Store tree-relevant data associated with nodes (e.g. branches or otus).

__init__(*taxon=None*, *branchlength=0.0*, *support=None*, *comment=None*)

Initialize the class.

class Bio.Nexus.Trees.**Tree**(*tree=None*, *weight=1.0*, *rooted=False*, *name=''*, *data=NodeData*, *values_are_support=False*, *max_support=1.0*)

Bases: *Chain*

Represent a tree using a chain of nodes with on predecessor (=ancestor) and multiple successors (=subclades).

__init__(*tree=None*, *weight=1.0*, *rooted=False*, *name=''*, *data=NodeData*, *values_are_support=False*, *max_support=1.0*)

Ntree(self,tree).

node(*node_id*)

Return the instance of node_id.

node = node(self,node_id)

split(*parent_id=None*, *n=2*, *branchlength=1.0*)

Speciation: generates n (default two) descendants of a node.

[new ids] = split(self,parent_id=None,n=2,branchlength=1.0):

search_taxon(*taxon*)

Return the first matching taxon in self.data.taxon. Not restricted to terminal nodes.

node_id = search_taxon(self,taxon)

prune(*taxon*)

Prune a terminal taxon from the tree.

id_of_previous_node = prune(self,taxon) If taxon is from a bifurcation, the connectiong node will be collapsed and its branchlength added to remaining terminal node. This might be no longer a meaningful value'

get_taxa(*node_id=None*)

Return a list of all otus downwards from a node.

nodes = get_taxa(self,node_id=None)

get_terminals()

Return a list of all terminal nodes.

is_terminal(*node*)

Return True if node is a terminal node.

is_internal(*node*)

Return True if node is an internal node.

is_preterminal(*node*)

Return True if all successors of a node are terminal ones.

count_terminals(*node=None*)

Count the number of terminal nodes that are attached to a node.

collapse_genera(*space_equals_underscore=True*)

Collapse all subtrees which belong to the same genus.

(i.e share the same first word in their taxon name.)

sum_branchlength(*root=None*, *node=None*)

Add up the branchlengths from root (default self.root) to node.

sum = sum_branchlength(self,root=None,node=None)

set_subtree(*node*)

Return subtree as a set of nested sets.

sets = set_subtree(self,node)

is_identical(*tree2*)

Compare tree and tree2 for identity.

result = is_identical(self,tree2)

is_compatible(*tree2*, *threshold*, *strict=True*)

Compare branches with support>threshold for compatibility.

result = is_compatible(self,tree2,threshold)

common_ancestor(*node1*, *node2*)

Return the common ancestor that connects two nodes.

node_id = common_ancestor(self,node1,node2)

distance(*node1*, *node2*)

Add and return the sum of the branchlengths between two nodes.

dist = distance(self,node1,node2)

is_monophyletic(*taxon_list*)

Return node_id of common ancestor if taxon_list is monophyletic, -1 otherwise.

result = is_monophyletic(self,taxon_list)

is_bifurcating(*node=None*)

Return True if tree downstream of node is strictly bifurcating.

branchlength2support()

Move values stored in data.branchlength to data.support, and set branchlength to 0.0.

This is necessary when support has been stored as branchlength (e.g. paup), and has thus been read in as branchlength.

convert_absolute_support(*nrep*)

Convert absolute support (clade-count) to rel. frequencies.

Some software (e.g. PHYLIP consense) just calculate how often clades appear, instead of calculating relative frequencies.

has_support(*node=None*)

Return True if any of the nodes has data.support != None.

randomize(*ntax=None*, *taxon_list=None*, *branchlength=1.0*, *branchlength_sd=None*, *bifurcate=True*)

Generate a random tree with ntax taxa and/or taxa from taxlabels.

new_tree = randomize(self,ntax=None,taxon_list=None,branchlength=1.0,branchlength_sd=None,bifurcate=True) Trees are bifurcating by default. (Polytomies not yet supported).

display()

Quick and dirty lists of all nodes.

to_string(*support_as_branchlengths=False*, *branchlengths_only=False*, *plain=True*, *plain_newick=False*, *ladderize=None*, *ignore_comments=True*)

Return a paup compatible tree line.

__str__()

Short version of to_string(), gives plain tree.

unroot()

Define a unrooted Tree structure, using data of a rooted Tree.

root_with_outgroup(*outgroup=None*)

Define a tree's root with a reference group outgroup.

merge_with_support(*bstrees=None*, *constree=None*, *threshold=0.5*, *outgroup=None*)

Merge clade support (from consensus or list of bootstrap-trees) with phylogeny.

tree=merge_bootstrap(phylo,bs_tree=<list_of_trees>) or tree=merge_bootstrap(phylo,consree=consensus_tree with clade support)

__annotations__ = {}

Bio.Nexus.Trees.**consensus**(*trees*, *threshold=0.5*, *outgroup=None*)

Compute a majority rule consensus tree of all clades with relative frequency>=threshold from a list of trees.

Bio.Nexus.cnexus module

Bio.Nexus.cnexus.**scanfile**()

Scan file and deal with comments and quotes.

Module contents

The Bio.Nexus contains a NEXUS file parser and objects to model this data.

28.1.21 Bio.PDB package

Subpackages

Bio.PDB.mmtf package

Submodules

Bio.PDB.mmtf.DefaultParser module

Code handle loading mmtf-python into Biopython's structures.

class Bio.PDB.mmtf.DefaultParser.**StructureDecoder**

Bases: object

Class to pass the data from mmtf-python into a Biopython data structure.

__init__()

Initialize the class.

init_structure(*total_num_bonds*, *total_num_atoms*, *total_num_groups*, *total_num_chains*, *total_num_models*, *structure_id*)

Initialize the structure object.

Parameters

- **total_num_bonds** – the number of bonds in the structure
- **total_num_atoms** – the number of atoms in the structure
- **total_num_groups** – the number of groups in the structure
- **total_num_chains** – the number of chains in the structure
- **total_num_models** – the number of models in the structure
- **structure_id** – the id of the structure (e.g. PDB id)

set_atom_info(*atom_name*, *serial_number*, *alternative_location_id*, *x*, *y*, *z*, *occupancy*, *temperature_factor*, *element*, *charge*)

Create an atom object an set the information.

Parameters

- **atom_name** – the atom name, e.g. CA for this atom
- **serial_number** – the serial id of the atom (e.g. 1)
- **alternative_location_id** – the alternative location id for the atom, if present
- **x** – the x coordinate of the atom
- **y** – the y coordinate of the atom
- **z** – the z coordinate of the atom
- **occupancy** – the occupancy of the atom
- **temperature_factor** – the temperature factor of the atom
- **element** – the element of the atom, e.g. C for carbon. According to IUPAC. Calcium is Ca
- **charge** – the formal atomic charge of the atom

set_chain_info(*chain_id*, *chain_name*, *num_groups*)

Set the chain information.

Parameters

- **chain_id** – the asym chain id from mmCIF
- **chain_name** – the auth chain id from mmCIF
- **num_groups** – the number of groups this chain has

set_entity_info(*chain_indices*, *sequence*, *description*, *entity_type*)

Set the entity level information for the structure.

Parameters

- **chain_indices** – the indices of the chains for this entity
- **sequence** – the one letter code sequence for this entity
- **description** – the description for this entity
- **entity_type** – the entity type (polymer,non-polymer,water)

set_group_info(*group_name*, *group_number*, *insertion_code*, *group_type*, *atom_count*, *bond_count*, *single_letter_code*, *sequence_index*, *secondary_structure_type*)

Set the information for a group.

Parameters

- **group_name** – the name of this group, e.g. LYS
- **group_number** – the residue number of this group
- **insertion_code** – the insertion code for this group
- **group_type** – a string indicating the type of group (as found in the chemcomp dictionary. Empty string if none available.
- **atom_count** – the number of atoms in the group

- **bond_count** – the number of unique bonds in the group
- **single_letter_code** – the single letter code of the group
- **sequence_index** – the index of this group in the sequence defined by the entity
- **secondary_structure_type** – the type of secondary structure used (types are according to DSSP and number to type mappings are defined in the specification)

set_model_info(*model_id*, *chain_count*)

Set the information for a model.

Parameters

- **model_id** – the index for the model
- **chain_count** – the number of chains in the model

set_xtal_info(*space_group*, *unit_cell*)

Set the crystallographic information for the structure.

Parameters

- **space_group** – the space group name, e.g. "P 21 21 21"
- **unit_cell** – an array of length 6 with the unit cell parameters in order: a, b, c, alpha, beta, gamma

set_header_info(*r_free*, *r_work*, *resolution*, *title*, *deposition_date*, *release_date*, *experimnetal_methods*)

Set the header information.

Parameters

- **r_free** – the measured R-Free for the structure
- **r_work** – the measure R-Work for the structure
- **resolution** – the resolution of the structure
- **title** – the title of the structure
- **deposition_date** – the deposition date of the structure
- **release_date** – the release date of the structure
- **experimnetal_methods** – the list of experimental methods in the structure

set_bio_assembly_trans(*bio_assembly_index*, *input_chain_indices*, *input_transform*)

Set the Bioassembly transformation information. A single bioassembly can have multiple transforms.

Parameters

- **bio_assembly_index** – the integer index of the bioassembly
- **input_chain_indices** – the list of integer indices for the chains of this bioassembly
- **input_transform** – the list of doubles for the transform of this bioassmbly transform.

finalize_structure()

Any functions needed to cleanup the structure.

set_group_bond(*atom_index_one*, *atom_index_two*, *bond_order*)

Add bonds within a group.

Parameters

- **atom_index_one** – the integer atom index (in the group) of the first partner in the bond

- **atom_index_two** – the integer atom index (in the group) of the second partner in the bond
- **bond_order** – the integer bond order

set_inter_group_bond(*atom_index_one*, *atom_index_two*, *bond_order*)

Add bonds between groups.

Parameters

- **atom_index_one** – the integer atom index (in the structure) of the first partner in the bond
- **atom_index_two** – the integer atom index (in the structure) of the second partner in the bond
- **bond_order** – the bond order

Bio.PDB.mmtf.mmtfio module

Write a MMTF file.

class Bio.PDB.mmtf.mmtfio.**MMTFIO**

Bases: *StructureIO*

Write a Structure object as a MMTF file.

Examples

```
>>> from Bio.PDB import MMCIFParser
>>> from Bio.PDB.mmtf import MMTFIO
>>> parser = MMCIFParser()
>>> structure = parser.get_structure("1a8o", "PDB/1A8O.cif")
>>> io=MMTFIO()
>>> io.set_structure(structure)
>>> io.save("bio-pdb-mmtf-out.mmtf")
>>> import os
>>> os.remove("bio-pdb-mmtf-out.mmtf")  # tidy up
```

__init__()

Initialise.

save(*filepath*, *select=_select*)

Save the structure to a file.

Parameters

- **filepath** (*string*) – output file
- **select** (*object*) – selects which entities will be written.

Typically select is a subclass of L{Select}, it should have the following methods:

- accept_model(model)
- accept_chain(chain)
- accept_residue(residue)
- accept_atom(atom)

These methods should return 1 if the entity is to be written out, 0 otherwise.

```
__annotations__ = {}
```

Module contents

Support for loading 3D structures stored in MMTF files.

Bio.PDB.mmtf.**get_from_decoded**(*decoder*)

Return structure from decoder.

class Bio.PDB.mmtf.**MMTFParser**

Bases: object

Class to get a Biopython structure from a URL or a filename.

static get_structure_from_url(*pdb_id*)

Get a structure from a URL - given a PDB id.

Parameters
pdb_id – the input PDB id

Returns
the structure

static get_structure(*file_path*)

Get a structure from a file - given a file path.

Parameters
file_path – the input file path

Returns
the structure

Submodules

Bio.PDB.AbstractPropertyMap module

Class that maps (chain_id, residue_id) to a residue property.

class Bio.PDB.AbstractPropertyMap.**AbstractPropertyMap**(*property_dict*, *property_keys*, *property_list*)

Bases: object

Define base class, map holder of residue properties.

__init__(*property_dict*, *property_keys*, *property_list*)

Initialize the class.

__contains__(*id*)

Check if the mapping has a property for this residue.

Parameters

- **chain_id** (*char*) – chain id
- **res_id** (*char*) – residue id

Examples

This is an incomplete but illustrative example:

```
if (chain_id, res_id) in apmap:
    res, prop = apmap[(chain_id, res_id)]
```

__getitem__(*key*)

Return property for a residue.

Parameters

- **chain_id** (*char*) – chain id
- **res_id** (*int or (char, int, char)*) – residue id

Returns
some residue property

Return type
anything (can be a tuple)

__len__()

Return number of residues for which the property is available.

Returns
number of residues

Return type
int

keys()

Return the list of residues.

Returns
list of residues for which the property was calculated

Return type
[(chain_id, res_id), (chain_id, res_id),...]

__iter__()

Iterate over the (entity, property) list.

Handy alternative to the dictionary-like access.

Returns
iterator

Examples

```
>>> entity_property_list = [
...     ('entity_1', 'property_1'),
...     ('entity_2', 'property_2')
... ]
>>> map = AbstractPropertyMap({}, [], entity_property_list)
>>> for (res, property) in iter(map):
...     print(res, property)
entity_1 property_1
entity_2 property_2
```

class Bio.PDB.AbstractPropertyMap.**AbstractResiduePropertyMap**(*property_dict*, *property_keys*, *property_list*)

Bases: *AbstractPropertyMap*

Define class for residue properties map.

__init__(*property_dict*, *property_keys*, *property_list*)

Initialize the class.

__annotations__ = {}

class Bio.PDB.AbstractPropertyMap.**AbstractAtomPropertyMap**(*property_dict*, *property_keys*, *property_list*)

Bases: *AbstractPropertyMap*

Define class for atom properties map.

__init__(*property_dict*, *property_keys*, *property_list*)

Initialize the class.

__annotations__ = {}

Bio.PDB.Atom module

Atom class, used in Structure objects.

class Bio.PDB.Atom.**Atom**(*name: str*, *coord: ndarray*, *bfactor: float | None*, *occupancy: float | None*, *altloc: str*, *fullname: str*, *serial_number*, *element: str | None = None*, *pqr_charge: float | None = None*, *radius: float | None = None*)

Bases: `object`

Define Atom class.

The Atom object stores atom name (both with and without spaces), coordinates, B factor, occupancy, alternative location specifier and (optionally) anisotropic B factor and standard deviations of B factor and positions.

In the case of PQR files, B factor and occupancy are replaced by atomic charge and radius.

__init__(*name: str*, *coord: ndarray*, *bfactor: float | None*, *occupancy: float | None*, *altloc: str*, *fullname: str*, *serial_number*, *element: str | None = None*, *pqr_charge: float | None = None*, *radius: float | None = None*)

Initialize Atom object.

Parameters

- **name** (*string*) – atom name (eg. "CA"). Note that spaces are normally stripped.
- **coord** (*NumPy array (Float0, length 3)*) – atomic coordinates (x,y,z)
- **bfactor** (*number*) – isotropic B factor
- **occupancy** (*number*) – occupancy (0.0-1.0)
- **altloc** (*string*) – alternative location specifier for disordered atoms
- **fullname** (*string*) – full atom name, including spaces, e.g. " CA ". Normally these spaces are stripped from the atom name.
- **element** (*uppercase string (or None if unknown)*) – atom element, e.g. "C" for Carbon, "HG" for mercury,

- **pqr_charge** (*number*) – atom charge
- **radius** (*number*) – atom radius

__eq__(*other*)

Test equality.

__ne__(*other*)

Test inequality.

__gt__(*other*)

Test greater than.

__ge__(*other*)

Test greater or equal.

__lt__(*other*)

Test less than.

__le__(*other*)

Test less or equal.

__hash__()

Return atom full identifier.

__repr__()

Print Atom object as <Atom atom_name>.

__sub__(*other*)

Calculate distance between two atoms.

Parameters
other (*L{Atom}*) – the other atom

Examples

This is an incomplete but illustrative example:

```
distance = atom1 - atom2
```

strictly_equals(*other: _AtomT*, *compare_coordinates: bool = False*) → bool

Compare this atom to the other atom using a strict definition of equality.

Indicates whether the atoms have the same name, B factor, occupancy, alternate location indicator (altloc), fullname, element, charge, and radius. If `compare_coordinates` is true, then the coordinates are also compared.

Parameters

- **other** (Atom) – The atom to compare this atom with
- **compare_coordinates** (*bool*) – Whether to compare the coordinates of the atoms

Returns
Whether the atoms are strictly equal

Return type
bool

set_serial_number(*n*)

Set serial number.

set_bfactor(*bfactor*)

Set isotroptic B factor.

set_coord(*coord*)

Set coordinates.

set_altloc(*altloc*)

Set alternative location specifier.

set_occupancy(*occupancy*)

Set occupancy.

set_sigatm(*sigatm_array*)

Set standard deviation of atomic parameters.

The standard deviation of atomic parameters consists of 3 positional, 1 B factor and 1 occupancy standard deviation.

Parameters
sigatm_array (*NumPy array (length 5)*) – standard deviations of atomic parameters.

set_siguij(*siguij_array*)

Set standard deviations of anisotropic temperature factors.

Parameters
siguij_array (*NumPy array (length 6)*) – standard deviations of anisotropic temperature factors.

set_anisou(*anisou_array*)

Set anisotropic B factor.

Parameters
anisou_array (*NumPy array (length 6)*) – anisotropic B factor.

set_charge(*pqr_charge*)

Set charge.

set_radius(*radius*)

Set radius.

flag_disorder()

Set the disordered flag to 1.

The disordered flag indicates whether the atom is disordered or not.

is_disordered()

Return the disordered flag (1 if disordered, 0 otherwise).

set_parent(*parent*)

Set the parent residue.

Arguments:

- parent - Residue object

detach_parent()

Remove reference to parent.

get_sigatm()

Return standard deviation of atomic parameters.

get_siguij()

Return standard deviations of anisotropic temperature factors.

get_anisou()

Return anisotropic B factor.

get_parent()

Return parent residue.

get_serial_number()

Return the serial number.

get_name()

Return atom name.

get_id()

Return the id of the atom (which is its atom name).

get_full_id()

Return the full id of the atom.

The full id of an atom is a tuple used to uniquely identify the atom and consists of the following elements: (structure id, model id, chain id, residue id, atom name, altloc)

get_coord()

Return atomic coordinates.

get_bfactor()

Return B factor.

get_occupancy()

Return occupancy.

get_fullname()

Return the atom name, including leading and trailing spaces.

get_altloc()

Return alternative location specifier.

get_level()

Return level.

get_charge()

Return charge.

get_radius()

Return radius.

transform(*rot*, *tran*)

Apply rotation and translation to the atomic coordinates.

Parameters

- **rot** (*3x3 NumPy array*) – A right multiplying rotation matrix
- **tran** (*size 3 NumPy array*) – the translation vector

Examples

This is an incomplete but illustrative example:

```
from numpy import pi, array
from Bio.PDB.vectors import Vector, rotmat
rotation = rotmat(pi, Vector(1, 0, 0))
translation = array((0, 0, 1), 'f')
atom.transform(rotation, translation)
```

get_vector()

Return coordinates as Vector.

Returns
coordinates as 3D vector

Return type
Bio.PDB.Vector class

copy()

Create a copy of the Atom.

Parent information is lost.

class Bio.PDB.Atom.**DisorderedAtom**(*id*)

Bases: *DisorderedEntityWrapper*

Contains all Atom objects that represent the same disordered atom.

One of these atoms is "selected" and all method calls not caught by DisorderedAtom are forwarded to the selected Atom object. In that way, a DisorderedAtom behaves exactly like a normal Atom. By default, the selected Atom object represents the Atom object with the highest occupancy, but a different Atom object can be selected by using the disordered_select(altloc) method.

__init__(*id*)

Create DisorderedAtom.

Arguments:

- id - string, atom name

__iter__()

Iterate through disordered atoms.

__repr__()

Return disordered atom identifier.

center_of_mass()

Return the center of mass of the DisorderedAtom as a numpy array.

Assumes all child atoms have the same mass (same element).

disordered_get_list()

Return list of atom instances.

Sorts children by altloc (empty, then alphabetical).

disordered_add(*atom*)

Add a disordered atom.

disordered_remove(*altloc*)

Remove a child atom altloc from the DisorderedAtom.

Arguments:

- altloc - name of the altloc to remove, as a string.

transform(*rot*, *tran*)

Apply rotation and translation to all children.

See the documentation of Atom.transform for details.

Bio.PDB.Chain module

Chain class, used in Structure objects.

class Bio.PDB.Chain.**Chain**(*id*)

Bases: *Entity*[Model, Residue]

Define Chain class.

Chain is an object of type Entity, stores residues and includes a method to access atoms from residues.

__init__(*id*)

Initialize the class.

__gt__(*other*)

Validate if id is greater than other.id.

__ge__(*other*)

Validate if id is greater or equal than other.id.

__lt__(*other*)

Validate if id is less than other.id.

__le__(*other*)

Validate if id is less or equal than other id.

__getitem__(*id*)

Return the residue with given id.

The id of a residue is (hetero flag, sequence identifier, insertion code). If id is an int, it is translated to (" ", id, " ") by the _translate_id method.

Arguments:

- id - (string, int, string) or int

__contains__(*id*)

Check if a residue with given id is present in this chain.

Arguments:

- id - (string, int, string) or int

__delitem__(*id*)

Delete item.

Arguments:

- id - (string, int, string) or int

__repr__()

Return the chain identifier.

get_unpacked_list()

Return a list of undisordered residues.

Some Residue objects hide several disordered residues (DisorderedResidue objects). This method unpacks them, ie. it returns a list of simple Residue objects.

has_id(*id*)

Return 1 if a residue with given id is present.

The id of a residue is (hetero flag, sequence identifier, insertion code).

If id is an int, it is translated to (" ", id, " ") by the _translate_id method.

Arguments:

- id - (string, int, string) or int

get_residues()

Return residues.

get_atoms()

Return atoms from residues.

atom_to_internal_coordinates(*verbose: bool = False*) → None

Create/update internal coordinates from Atom X,Y,Z coordinates.

Internal coordinates are bond length, angle and dihedral angles.

Parameters
bool (*verbose*) – default False describe runtime problems

internal_to_atom_coordinates(*verbose: bool = False*, *start: int | None = None*, *fin: int | None = None*)

Create/update atom coordinates from internal coordinates.

Parameters
bool (*verbose*) – default False describe runtime problems

Param
start, fin integers optional sequence positions for begin, end of subregion to process. N.B. this activates serial residue assembly, <start> residue CA will be at origin

Raises
Exception – if any chain does not have .internal_coord attribute

__annotations__ = {}

__orig_bases__ = (Bio.PDB.Entity.Entity[ForwardRef('Model'), ForwardRef('Residue')],)

__parameters__ = ()

Bio.PDB.DSSP module

Use the DSSP program to calculate secondary structure and accessibility.

You need to have a working version of DSSP (and a license, free for academic use) in order to use this. For DSSP, see https://swift.cmbi.umcn.nl/gv/dssp/.

The following Accessible surface area (ASA) values can be used, defaulting to the Sander and Rost values:

Ahmad
Ahmad et al. 2003 https://doi.org/10.1002/prot.10328

Miller
Miller et al. 1987 https://doi.org/10.1016/0022-2836(87)90038-6

Sander
Sander and Rost 1994 https://doi.org/10.1002/prot.340200303

Wilke
Tien et al. 2013 https://doi.org/10.1371/journal.pone.0080635

The DSSP codes for secondary structure used here are:

Code	Structure
H	Alpha helix (4-12)
B	Isolated beta-bridge residue
E	Strand
G	3-10 helix
I	Pi helix
T	Turn
S	Bend
-	None

Usage

The DSSP class can be used to run DSSP on a PDB or mmCIF file, and provides a handle to the DSSP secondary structure and accessibility.

Note that DSSP can only handle one model, and will only run calculations on the first model in the provided PDB file.

Examples

Typical use:

```
from Bio.PDB import PDBParser
from Bio.PDB.DSSP import DSSP
p = PDBParser()
structure = p.get_structure("1MOT", "/local-pdb/1mot.pdb")
model = structure[0]
dssp = DSSP(model, "/local-pdb/1mot.pdb")
```

Note that the recent DSSP executable from the DSSP-2 package was renamed from `dssp` to `mkdssp`. If using a recent DSSP release, you may need to provide the name of your DSSP executable:

```
dssp = DSSP(model, '/local-pdb/1mot.pdb', dssp='mkdssp')
```

DSSP data is accessed by a tuple - (chain id, residue id):

```
a_key = list(dssp.keys())[2]
dssp[a_key]
```

The dssp data returned for a single residue is a tuple in the form:

Tuple Index	Value
0	DSSP index
1	Amino acid
2	Secondary structure
3	Relative ASA
4	Phi
5	Psi
6	NH–>O_1_relidx
7	NH–>O_1_energy
8	O–>NH_1_relidx
9	O–>NH_1_energy
10	NH–>O_2_relidx
11	NH–>O_2_energy
12	O–>NH_2_relidx
13	O–>NH_2_energy

Bio.PDB.DSSP.**version**(*version_string*)

Parse semantic version scheme for easy comparison.

Bio.PDB.DSSP.**ss_to_index**(*ss*)

Secondary structure symbol to index.

H=0 E=1 C=2

Bio.PDB.DSSP.**dssp_dict_from_pdb_file**(*in_file*, *DSSP='dssp'*, *dssp_version='3.9.9'*)

Create a DSSP dictionary from a PDB file.

Parameters

in_file
[string] pdb file

DSSP
[string] DSSP executable (argument to subprocess)

dssp_version
[string] Version of DSSP executable

Returns

(out_dict, keys)
[tuple] a dictionary that maps (chainid, resid) to amino acid type, secondary structure code and accessibility.

Examples

How dssp_dict_from_pdb_file could be used:

```
from Bio.PDB.DSSP import dssp_dict_from_pdb_file
dssp_tuple = dssp_dict_from_pdb_file("/local-pdb/1fat.pdb")
dssp_dict = dssp_tuple[0]
print(dssp_dict['A',(' ', 1, ' ')])
```

Bio.PDB.DSSP.**make_dssp_dict**(*filename*)

DSSP dictionary mapping identifiers to properties.

Return a DSSP dictionary that maps (chainid, resid) to aa, ss and accessibility, from a DSSP file.

Parameters

filename
[string] the DSSP output file

class Bio.PDB.DSSP.**DSSP**(*model*, *in_file*, *dssp='dssp'*, *acc_array='Sander'*, *file_type=''*)

Bases: *AbstractResiduePropertyMap*

Run DSSP and parse secondary structure and accessibility.

Run DSSP on a PDB/mmCIF file, and provide a handle to the DSSP secondary structure and accessibility.

Note that DSSP can only handle one model.

Examples

How DSSP could be used:

```
from Bio.PDB import PDBParser
from Bio.PDB.DSSP import DSSP
p = PDBParser()
structure = p.get_structure("1MOT", "/local-pdb/1mot.pdb")
model = structure[0]
dssp = DSSP(model, "/local-pdb/1mot.pdb")
# DSSP data is accessed by a tuple (chain_id, res_id)
a_key = list(dssp.keys())[2]
# (dssp index, amino acid, secondary structure, relative ASA, phi, psi,
# NH_O_1_relidx, NH_O_1_energy, O_NH_1_relidx, O_NH_1_energy,
# NH_O_2_relidx, NH_O_2_energy, O_NH_2_relidx, O_NH_2_energy)
dssp[a_key]
```

__init__(*model*, *in_file*, *dssp='dssp'*, *acc_array='Sander'*, *file_type=''*)

Create a DSSP object.

Parameters

model
[Model] The first model of the structure

in_file
[string] Either a PDB file or a DSSP file.

dssp
[string] The dssp executable (ie. the argument to subprocess)

acc_array
[string] Accessible surface area (ASA) from either Miller et al. (1987), Sander & Rost (1994), Wilke: Tien et al. 2013, or Ahmad et al. (2003) as string Sander/Wilke/Miller/Ahmad. Defaults to Sander.

file_type: string
File type switch: either PDB, MMCIF or DSSP. Inferred from the file extension by default.

`__annotations__ = {}`

Bio.PDB.Dice module

Code for chopping up (dicing) a structure.

This module is used internally by the Bio.PDB.extract() function.

`class Bio.PDB.Dice.ChainSelector(chain_id, start, end, model_id=0)`

Bases: `object`

Only accepts residues with right chainid, between start and end.

Remove hydrogens, waters and ligands. Only use model 0 by default.

`__init__(chain_id, start, end, model_id=0)`

Initialize the class.

`accept_model(model)`

Verify if model match the model identifier.

`accept_chain(chain)`

Verify if chain match chain identifier.

`accept_residue(residue)`

Verify if a residue sequence is between the start and end sequence.

`accept_atom(atom)`

Verify if atoms are not Hydrogen.

`Bio.PDB.Dice.extract(structure, chain_id, start, end, filename)`

Write out selected portion to filename.

Bio.PDB.Entity module

Base class for Residue, Chain, Model and Structure classes.

It is a simple container class, with list and dictionary like properties.

`class Bio.PDB.Entity.Entity(id)`

Bases: `Generic[_Parent, _Child]`

Basic container object for PDB hierarchy.

Structure, Model, Chain and Residue are subclasses of Entity. It deals with storage and lookup.

`level: str`

`__init__(id)`

Initialize the class.

parent: _Parent | None

child_list: list[_Child]

child_dict: dict[Any, _Child]

__len__()

Return the number of children.

__getitem__(*id*)

Return the child with given id.

__delitem__(*id*)

Remove a child.

__contains__(*id*)

Check if there is a child element with the given id.

__iter__()

Iterate over children.

__eq__(*other*)

Test for equality. This compares full_id including the IDs of all parents.

__ne__(*other*)

Test for inequality.

__gt__(*other*)

Test greater than.

__ge__(*other*)

Test greater or equal.

__lt__(*other*)

Test less than.

__le__(*other*)

Test less or equal.

__hash__()

Hash method to allow uniqueness (set).

property id

Return identifier.

strictly_equals(*other: _Self*, *compare_coordinates: bool = False*) → bool

Compare this entity to the other entity for equality.

Recursively compare the children of this entity to the other entity's children. Compare most properties including names and IDs.

Parameters

- **other** (Entity) – The entity to compare this entity with
- **compare_coordinates** (*bool*) – Whether to compare atomic coordinates

Returns

Whether the two entities are strictly equal

Return type
bool

get_level()

Return level in hierarchy.

A - atom R - residue C - chain M - model S - structure

set_parent(*entity: _Parent*)

Set the parent Entity object.

detach_parent()

Detach the parent.

detach_child(*id*)

Remove a child.

add(*entity: _Child*)

Add a child to the Entity.

insert(*pos: int*, *entity: _Child*)

Add a child to the Entity at a specified position.

get_iterator()

Return iterator over children.

get_list()

Return a copy of the list of children.

has_id(*id*)

Check if a child with given id exists.

get_parent()

Return the parent Entity object.

get_id()

Return the id.

get_full_id()

Return the full id.

The full id is a tuple containing all id's starting from the top object (Structure) down to the current object. A full id for a Residue object e.g. is something like:

("1abc", 0, "A", (" ", 10, "A"))

This corresponds to:

Structure with id "1abc" Model with id 0 Chain with id "A" Residue with id (" ", 10, "A")

The Residue id indicates that the residue is not a hetero-residue (or a water) because it has a blank hetero field, that its sequence identifier is 10 and its insertion code "A".

transform(*rot*, *tran*)

Apply rotation and translation to the atomic coordinates.

Parameters

- **rot** (*3x3 NumPy array*) – A right multiplying rotation matrix
- **tran** (*size 3 NumPy array*) – the translation vector

Examples

This is an incomplete but illustrative example:

```
from numpy import pi, array
from Bio.PDB.vectors import Vector, rotmat
rotation = rotmat(pi, Vector(1, 0, 0))
translation = array((0, 0, 1), 'f')
entity.transform(rotation, translation)
```

center_of_mass(*geometric=False*)

Return the center of mass of the Entity as a numpy array.

If geometric is True, returns the center of geometry instead.

copy()

Copy entity recursively.

__annotations__ = {'child_dict': dict[typing.Any, ~_Child], 'child_list': list[~_Child], 'level': <class 'str'>, 'parent': typing.Optional[~_Parent]}

__orig_bases__ = (typing.Generic[~_Parent, ~_Child],)

__parameters__ = (~_Parent, ~_Child)

class Bio.PDB.Entity.**DisorderedEntityWrapper**(*id*)

Bases: object

Wrapper class to group equivalent Entities.

This class is a simple wrapper class that groups a number of equivalent Entities and forwards all method calls to one of them (the currently selected object). DisorderedResidue and DisorderedAtom are subclasses of this class.

E.g.: A DisorderedAtom object contains a number of Atom objects, where each Atom object represents a specific position of a disordered atom in the structure.

__init__(*id*)

Initialize the class.

__getattr__(*method*)

Forward the method call to the selected child.

__getitem__(*id*)

Return the child with the given id.

__setitem__(*id*, *child*)

Add a child, associated with a certain id.

__contains__(*id*)

Check if the child has the given id.

__iter__()

Return the number of children.

__len__()

Return the number of children.

__sub__(*other*)

Subtraction with another object.

__gt__(*other*)
Return if child is greater than other.

__ge__(*other*)
Return if child is greater or equal than other.

__lt__(*other*)
Return if child is less than other.

__le__(*other*)
Return if child is less or equal than other.

copy()
Copy disorderd entity recursively.

get_id()
Return the id.

strictly_equals(*other:* DisorderedEntityWrapper, *compare_coordinates: bool = False*) → bool
Compare this entity to the other entity using a strict definition of equality.

Recursively compare the children of this entity to the other entity's children. Compare most properties including the selected child, names, and IDs.

Parameters

- **other** (DisorderedEntityWrapper) – The entity to compare this entity with
- **compare_coordinates** (*bool*) – Whether to compare atomic coordinates

Returns
Whether the two entities are strictly equal

Return type
bool

disordered_has_id(*id*)
Check if there is an object present associated with this id.

detach_parent()
Detach the parent.

get_parent()
Return parent.

set_parent(*parent*)
Set the parent for the object and its children.

disordered_select(*id*)
Select the object with given id as the currently active object.

Uncaught method calls are forwarded to the selected child object.

disordered_add(*child*)
Add disordered entry.

This is implemented by DisorderedAtom and DisorderedResidue.

disordered_remove(*child*)
Remove disordered entry.

This is implemented by DisorderedAtom and DisorderedResidue.

__annotations__ = {}

is_disordered()

Return 2, indicating that this Entity is a collection of Entities.

disordered_get_id_list()

Return a list of id's.

disordered_get(*id=None*)

Get the child object associated with id.

If id is None, the currently selected child is returned.

disordered_get_list()

Return list of children.

Bio.PDB.FragmentMapper module

Classify protein backbone structure with Kolodny et al's fragment libraries.

It can be regarded as a form of objective secondary structure classification. Only fragments of length 5 or 7 are supported (ie. there is a 'central' residue).

Full reference:

Kolodny R, Koehl P, Guibas L, Levitt M. Small libraries of protein fragments model native protein structures accurately. J Mol Biol. 2002 323(2):297-307.

The definition files of the fragments can be obtained from:

http://github.com/csblab/fragments/

You need these files to use this module.

The following example uses the library with 10 fragments of length 5. The library files can be found in directory 'fragment_data'.

```
>>> from Bio.PDB.PDBParser import PDBParser
>>> from Bio.PDB.FragmentMapper import FragmentMapper
>>> parser = PDBParser()
>>> structure = parser.get_structure("1a8o", "PDB/1A8O.pdb")
>>> model = structure[0]
>>> fm = FragmentMapper(model, lsize=10, flength=5, fdir="PDB")
>>> chain = model['A']
>>> res152 = chain[152]
>>> res157 = chain[157]
>>> res152 in fm # is res152 mapped? (fragment of a C-alpha polypeptide)
False
>>> res157 in fm # is res157 mapped? (fragment of a C-alpha polypeptide)
True
```

class Bio.PDB.FragmentMapper.**Fragment**(*length*, *fid*)

Bases: object

Represent a polypeptide C-alpha fragment.

__init__(*length*, *fid*)

Initialize fragment object.

Parameters

- **length** (*int*) – length of the fragment
- **fid** (*int*) – id for the fragment

get_resname_list()

Get residue list.

Returns
the residue names

Return type
[string, string,...]

get_id()

Get identifier for the fragment.

Returns
id for the fragment

Return type
int

get_coords()

Get the CA coordinates in the fragment.

Returns
the CA coords in the fragment

Return type
NumPy (Nx3) array

add_residue(*resname*, *ca_coord*)

Add a residue.

Parameters

- **resname** (*string*) – residue name (eg. GLY).
- **ca_coord** (*NumPy array with length 3*) – the c-alpha coordinates of the residues

__len__()

Return length of the fragment.

__sub__(*other*)

Return rmsd between two fragments.

Returns
rmsd between fragments

Return type
float

Examples

This is an incomplete but illustrative example:

```
rmsd = fragment1 - fragment2
```

__repr__()

Represent the fragment object as a string.

Returns <Fragment length=L id=ID> where L=length of fragment and ID the identifier (rank in the library).

class Bio.PDB.FragmentMapper.**FragmentMapper**(*model*, *lsize=20*, *flength=5*, *fdir='.'*)

Bases: object

Map polypeptides in a model to lists of representative fragments.

__init__(*model*, *lsize=20*, *flength=5*, *fdir='.'*)

Create instance of FragmentMapper.

Parameters

- **model** (*L{Model}*) – the model that will be mapped
- **lsize** (*int*) – number of fragments in the library
- **flength** (*int*) – length of fragments in the library
- **fdir** (*string*) – directory where the definition files are found (default=".")

__contains__(*res*)

Check if the given residue is in any of the mapped fragments.

__getitem__(*res*)

Get an entry.

Returns
fragment classification

Return type
L{Fragment}

Bio.PDB.HSExposure module

Half-sphere exposure and coordination number calculation.

class Bio.PDB.HSExposure.**HSExposureCA**(*model*, *radius=12*, *offset=0*)

Bases: _AbstractHSExposure

Class to calculate HSE based on the approximate CA-CB vectors.

Uses three consecutive CA positions.

__init__(*model*, *radius=12*, *offset=0*)

Initialize class.

Parameters

- **model** (*L{Model}*) – the model that contains the residues
- **radius** (*float*) – radius of the sphere (centred at the CA atom)

- **offset** (*int*) – number of flanking residues that are ignored in the calculation of the number of neighbors

pcb_vectors_pymol(*filename='hs_exp.py'*)

Write PyMol script for visualization.

Write a PyMol script that visualizes the pseudo CB-CA directions at the CA coordinates.

Parameters
filename (*string*) – the name of the pymol script file

__annotations__ = {}

class Bio.PDB.HSExposure.**HSExposureCB**(*model*, *radius=12*, *offset=0*)

Bases: _AbstractHSExposure

Class to calculate HSE based on the real CA-CB vectors.

__init__(*model*, *radius=12*, *offset=0*)

Initialize class.

Parameters

- **model** (*L{Model}*) – the model that contains the residues
- **radius** (*float*) – radius of the sphere (centred at the CA atom)
- **offset** (*int*) – number of flanking residues that are ignored in the calculation of the number of neighbors

__annotations__ = {}

class Bio.PDB.HSExposure.**ExposureCN**(*model*, *radius=12.0*, *offset=0*)

Bases: *AbstractPropertyMap*

Residue exposure as number of CA atoms around its CA atom.

__init__(*model*, *radius=12.0*, *offset=0*)

Initialize class.

A residue's exposure is defined as the number of CA atoms around that residue's CA atom. A dictionary is returned that uses a L{Residue} object as key, and the residue exposure as corresponding value.

Parameters

- **model** (*L{Model}*) – the model that contains the residues
- **radius** (*float*) – radius of the sphere (centred at the CA atom)
- **offset** (*int*) – number of flanking residues that are ignored in the calculation of the number of neighbors

__annotations__ = {}

Bio.PDB.MMCIF2Dict module

Turn an mmCIF file into a dictionary.

class Bio.PDB.MMCIF2Dict.**MMCIF2Dict**(*filename*)

Bases: dict

Parse a mmCIF file and return a dictionary.

__init__(*filename*)

Parse a mmCIF file and return a dictionary.

Arguments:

- file - name of the PDB file OR an open filehandle

Bio.PDB.MMCIFParser module

mmCIF parsers.

class Bio.PDB.MMCIFParser.**MMCIFParser**(*structure_builder=None*, *auth_chains=True*, *auth_residues=True*, *QUIET=False*)

Bases: object

Parse a mmCIF file and return a Structure object.

__init__(*structure_builder=None*, *auth_chains=True*, *auth_residues=True*, *QUIET=False*)

Create a PDBParser object.

The mmCIF parser calls a number of standard methods in an aggregated StructureBuilder object. Normally this object is instantiated by the MMCIParser object itself, but if the user provides his/her own StructureBuilder object, the latter is used instead.

Arguments:

- structure_builder - an optional user implemented StructureBuilder class.
- auth_chains - True by default. If true, use the author chain IDs. If false, use the re-assigned mmCIF chain IDs.
- auth_residues - True by default. If true, use the author residue numbering. If false, use the mmCIF "label" residue numbering, which has no insertion codes, and strictly increments residue numbers. NOTE: Non-polymers such as water don't have a "label" residue number, and will be skipped.
- QUIET - Evaluated as a Boolean. If true, warnings issued in constructing the SMCRA data will be suppressed. If false (DEFAULT), they will be shown. These warnings might be indicative of problems in the mmCIF file!

get_structure(*structure_id*, *filename*)

Return the structure.

Arguments:

- structure_id - string, the id that will be used for the structure
- filename - name of mmCIF file, OR an open text mode file handle

class Bio.PDB.MMCIFParser.**FastMMCIFParser**(*structure_builder=None*, *auth_chains=True*, *auth_residues=True*, *QUIET=False*)

Bases: object

Parse an MMCIF file and return a Structure object.

__init__(*structure_builder=None*, *auth_chains=True*, *auth_residues=True*, *QUIET=False*)

Create a FastMMCIFParser object.

The mmCIF parser calls a number of standard methods in an aggregated StructureBuilder object. Normally this object is instantiated by the parser object itself, but if the user provides his/her own StructureBuilder object, the latter is used instead.

The main difference between this class and the regular MMCIFParser is that only 'ATOM' and 'HETATM' lines are parsed here. Use if you are interested only in coordinate information.

Arguments:

- structure_builder - an optional user implemented StructureBuilder class.
- auth_chains - True by default. If true, use the author chain IDs. If false, use the re-assigned mmCIF chain IDs.
- auth_residues - True by default. If true, use the author residue numbering. If false, use the mmCIF "label" residue numbering, which has no insertion codes, and strictly increments residue numbers. NOTE: Non-polymers such as water don't have a "label" residue number, and will be skipped.
- QUIET - Evaluated as a Boolean. If true, warnings issued in constructing the SMCRA data will be suppressed. If false (DEFAULT), they will be shown. These warnings might be indicative of problems in the mmCIF file!

get_structure(*structure_id*, *filename*)

Return the structure.

Arguments:

- structure_id - string, the id that will be used for the structure
- filename - name of the mmCIF file OR an open filehandle

Bio.PDB.Model module

Model class, used in Structure objects.

class Bio.PDB.Model.**Model**(*id*, *serial_num=None*)

Bases: *Entity*[Structure, Chain]

The object representing a model in a structure.

In a structure derived from an X-ray crystallography experiment, only a single model will be present (with some exceptions). NMR structures normally contain many different models.

__init__(*id*, *serial_num=None*)

Initialize.

Arguments:

- id - int
- serial_num - int

__repr__()

Return model identifier.

get_chains()

Return chains.

get_residues()

Return residues.

get_atoms()

Return atoms.

atom_to_internal_coordinates(*verbose: bool = False*) → None

Create/update internal coordinates from Atom X,Y,Z coordinates.

Internal coordinates are bond length, angle and dihedral angles.

Parameters
bool (*verbose*) – default False describe runtime problems

__annotations__ = {}

__orig_bases__ = (Bio.PDB.Entity.Entity[ForwardRef('Structure'), ForwardRef('Chain')],)

__parameters__ = ()

internal_to_atom_coordinates(*verbose: bool = False*) → None

Create/update atom coordinates from internal coordinates.

Parameters
bool (*verbose*) – default False describe runtime problems

Raises
Exception – if any chain does not have .pic attribute

Bio.PDB.NACCESS module

Interface for the program NACCESS.

See: http://wolf.bms.umist.ac.uk/naccess/ Atomic Solvent Accessible Area Calculations

errors likely to occur with the binary: default values are often due to low default settings in accall.pars - e.g. max cubes error: change in accall.pars and recompile binary

use naccess -y, naccess -h or naccess -w to include HETATM records

Bio.PDB.NACCESS.**run_naccess**(*model*, *pdb_file*, *probe_size=None*, *z_slice=None*, *naccess='naccess'*, *temp_path='/tmp/'*)

Run naccess for a pdb file.

Bio.PDB.NACCESS.**process_rsa_data**(*rsa_data*)

Process the .rsa output file: residue level SASA data.

Bio.PDB.NACCESS.**process_asa_data**(*rsa_data*)

Process the .asa output file: atomic level SASA data.

class Bio.PDB.NACCESS.**NACCESS**(*model*, *pdb_file=None*, *naccess_binary='naccess'*, *tmp_directory='/tmp'*)

Bases: *AbstractResiduePropertyMap*

Define NACCESS class for residue properties map.

__init__(*model*, *pdb_file=None*, *naccess_binary='naccess'*, *tmp_directory='/tmp'*)

Initialize the class.

__annotations__ = {}

class Bio.PDB.NACCESS.**NACCESS_atomic**(*model*, *pdb_file=None*, *naccess_binary='naccess'*, *tmp_directory='/tmp'*)

Bases: *AbstractAtomPropertyMap*

Define NACCESS atomic class for atom properties map.

__init__(*model*, *pdb_file=None*, *naccess_binary='naccess'*, *tmp_directory='/tmp'*)

Initialize the class.

__annotations__ = {}

Bio.PDB.NeighborSearch module

Fast atom neighbor lookup using a KD tree (implemented in C).

class Bio.PDB.NeighborSearch.**NeighborSearch**(*atom_list*, *bucket_size=10*)

Bases: `object`

Class for neighbor searching.

This class can be used for two related purposes:

1. To find all atoms/residues/chains/models/structures within radius of a given query position.
2. To find all atoms/residues/chains/models/structures that are within a fixed radius of each other.

NeighborSearch makes use of the KDTree class implemented in C for speed.

__init__(*atom_list*, *bucket_size=10*)

Create the object.

Arguments:

- atom_list - list of atoms. This list is used in the queries. It can contain atoms from different structures.
- bucket_size - bucket size of KD tree. You can play around with this to optimize speed if you feel like it.

search(*center*, *radius*, *level='A'*)

Neighbor search.

Return all atoms/residues/chains/models/structures that have at least one atom within radius of center. What entity level is returned (e.g. atoms or residues) is determined by level (A=atoms, R=residues, C=chains, M=models, S=structures).

Arguments:

- center - NumPy array
- radius - float

- level - char (A, R, C, M, S)

search_all(*radius*, *level='A'*)

All neighbor search.

Search all entities that have atoms pairs within radius.

Arguments:

- radius - float
- level - char (A, R, C, M, S)

Bio.PDB.PDBExceptions module

Some Bio.PDB-specific exceptions.

exception Bio.PDB.PDBExceptions.**PDBException**

Bases: Exception

Define class PDBException.

exception Bio.PDB.PDBExceptions.**PDBConstructionException**

Bases: Exception

Define class PDBConstructionException.

exception Bio.PDB.PDBExceptions.**PDBConstructionWarning**

Bases: *BiopythonWarning*

Define class PDBConstructionWarning.

__annotations__ = {}

exception Bio.PDB.PDBExceptions.**PDBIOException**

Bases: Exception

Define class PDBIOException.

Bio.PDB.PDBIO module

Output of PDB files.

class Bio.PDB.PDBIO.**Select**

Bases: object

Select everything for PDB output (for use as a base class).

Default selection (everything) during writing - can be used as base class to implement selective output. This selects which entities will be written out.

__repr__()

Represent the output as a string for debugging.

accept_model(*model*)

Overload this to reject models for output.

accept_chain(*chain*)

Overload this to reject chains for output.

accept_residue(*residue*)

Overload this to reject residues for output.

accept_atom(*atom*)

Overload this to reject atoms for output.

class Bio.PDB.PDBIO.**StructureIO**

Bases: `object`

Base class to derive structure file format writers from.

__init__()

Initialise.

set_structure(*pdb_object*)

Check what the user is providing and build a structure.

class Bio.PDB.PDBIO.**PDBIO**(*use_model_flag=0*, *is_pqr=False*)

Bases: *StructureIO*

Write a Structure object (or a subset of a Structure object) as a PDB or PQR file.

Examples

```
>>> from Bio.PDB import PDBParser
>>> from Bio.PDB.PDBIO import PDBIO
>>> parser = PDBParser()
>>> structure = parser.get_structure("1a8o", "PDB/1A8O.pdb")
>>> io=PDBIO()
>>> io.set_structure(structure)
>>> io.save("bio-pdb-pdbio-out.pdb")
>>> import os
>>> os.remove("bio-pdb-pdbio-out.pdb")  # tidy up
```

__init__(*use_model_flag=0*, *is_pqr=False*)

Create the PDBIO object.

Parameters

- **use_model_flag** (*int*) – if 1, force use of the MODEL record in output.
- **is_pqr** (*Boolean*) – if True, build PQR file. Otherwise build PDB file.

save(*file*, *select=_select*, *write_end=True*, *preserve_atom_numbering=False*)

Save structure to a file.

Parameters

- **file** (*string or filehandle*) – output file
- **select** (*object*) – selects which entities will be written.

Typically select is a subclass of L{Select}, it should have the following methods:

- accept_model(model)
- accept_chain(chain)
- accept_residue(residue)
- accept_atom(atom)

These methods should return 1 if the entity is to be written out, 0 otherwise.

Typically select is a subclass of L{Select}.

__annotations__ = {}

Bio.PDB.PDBList module

Access the PDB over the internet (e.g. to download structures).

class Bio.PDB.PDBList.**PDBList**(*server='https://files.wwpdb.org'*, *pdb=None*, *obsolete_pdb=None*, *verbose=True*)

Bases: object

Quick access to the structure lists on the PDB or its mirrors.

This class provides quick access to the structure lists on the PDB server or its mirrors. The structure lists contain four-letter PDB codes, indicating that structures are new, have been modified or are obsolete. The lists are released on a weekly basis.

It also provides a function to retrieve PDB files from the server. To use it properly, prepare a directory /pdb or the like, where PDB files are stored.

All available file formats (PDB, PDBx/mmCif, PDBML, mmtf) are supported. Please note that large structures (containing >62 chains and/or 99999 ATOM lines) are no longer stored as a single PDB file and by default (when PDB format selected) are not downloaded.

Large structures can be downloaded in other formats, including PDBx/mmCif or as a .tar file (a collection of PDB-like formatted files for a given structure).

If you want to use this module from inside a proxy, add the proxy variable to your environment, e.g. in Unix: export HTTP_PROXY='http://realproxy.charite.de:888' (This can also be added to ~/.bashrc)

PDB_REF = '\n The Protein Data Bank: a computer-based archival file for macromolecular structures.\n F.C.Bernstein, T.F.Koetzle, G.J.B.Williams, E.F.Meyer Jr, M.D.Brice, J.R.Rodgers, O.Kennard, T.Shimanouchi, M.Tasumi\n J. Mol. Biol. 112 pp. 535-542 (1977)\n http://www.pdb.org/.\n '

__init__(*server='https://files.wwpdb.org'*, *pdb=None*, *obsolete_pdb=None*, *verbose=True*)

Initialize the class with the default server or a custom one.

Argument pdb is the local path to use, defaulting to the current directory at the moment of initialisation.

static get_status_list(*url*)

Retrieve a list of pdb codes in the weekly pdb status file from given URL.

Used by get_recent_changes. Typical contents of the list files parsed by this method is now very simply - one PDB name per line.

get_recent_changes()

Return three lists of the newest weekly files (added,mod,obsolete).

Reads the directories with changed entries from the PDB server and returns a tuple of three URL's to the files of new, modified and obsolete entries from the most recent list. The directory with the largest numerical name is used. Returns None if something goes wrong.

Contents of the data/status dir (20031013 would be used);:

drwxrwxr-x 2 1002 sysadmin 512 Oct 6 18:28 20031006 drwxrwxr-x 2 1002 sysadmin 512 Oct 14 02:14 20031013 -rw-r–r– 1 1002 sysadmin 1327 Mar 12 2001 README

get_all_entries()

Retrieve the big file containing all the PDB entries and some annotation.

Returns a list of PDB codes in the index file.

get_all_obsolete()

Return a list of all obsolete entries ever in the PDB.

Returns a list of all obsolete pdb codes that have ever been in the PDB.

Gets and parses the file from the PDB server in the format (the first pdb_code column is the one used). The file looks like this:

```
 LIST OF OBSOLETE COORDINATE ENTRIES AND SUCCESSORS
OBSLTE    31-JUL-94 116L     216L
...
OBSLTE    29-JAN-96 1HFT     2HFT
OBSLTE    21-SEP-06 1HFV     2J5X
OBSLTE    21-NOV-03 1HG6
OBSLTE    18-JUL-84 1HHB     2HHB 3HHB
OBSLTE    08-NOV-96 1HID     2HID
OBSLTE    01-APR-97 1HIU     2HIU
OBSLTE    14-JAN-04 1HKE     1UUZ
...
```

retrieve_pdb_file(*pdb_code*, *obsolete=False*, *pdir=None*, *file_format=None*, *overwrite=False*)

Fetch PDB structure file from PDB server, and store it locally.

The PDB structure's file name is returned as a single string. If obsolete == True, the file will be saved in a special file tree.

NOTE. The default download format has changed from PDB to PDBx/mmCif

Parameters

- **pdb_code** (*string*) – 4-symbols structure Id from PDB (e.g. 3J92).
- **file_format** (*string*) – File format. Available options:
  - "mmCif" (default, PDBx/mmCif file),
  - "pdb" (format PDB),
  - "xml" (PDBML/XML format),
  - "mmtf" (highly compressed),
  - "bundle" (PDB formatted archive for large structure)
- **overwrite** (*bool*) – if set to True, existing structure files will be overwritten. Default: False
- **obsolete** (*bool*) – Has a meaning only for obsolete structures. If True, download the obsolete structure to 'obsolete' folder, otherwise download won't be performed. This option doesn't work for mmtf format as obsoleted structures aren't stored in mmtf. Also doesn't have meaning when parameter pdir is specified. Note: make sure that you are about to download the really obsolete structure. Trying to download non-obsolete structure into obsolete folder will not work and you face the "structure doesn't exists" error. Default: False
- **pdir** (*string*) – put the file in this directory (default: create a PDB-style directory tree)

Returns
filename

Return type
string

update_pdb(*file_format=None*, *with_assemblies=False*)

Update your local copy of the PDB files.

I guess this is the 'most wanted' function from this module. It gets the weekly lists of new and modified pdb entries and automatically downloads the according PDB files. You can call this module as a weekly cron job.

download_pdb_files(*pdb_codes: list[str]*, *obsolete: bool = False*, *pdir: str | None = None*, *file_format: str | None = None*, *overwrite: bool = False*, *max_num_threads: int | None = None*)

Fetch set of PDB structure files from the PDB server and store them locally.

Parameters

- **pdb_codes** – A list of 4-symbol PDB structure IDs
- **obsolete** – Has a meaning only for obsolete structures. If True, download the obsolete structure to 'obsolete' folder. Otherwise, the download won't be performed. This option doesn't work for mmtf format as obsolete structures are not available as mmtf. (default: `False`)
- **pdir** – Put the file in this directory. By default, create a PDB-style directory tree.
- **file_format** – File format. Available options:
  - "mmCif" (default, PDBx/mmCif file),
  - "pdb" (format PDB),
  - "xml" (PMDML/XML format),
  - "mmtf" (highly compressed),
  - "bundle" (PDB formatted archive for large structure).
- **overwrite** – If set to true, existing structure files will be overwritten. (default: `False`)
- **max_num_threads** – The maximum number of threads to use when downloading files

get_all_assemblies(*file_format: str = ''*) → list[tuple[str, str]]

Retrieve the list of PDB entries with an associated bio assembly.

The requested list will be cached to avoid multiple calls to the server.

Parameters
file_format (*str*) – A legacy parameter that is left to avoid breaking changes

Returns
the assemblies

Return type
list

retrieve_assembly_file(*pdb_code*, *assembly_num*, *pdir=None*, *file_format=None*, *overwrite=False*)

Fetch one or more assembly structures associated with a PDB entry.

Unless noted below, parameters are described in `retrieve_pdb_file`.

Parameters
assembly_num (*str*) – assembly number to download.

:rtype : str :return: file name of the downloaded assembly file.

download_all_assemblies(*listfile: str | None = None*, *file_format: str | None = None*, *max_num_threads: int | None = None*)

Retrieve all biological assemblies not in the local PDB copy.

Parameters

- **listfile** – File name to which all assembly codes will be written
- **file_format** – Format in which to download the entries. Available options are "mmCif" or "pdb". Defaults to "mmCif".
- **max_num_threads** – The maximum number of threads to use while downloading the assemblies

download_entire_pdb(*listfile: str | None = None*, *file_format: str | None = None*, *max_num_threads: int | None = None*)

Retrieve all PDB entries not present in the local PDB copy.

NOTE: The default download format has changed from PDB to PDBx/mmCif.

Parameters

- **listfile** – Filename to which all PDB codes will be written
- **file_format** – File format. Available options:
  - "mmCif" (default, PDBx/mmCif file),
  - "pdb" (format PDB),
  - "xml" (PMDML/XML format),
  - "mmtf" (highly compressed),
  - "bundle" (PDB formatted archive for large structure)
- **max_num_threads** – The maximum number of threads to use while downloading PDB entries

download_obsolete_entries(*listfile: str | None = None*, *file_format: str | None = None*, *max_num_threads: int | None = None*)

Retrieve all obsolete PDB entries not present in local obsolete PDB copy.

NOTE: The default download format has changed from PDB to PDBx/mmCif.

Parameters

- **listfile** – Filename to which all PDB codes will be written
- **file_format** – File format. Available options:
  - "mmCif" (default, PDBx/mmCif file),
  - "pdb" (PDB format),
  - "xml" (PMDML/XML format).
- **max_num_threads** – The maximum number of threads to use while downloading PDB entries

get_seqres_file(*savefile='pdb_seqres.txt'*)

Retrieve and save a (big) file containing all the sequences of PDB entries.

Bio.PDB.PDBMLParser module

This module contains a parser for PDBML (PDB XML) files.

PDBML is a representation of PDB data in XML format.

See https://pdbml.wwpdb.org/.

class Bio.PDB.PDBMLParser.**PDBMLParser**

Bases: object

A parser for PDBML (PDB XML) files. See https://pdbml.wwpdb.org/.

This parser is based on the mmCIF parser also provided in the PDB package in the sense that the structure object returned by this parser is equal to the structure returned by the mmCIF parser for any given PDB structure.

__init__()

Initialize a PDBML parser.

get_structure(*source: int | str | bytes | PathLike | TextIO*) → *Structure*

Parse and return the PDB structure from XML source.

Parameters
source (*Union[int, str, bytes, PathLike, TextIO]*) – The XML representation of the PDB structure

Returns
the PDB structure

Return type
Bio.PDB.Structure.Structure

Bio.PDB.PDBParser module

Parser for PDB files.

class Bio.PDB.PDBParser.**PDBParser**(*PERMISSIVE=True*, *get_header=False*, *structure_builder=None*, *QUIET=False*, *is_pqr=False*)

Bases: object

Parse a PDB file and return a Structure object.

__init__(*PERMISSIVE=True*, *get_header=False*, *structure_builder=None*, *QUIET=False*, *is_pqr=False*)

Create a PDBParser object.

The PDB parser call a number of standard methods in an aggregated StructureBuilder object. Normally this object is instantiated by the PDBParser object itself, but if the user provides his/her own StructureBuilder object, the latter is used instead.

Arguments:

- PERMISSIVE - Evaluated as a Boolean. If false, exceptions in constructing the SMCRA data structure are fatal. If true (DEFAULT), the exceptions are caught, but some residues or atoms will be missing. THESE EXCEPTIONS ARE DUE TO PROBLEMS IN THE PDB FILE!.
- get_header - unused argument kept for historical compatibility.
- structure_builder - an optional user implemented StructureBuilder class.

- QUIET - Evaluated as a Boolean. If true, warnings issued in constructing the SMCRA data will be suppressed. If false (DEFAULT), they will be shown. These warnings might be indicative of problems in the PDB file!
- is_pqr - Evaluated as a Boolean. Specifies the type of file to be parsed. If false (DEFAULT) a .pdb file format is assumed. Set it to true if you want to parse a .pqr file instead.

`get_structure`(*id*, *file*)

Return the structure.

Arguments:

- id - string, the id that will be used for the structure
- file - name of the PDB file OR an open filehandle

`get_header`()

Return the header.

`get_trailer`()

Return the trailer.

Bio.PDB.PICIO module

PICIO: read and write Protein Internal Coordinate (.pic) data files.

Bio.PDB.PICIO.**read_PIC**(*file: TextIO*, *verbose: bool = False*, *quick: bool = False*, *defaults: bool = False*) → *Structure*

Load Protein Internal Coordinate (.pic) data from file.

PIC file format:

- comment lines start with #
- **(optional) PDB HEADER record**
  - idcode and deposition date recommended but optional
  - deposition date in PDB format or as changed by Biopython
- (optional) PDB TITLE record
- **repeat:**
  - Biopython Residue Full ID - sets residue IDs of returned structure
  - (optional) PDB N, CA, C ATOM records for chain start
  - (optional) PIC Hedra records for residue
  - (optional) PIC Dihedra records for residue
  - (optional) BFAC records listing AtomKeys and b-factors

An improvement would define relative positions for HOH (water) entries.

Defaults will be supplied for any value if defaults=True. Default values are supplied in ic_data.py, but structures degrade quickly with any deviation from true coordinates. Experiment with *Bio.PDB.internal_coords.IC_Residue.pic_flags* options to *write_PIC()* to verify this.

N.B. dihedron (i-1)C-N-CA-CB is ignored in assembly if O exists.

C-beta is by default placed using O-C-CA-CB, but O is missing in some PDB file residues, which means the sidechain cannot be placed. The alternate CB path (i-1)C-N-CA-CB is provided to circumvent this, but if this is

needed then it must be adjusted in conjunction with PHI ((i-1)C-N-CA-C) as they overlap (see *bond_set()* and *bond_rotate()* to handle this automatically).

Parameters

- **file** (*Bio.File*) – *as_handle()* file name or handle
- **verbose** (*bool*) – complain when lines not as expected
- **quick** (*bool*) – don't check residues for all dihedra (no default values)
- **defaults** (*bool*) – create di/hedra as needed from reference database. Amide proton created if 'H' is in IC_Residue.accept_atoms

Returns

Biopython Structure object, Residues with .internal_coord attributes but no coordinates except for chain start N, CA, C atoms if supplied, **OR** None on parse fail (silent unless verbose=True)

Bio.PDB.PICIO.**read_PIC_seq**(*seqRec:* SeqRecord, *pdbid: str | None = None*, *title: str | None = None*, *chain: str | None = None*) → *Structure*

Read *SeqRecord* into Structure with default internal coords.

Bio.PDB.PICIO.**enumerate_atoms**(*entity*)

Ensure all atoms in entity have serial_number set.

Bio.PDB.PICIO.**pdb_date**(*datestr: str*) → str

Convert yyyy-mm-dd date to dd-month-yy.

Bio.PDB.PICIO.**write_PIC**(*entity*, *file*, *pdbid=None*, *chainid=None*, *picFlags: int = IC_Residue.picFlagsDefault*, *hCut: float | None = None*, *pCut: float | None = None*)

Write Protein Internal Coordinates (PIC) to file.

See *read_PIC()* for file format. See `IC_Residue.pic_accuracy` to vary numeric accuracy. Recurses to lower entity levels (M, C, R).

Parameters

- **entity** (Entity) – Biopython PDB Entity object: S, M, C or R
- **file** (*Bio.File*) – *as_handle()* file name or handle
- **pdbid** (*str*) – PDB idcode, read from entity if not supplied
- **chainid** (*char*) – PDB Chain ID, set from C level entity.id if needed
- **picFlags** (*int*) – boolean flags controlling output, defined in *Bio.PDB.internal_coords.IC_Residue.pic_flags*
  - "psi",
  - "omg",
  - "phi",
  - "tau", # tau hedron (N-Ca-C)
  - "chi1",
  - "chi2",
  - "chi3",
  - "chi4",
  - "chi5",

- ”pomg”, # proline omega
- ”chi”, # chi1 through chi5
- ”classic_b”, # psi | phi | tau | pomg
- ”classic”, # classic_b | chi
- ”hedra”, # all hedra including bond lengths
- ”primary”, # all primary dihedra
- ”secondary”, # all secondary dihedra (fixed angle from primary dihedra)
- ”all”, # hedra | primary | secondary
- ”initAtoms”, # XYZ coordinates of initial Tau (N-Ca-C)
- ”bFactors”

default is everything:

```
picFlagsDefault = (
    pic_flags.all | pic_flags.initAtoms | pic_flags.bFactors
)
```

Usage in your code:

```
# just primary dihedra and all hedra
picFlags = (
    IC_Residue.pic_flags.primary | IC_Residue.pic_flags.hedra
)

# no B-factors:
picFlags = IC_Residue.picFlagsDefault
picFlags &= ~IC_Residue.pic_flags.bFactors
```

read_PIC() with *(defaults=True)* will use default values for anything left out

- **hCut** (*float*) – default None only write hedra with ref db angle std dev greater than this value
- **pCut** (*float*) – default None only write primary dihedra with ref db angle std dev greater than this value

Default values:

Data averaged from Sep 2019 Dunbrack cullpdb_pc20_res2.2_R1.0.

Please see

PISCES: A Protein Sequence Culling Server

‘G. Wang and R. L. Dunbrack, Jr. PISCES: a protein sequence culling server. Bioinformatics, 19:1589-1591, 2003.’

‘primary’ and ‘secondary’ dihedra are defined in ic_data.py. Specifically, secondary dihedra can be determined as a fixed rotation from another known angle, for example N-Ca-C-O can be estimated from N-Ca-C-N (psi).

Standard deviations are listed in <biopython distribution>/Bio/PDB/ic_data.py for default values, and can be used to limit which hedra and dihedra are defaulted vs. output exact measurements from structure (see hCut and pCut above). Default values for primary dihedra (psi, phi, omega, chi1, etc.) are chosen as the most common integer value, not an average.

Raises

- *PDBException* – if entity level is A (Atom)
- **Exception** – if entity does not have .level attribute

Bio.PDB.PSEA module

Wrappers for PSEA, a program for secondary structure assignment.

See this citation for P-SEA, PMID: 9183534

Labesse G, Colloc'h N, Pothier J, Mornon J-P: P-SEA: a new efficient assignment of secondary structure from C_alpha. Comput Appl Biosci 1997 , 13:291-295

ftp://ftp.lmcp.jussieu.fr/pub/sincris/software/protein/p-sea/

`Bio.PDB.PSEA.`**run_psea**(*fname*, *verbose=False*)

Run PSEA and return output filename.

Note that this assumes the P-SEA binary is called "psea" and that it is on the path.

Note that P-SEA will write an output file in the current directory using the input filename with extension ".sea".

Note that P-SEA will not write output to the terminal while run unless
verbose is set to True.

`Bio.PDB.PSEA.`**psea**(*pname*)

Parse PSEA output file.

`Bio.PDB.PSEA.`**psea2HEC**(*pseq*)

Translate PSEA secondary structure string into HEC.

`Bio.PDB.PSEA.`**annotate**(*m*, *ss_seq*)

Apply secondary structure information to residues in model.

class `Bio.PDB.PSEA.`**PSEA**(*model*, *filename*)

Bases: `object`

Define PSEA class.

PSEA object is a wrapper to PSEA program for secondary structure assignment.

__init__(*model*, *filename*)

Initialize the class.

get_seq()

Return secondary structure string.

Bio.PDB.Polypeptide module

Polypeptide-related classes (construction and representation).

Simple example with multiple chains,

```
>>> from Bio.PDB.PDBParser import PDBParser
>>> from Bio.PDB.Polypeptide import PPBuilder
>>> structure = PDBParser().get_structure('2BEG', 'PDB/2BEG.pdb')
>>> ppb=PPBuilder()
```

(continues on next page)

(continued from previous page)

```
>>> for pp in ppb.build_peptides(structure):
...     print(pp.get_sequence())
LVFFAEDVGSNKGAIIGLMVGGVVIA
LVFFAEDVGSNKGAIIGLMVGGVVIA
LVFFAEDVGSNKGAIIGLMVGGVVIA
LVFFAEDVGSNKGAIIGLMVGGVVIA
LVFFAEDVGSNKGAIIGLMVGGVVIA
```

Example with non-standard amino acids using HETATM lines in the PDB file, in this case selenomethionine (MSE):

```
>>> from Bio.PDB.PDBParser import PDBParser
>>> from Bio.PDB.Polypeptide import PPBuilder
>>> structure = PDBParser().get_structure('1A8O', 'PDB/1A8O.pdb')
>>> ppb=PPBuilder()
>>> for pp in ppb.build_peptides(structure):
...     print(pp.get_sequence())
DIRQGPKEPFRDYVDRFYKTLRAEQASQEVKNW
TETLLVQNANPDCKTILKALGPGATLEE
TACQG
```

If you want to, you can include non-standard amino acids in the peptides:

```
>>> for pp in ppb.build_peptides(structure, aa_only=False):
...     print(pp.get_sequence())
...     print("%s %s" % (pp.get_sequence()[0], pp[0].get_resname()))
...     print("%s %s" % (pp.get_sequence()[-7], pp[-7].get_resname()))
...     print("%s %s" % (pp.get_sequence()[-6], pp[-6].get_resname()))
MDIRQGPKEPFRDYVDRFYKTLRAEQASQEVKNWMTETLLVQNANPDCKTILKALGPGATLEEMMTACQG
M MSE
M MSE
M MSE
```

In this case the selenomethionines (the first and also seventh and sixth from last residues) have been shown as M (methionine) by the get_sequence method.

`Bio.PDB.Polypeptide.`**`index_to_one`**(*index*)

Index to corresponding one letter amino acid name.

```
>>> index_to_one(0)
'A'
>>> index_to_one(19)
'Y'
```

`Bio.PDB.Polypeptide.`**`one_to_index`**(*s*)

One letter code to index.

```
>>> one_to_index('A')
0
>>> one_to_index('Y')
19
```

`Bio.PDB.Polypeptide.`**`index_to_three`**(*i*)

Index to corresponding three letter amino acid name.

```
>>> index_to_three(0)
'ALA'
>>> index_to_three(19)
'TYR'
```

Bio.PDB.Polypeptide.**three_to_index**(*s*)

Three letter code to index.

```
>>> three_to_index('ALA')
0
>>> three_to_index('TYR')
19
```

Bio.PDB.Polypeptide.**is_aa**(*residue*, *standard=False*)

Return True if residue object/string is an amino acid.

Parameters

- **residue** (*L{Residue} or string*) – a L{Residue} object OR a three letter amino acid code
- **standard** (*boolean*) – flag to check for the 20 AA (default false)

```
>>> is_aa('ALA')
True
```

Known three letter codes for modified amino acids are supported,

```
>>> is_aa('FME')
True
>>> is_aa('FME', standard=True)
False
```

Bio.PDB.Polypeptide.**is_nucleic**(*residue*, *standard=False*)

Return True if residue object/string is a nucleic acid.

Parameters

- **residue** (*L{Residue} or string*) – a L{Residue} object OR a three letter code
- **standard** (*boolean*) – flag to check for the 8 (DNA + RNA) canonical bases. Default is False.

```
>>> is_nucleic('DA ')
True
```

```
>>> is_nucleic('A  ')
True
```

Known three letter codes for modified nucleotides are supported,

```
>>> is_nucleic('A2L')
True
>>> is_nucleic('A2L', standard=True)
False
```

class Bio.PDB.Polypeptide.**Polypeptide**(*iterable=()*, */*)

Bases: list

A polypeptide is simply a list of L{Residue} objects.

get_ca_list()

Get list of C-alpha atoms in the polypeptide.

Returns
the list of C-alpha atoms

Return type
[L{Atom}, L{Atom}, ...]

get_phi_psi_list()

Return the list of phi/psi dihedral angles.

get_tau_list()

List of tau torsions angles for all 4 consecutive Calpha atoms.

get_theta_list()

List of theta angles for all 3 consecutive Calpha atoms.

get_sequence()

Return the AA sequence as a Seq object.

Returns
polypeptide sequence

Return type
L{Seq}

__repr__()

Return string representation of the polypeptide.

Return <Polypeptide start=START end=END>, where START and END are sequence identifiers of the outer residues.

class Bio.PDB.Polypeptide.**CaPPBuilder**(*radius=4.3*)

Bases: _PPBuilder

Use CA–CA distance to find polypeptides.

__init__(*radius=4.3*)

Initialize the class.

class Bio.PDB.Polypeptide.**PPBuilder**(*radius=1.8*)

Bases: _PPBuilder

Use C–N distance to find polypeptides.

__init__(*radius=1.8*)

Initialize the class.

__annotations__ = {}

Bio.PDB.Residue module

Residue class, used by Structure objects.

class Bio.PDB.Residue.**Residue**(*id*, *resname*, *segid*)

Bases: *Entity*[Chain, Atom]

Represents a residue. A Residue object stores atoms.

__init__(*id*, *resname*, *segid*)

Initialize the class.

__repr__()

Return the residue full id.

strictly_equals(*other: _ResidueT*, *compare_coordinates: bool = False*) → bool

Compare this residue to the other residue using a strict definition of equality.

The residues are equal if they have the same name, identifier, and their constituent atoms are strictly equal.

Parameters

- **other** (Residue) – The residue to compare this residue to
- **compare_coordinates** (*bool*) – Whether to compare the coordinates of the atoms

Returns
Whether the residues are strictly equal

Return type
bool

add(*atom*)

Add an Atom object.

Checks for adding duplicate atoms, and raises a PDBConstructionException if so.

flag_disordered()

Set the disordered flag.

is_disordered()

Return 1 if the residue contains disordered atoms.

get_resname()

Return the residue name.

get_unpacked_list()

Return the list of all atoms, unpack DisorderedAtoms.

get_segid()

Return the segment identifier.

get_atoms()

Return atoms.

__annotations__ = {}

__orig_bases__ = (Bio.PDB.Entity.Entity[ForwardRef('Chain'), ForwardRef('Atom')],)

__parameters__ = ()

class Bio.PDB.Residue.**DisorderedResidue**(*id*)

Bases: *DisorderedEntityWrapper*

DisorderedResidue is a wrapper around two or more Residue objects.

It is used to represent point mutations (e.g. there is a Ser 60 and a Cys 60 residue, each with 50 % occupancy).

__init__(*id*)

Initialize the class.

__repr__()

Return disordered residue full identifier.

add(*atom*)

Add atom to residue.

sort()

Sort the atoms in the child Residue objects.

disordered_add(*residue*)

Add a residue object and use its resname as key.

Arguments:

- residue - Residue object

disordered_remove(*resname*)

Remove a child residue from the DisorderedResidue.

Arguments:

- resname - name of the child residue to remove, as a string.

__annotations__ = {}

Bio.PDB.ResidueDepth module

Calculation of residue depth using command line tool MSMS.

This module uses Michel Sanner's MSMS program for the surface calculation. See: http://mgltools.scripps.edu/packages/MSMS

Residue depth is the average distance of the atoms of a residue from the solvent accessible surface.

Residue Depth:

```
from Bio.PDB.ResidueDepth import ResidueDepth
from Bio.PDB.PDBParser import PDBParser
parser = PDBParser()
structure = parser.get_structure("1a8o", "Tests/PDB/1A8O.pdb")
model = structure[0]
rd = ResidueDepth(model)
print(rd['A',(' ', 152, ' ')])
```

Direct MSMS interface, typical use:

```
from Bio.PDB.ResidueDepth import get_surface
surface = get_surface(model)
```

The surface is a NumPy array with all the surface vertices.

Distance to surface:

```
from Bio.PDB.ResidueDepth import min_dist
coord = (1.113, 35.393,  9.268)
dist = min_dist(coord, surface)
```

where coord is the coord of an atom within the volume bound by the surface (ie. atom depth).

To calculate the residue depth (average atom depth of the atoms in a residue):

```
from Bio.PDB.ResidueDepth import residue_depth
chain = model['A']
res152 = chain[152]
rd = residue_depth(res152, surface)
```

Bio.PDB.ResidueDepth.**get_surface**(*model*, *MSMS='msms'*)

Represent molecular surface as a vertex list array.

Return a NumPy array that represents the vertex list of the molecular surface.

Arguments:

- model - BioPython PDB model object (used to get atoms for input model)
- MSMS - msms executable (used as argument to subprocess.call)

Bio.PDB.ResidueDepth.**min_dist**(*coord*, *surface*)

Return minimum distance between coord and surface.

Bio.PDB.ResidueDepth.**residue_depth**(*residue*, *surface*)

Residue depth as average depth of all its atoms.

Return average distance to surface for all atoms in a residue, ie. the residue depth.

Bio.PDB.ResidueDepth.**ca_depth**(*residue*, *surface*)

Return CA depth.

class Bio.PDB.ResidueDepth.**ResidueDepth**(*model*, *msms_exec=None*)

Bases: *AbstractPropertyMap*

Calculate residue and CA depth for all residues.

__init__(*model*, *msms_exec=None*)

Initialize the class.

__annotations__ = {}

Bio.PDB.SASA module

Calculation of solvent accessible surface areas for Bio.PDB entities.

Uses the "rolling ball" algorithm developed by Shrake & Rupley algorithm, which uses a sphere (of equal radius to a solvent molecule) to probe the surface of the molecule.

Reference:

Shrake, A; Rupley, JA. (1973). J Mol Biol "Environment and exposure to solvent of protein atoms. Lysozyme and insulin".

class Bio.PDB.SASA.**ShrakeRupley**(*probe_radius=1.40*, *n_points=100*, *radii_dict=None*)

Bases: object

Calculates SASAs using the Shrake-Rupley algorithm.

__init__(*probe_radius=1.40*, *n_points=100*, *radii_dict=None*)

Initialize the class.

Parameters

- **probe_radius** (*float*) – radius of the probe in A. Default is 1.40, roughly the radius of a water molecule.
- **n_points** (*int*) – resolution of the surface of each atom. Default is 100. A higher number of points results in more precise measurements, but slows down the calculation.
- **radii_dict** (*dict*) – user-provided dictionary of atomic radii to use in the calculation. Values will replace/complement those in the default ATOMIC_RADII dictionary.

```
>>> sr = ShrakeRupley()
>>> sr = ShrakeRupley(n_points=960)
>>> sr = ShrakeRupley(radii_dict={"O": 3.1415})
```

compute(*entity*, *level='A'*)

Calculate surface accessibility surface area for an entity.

The resulting atomic surface accessibility values are attached to the .sasa attribute of each entity (or atom), depending on the level. For example, if level="R", all residues will have a .sasa attribute. Atoms will always be assigned a .sasa attribute with their individual values.

Parameters

- **entity** (*Bio.PDB.Entity*) – input entity.
- **level** – the level at which ASA values are assigned, which can be one of "A" (Atom), "R" (Residue), "C" (Chain), "M" (Model), or "S" (Structure). The ASA value of an entity is the sum of all ASA values of its children. Defaults to "A".

```
>>> from Bio.PDB import PDBParser
>>> from Bio.PDB.SASA import ShrakeRupley
>>> p = PDBParser(QUIET=1)
>>> # This assumes you have a local copy of 1LCD.pdb in a directory called "PDB"
>>> struct = p.get_structure("1LCD", "PDB/1LCD.pdb")
>>> sr = ShrakeRupley()
>>> sr.compute(struct, level="S")
>>> print(round(struct.sasa, 2))
7053.43
>>> print(round(struct[0]["A"][11]["OE1"].sasa, 2))
9.64
```

Bio.PDB.SCADIO module

SCADIO: write OpenSCAD program to create protein structure 3D model.

3D printing a protein structure is a non-trivial exercise due to the overall complexity and the general requirement for supporting overhang regions while printing. This software is a path to generating a model for printing (e.g. an STL file), and does not address the issues around converting the model to a physical product. OpenSCAD <http://www.openscad.org/> can create a printable model from the script this software produces. MeshMixer <http://www.meshmixer.com/>, various slicer software, and the 3D printer technology available to you provide options for addressing the problems around physically rendering the model.

The model generated here consists of OpenSCAD primitives, e.g. spheres and cylinders, representing individual atoms and bonds in an explicit model of a protein structure. The benefit is that individual atoms/bonds may be selected for specific print customizations relevant to 3D printing (such as rotatable bond mechanisms or hydrogen bond magnets). Alternatively, use e.g. Chimera to render a structure as ribbons or similar for printing as a single object.

I suggest generating your initial model using the OpenSCAD script provided here, then modifying that script according to your needs. Changing the atomScale and bondRadius values can simplify the model by removing gaps and the corresponding need for supports, or you may wish to modify the hedronDispatch() routine to select residues or chain sections for printing separately and subsequently joining with rotatable bonds. During this development phase you will likely have your version include only the data matrices generated here, by using the *includeCode=False* option to write_SCAD(). An example project using rotatable backbone and magnetic hydrogen bonds is at <https://www.thingiverse.com/thing:3957471>.

Bio.PDB.SCADIO.**write_SCAD**(*entity*, *file*, *scale=None*, *pdbid=None*, *backboneOnly=False*, *includeCode=True*, *maxPeptideBond=None*, *start=None*, *fin=None*, *handle='protein'*)

Write hedron assembly to file as OpenSCAD matrices.

This routine calls both *IC_Chain.internal_to_atom_coordinates()* and *IC_Chain.atom_to_internal_coordinates()* due to requirements for scaling, explicit bonds around rings, and setting the coordinate space of the output model.

Output data format is primarily:

- **matrix for each hedron:**
  len1, angle2, len3, atom covalent bond class, flags to indicate atom/bond represented in previous hedron (OpenSCAD very slow with redundant overlapping elements), flags for bond features
- transform matrices to assemble each hedron into residue dihedra sets
- transform matrices for each residue to position in chain

OpenSCAD software is included in this Python file to process these matrices into a model suitable for a 3D printing project.

Parameters

- **entity** – Biopython PDB *Structure* entity structure data to export
- **file** – Bipoython *as_handle()* filename or open file pointer file to write data to
- **scale** (*float*) – units (usually mm) per angstrom for STL output, written in output
- **pdbid** (*str*) – PDB idcode, written in output. Defaults to '0PDB' if not supplied and no 'idcode' set in entity
- **backboneOnly** (*bool*) – default False. Do not output side chain data past Cbeta if True
- **includeCode** (*bool*) – default True. Include OpenSCAD software (inline below) so output file can be loaded into OpenSCAD; if False, output data matrices only

- **maxPeptideBond** (*float*) – Optional default None. Override the cut-off in IC_Chain class (default 1.4) for detecting chain breaks. If your target has chain breaks, pass a large number here to create a very long 'bond' spanning the break.
- **start,fin** (*int*) – default None Parameters for internal_to_atom_coords() to limit chain segment.
- **handle** (*str*) – default 'protein' name for top level of generated OpenSCAD matrix structure

See *IC_Residue.set_flexible()* to set flags for specific residues to have rotatable bonds, and *IC_Residue.set_hbond()* to include cavities for small magnets to work as hydrogen bonds. See <https://www.thingiverse.com/thing:3957471> for implementation example.

The OpenSCAD code explicitly creates spheres and cylinders to represent atoms and bonds in a 3D model. Options are available to support rotatable bonds and magnetic hydrogen bonds.

Matrices are written to link, enumerate and describe residues, dihedra, hedra, and chains, mirroring contents of the relevant IC_* data structures.

The OpenSCAD matrix of hedra has additional information as follows:

- **the atom and bond state (single, double, resonance) are logged**
  so that covalent radii may be used for atom spheres in the 3D models
- bonds and atoms are tracked so that each is only created once
- **bond options for rotation and magnet holders for hydrogen bonds**
  may be specified (see *IC_Residue.set_flexible()* and *IC_Residue.set_hbond()*)

Note the application of *Bio.PDB.internal_coords.IC_Chain.MaxPeptideBond* : missing residues may be linked (joining chain segments with arbitrarily long bonds) by setting this to a large value.

Note this uses the serial assembly per residue, placing each residue at the origin and supplying the coordinate space transform to OpenaSCAD

All ALTLOC (disordered) residues and atoms are written to the output model. (see *Bio.PDB.internal_coords.IC_Residue.no_altloc*)

Bio.PDB.Selection module

Selection of atoms, residues, etc.

Bio.PDB.Selection.**uniqueify**(*items*)

Return a list of the unique items in the given iterable.

Order is NOT preserved.

Bio.PDB.Selection.**get_unique_parents**(*entity_list*)

Translate a list of entities to a list of their (unique) parents.

Bio.PDB.Selection.**unfold_entities**(*entity_list*, *target_level*)

Unfold entities list to a child level (e.g. residues in chain).

Unfold a list of entities to a list of entities of another level. E.g.:

list of atoms -> list of residues list of modules -> list of atoms list of residues -> list of chains

- entity_list - list of entities or a single entity
- target_level - char (A, R, C, M, S)

Note that if entity_list is an empty list, you get an empty list back:

```
>>> unfold_entities([], "A")
[]
```

Bio.PDB.Structure module

The structure class, representing a macromolecular structure.

class Bio.PDB.Structure.**Structure**(*id*)

Bases: *Entity*[None, Model]

The Structure class contains a collection of Model instances.

__init__(*id*)

Initialize the class.

__repr__()

Return the structure identifier.

get_models()

Return models.

get_chains()

Return chains from models.

get_residues()

Return residues from chains.

get_atoms()

Return atoms from residue.

atom_to_internal_coordinates(*verbose: bool = False*) → None

Create/update internal coordinates from Atom X,Y,Z coordinates.

Internal coordinates are bond length, angle and dihedral angles.

Parameters

bool (*verbose*) – default False describe runtime problems

internal_to_atom_coordinates(*verbose: bool = False*) → None

Create/update atom coordinates from internal coordinates.

Parameters

bool (*verbose*) – default False describe runtime problems

Raises

Exception – if any chain does not have .internal_coord attribute

__annotations__ = {}

__orig_bases__ = (Bio.PDB.Entity.Entity[NoneType, ForwardRef('Model')],)

__parameters__ = ()

Bio.PDB.StructureAlignment module

Map residues of two structures to each other based on a FASTA alignment.

class Bio.PDB.StructureAlignment.**StructureAlignment**(*fasta_align*, *m1*, *m2*, *si=0*, *sj=1*)

Bases: object

Class to align two structures based on an alignment of their sequences.

__init__(*fasta_align*, *m1*, *m2*, *si=0*, *sj=1*)

Initialize.

Attributes:

- fasta_align - Alignment object
- m1, m2 - two models
- si, sj - the sequences in the Alignment object that correspond to the structures

get_maps()

Map residues between the structures.

Return two dictionaries that map a residue in one structure to the equivealent residue in the other structure.

get_iterator()

Create an iterator over all residue pairs.

Bio.PDB.StructureBuilder module

Consumer class that builds a Structure object.

This is used by the PDBParser and MMCIFparser classes.

class Bio.PDB.StructureBuilder.**StructureBuilder**

Bases: object

Deals with constructing the Structure object.

The StructureBuilder class is used by the PDBParser classes to translate a file to a Structure object.

__init__()

Initialize this instance.

set_header(*header*)

Set header.

set_line_counter(*line_counter: int*)

Tracks line in the PDB file that is being parsed.

Arguments:

- line_counter - int

init_structure(*structure_id: str*)

Initialize a new Structure object with given id.

Arguments:

- structure_id - string

init_model(*model_id: int*, *serial_num: int | None = None*)

Create a new Model object with given id.

Arguments:

- id - int
- serial_num - int

init_chain(*chain_id: str*)

Create a new Chain object with given id.

Arguments:

- chain_id - string

init_seg(*segid: str*)

Flag a change in segid.

Arguments:

- segid - string

init_residue(*resname: str*, *field: str*, *resseq: int*, *icode: str*)

Create a new Residue object.

Arguments:

- resname - string, e.g. "ASN"
- field - hetero flag, "W" for waters, "H" for hetero residues, otherwise blank.
- resseq - int, sequence identifier
- icode - string, insertion code

init_atom(*name: str*, *coord: ndarray*, *b_factor: float*, *occupancy: float*, *altloc: str*, *fullname: str*, *serial_number=None*, *element: str | None = None*, *pqr_charge: float | None = None*, *radius: float | None = None*, *is_pqr: bool = False*)

Create a new Atom object.

Arguments:

- name - string, atom name, e.g. CA, spaces should be stripped
- coord - NumPy array (Float0, length 3), atomic coordinates
- b_factor - float, B factor
- occupancy - float
- altloc - string, alternative location specifier
- fullname - string, atom name including spaces, e.g. " CA "
- element - string, upper case, e.g. "HG" for mercury
- pqr_charge - float, atom charge (PQR format)
- radius - float, atom radius (PQR format)
- is_pqr - boolean, flag to specify if a .pqr file is being parsed

set_anisou(*anisou_array*)

Set anisotropic B factor of current Atom.

set_siguij(*siguij_array*)

Set standard deviation of anisotropic B factor of current Atom.

set_sigatm(*sigatm_array*)

Set standard deviation of atom position of current Atom.

get_structure()

Return the structure.

set_symmetry(*spacegroup*, *cell*)

Set symmetry.

Bio.PDB.Superimposer module

Superimpose two structures.

class Bio.PDB.Superimposer.**Superimposer**

Bases: object

Rotate/translate one set of atoms on top of another to minimize RMSD.

__init__()

Initialize the class.

set_atoms(*fixed*, *moving*)

Prepare translation/rotation to minimize RMSD between atoms.

Put (translate/rotate) the atoms in fixed on the atoms in moving, in such a way that the RMSD is minimized.

Parameters

- **fixed** – list of (fixed) atoms
- **moving** – list of (moving) atoms

apply(*atom_list*)

Rotate/translate a list of atoms.

Bio.PDB.alphafold_db module

A module for interacting with the AlphaFold Protein Structure Database.

See the database website and the API docs.

Bio.PDB.alphafold_db.**get_predictions**(*qualifier: str*) → Iterator[dict]

Get all AlphaFold predictions for a UniProt accession.

Parameters

qualifier (*str*) – A UniProt accession, e.g. P00520

Returns

The AlphaFold predictions

Return type

Iterator[dict]

Bio.PDB.alphafold_db.**download_cif_for**(*prediction: dict*, *directory: str | bytes | PathLike | None = None*) → str

Download the mmCIF file for an AlphaFold prediction.

Downloads the file to the current working directory if no destination is specified.

Parameters

- **prediction** (*dict*) – An AlphaFold prediction
- **directory** (*Union[int, str, bytes, PathLike], optional*) – The directory to write the mmCIF data to, defaults to the current working directory

Returns
The path to the mmCIF file

Return type
str

Bio.PDB.alphafold_db.**get_structural_models_for**(*qualifier: str*, *mmcif_parser: MMCIFParser | None = None*, *directory: str | bytes | PathLike | None = None*) → Iterator[*Structure*]

Get the PDB structures for a UniProt accession.

Downloads the mmCIF files to the directory if they are not present.

Parameters

- **qualifier** (*str*) – A UniProt accession, e.g. P00520
- **mmcif_parser** (*MMCIFParser, optional*) – The mmCIF parser to use, defaults to `MMCIFParser()`
- **directory** (*Union[int, str, bytes, PathLike], optional*) – The directory to store the mmCIF data, defaults to the current working directory

Returns
An iterator over the PDB structures

Return type
Iterator[PDBStructure]

Bio.PDB.binary_cif module

A module to interact with BinaryCIF-formatted files.

class Bio.PDB.binary_cif.**BinaryCIFParser**

Bases: `object`

A parser for BinaryCIF files.

See the BinaryCIF specification.

__init__()

Initialize a BinaryCIF parser.

get_structure(*id: str | None*, *source: str*) → *Structure*

Parse and return the PDB structure from a BinaryCIF file.

Parameters

- **id** (*str*) – the PDB code for this structure

- **source** (*str*) – the path to the BinaryCIF file

Returns
the PDB structure

Return type
Bio.PDB.Structure.Structure

Bio.PDB.ccealign module

Pairwise structure alignment of 3D structures using combinatorial extension.

This module implements a single function: run_cealign. Refer to its docstring for more documentation on usage and implementation.

`Bio.PDB.ccealign.`**`run_cealign`**(*coordsA*, *coordsB*, *fragmentSize*, *gapMax*) → list

Find the optimal alignments between two structures, using CEAlign.

Arguments: - listA: List of lists with coordinates for structure A. - listB: List of lists with coordinates for structure B. - fragmentSize: Size of fragments to be used in alignment. - gapMax: Maximum gap allowed between two aligned fragment pairs.

Bio.PDB.cealign module

Protein Structural Alignment using Combinatorial Extension.

Python code written by Joao Rodrigues. C++ code and Python/C++ interface adapted from open-source Pymol and originally written by Jason Vertrees. The original license and notices are available in *cealign* folder.

Reference

Shindyalov, I.N., Bourne P.E. (1998). "Protein structure alignment by incremental combinatorial extension (CE) of the optimal path". Protein Engineering. 11 (9): 739–747. PMID 9796821.

class `Bio.PDB.cealign.`**`CEAligner`**(*window_size=8*, *max_gap=30*)

Bases: `object`

Protein Structure Alignment by Combinatorial Extension.

`__init__`(*window_size=8*, *max_gap=30*)

Superimpose one set of atoms onto another using structural data.

Structures are superimposed using guide atoms, CA and C4', for protein and nucleic acid molecules respectively.

Parameters

window_size
[float, optional] CE algorithm parameter. Used to define paths when building the CE similarity matrix. Default is 8.

max_gap
[float, optional] CE algorithm parameter. Maximum gap size. Default is 30.

get_guide_coord_from_structure(*structure*)

Return the coordinates of guide atoms in the structure.

We use guide atoms (C-alpha and C4' atoms) since it is much faster than using all atoms in the calculation without a significant loss in accuracy.

set_reference(*structure*)

Define a reference structure onto which all others will be aligned.

align(*structure*, *transform=True*)

Align the input structure onto the reference structure.

Parameters

transform: bool, optional
If True (default), apply the rotation/translation that minimizes the RMSD between the two structures to the input structure. If False, the structure is not modified but the optimal RMSD will still be calculated.

Bio.PDB.ic_data module

Per residue backbone and sidechain hedra and dihedra definitions.

Find this file in <your biopython distribution>/Bio/PDB/ic_data.py

Listed in order of output for internal coordinates (.pic) output file. Require sufficient overlap to link all defined dihedra. Entries in these tables without corresponding atom coordinates are ignored.

<http://www.imgt.org/IMGTeducation/Aide-memoire/_UK/aminoacids/formuleAA/> for naming of individual atoms

Bio.PDB.ic_rebuild module

Convert XYZ Structure to internal coordinates and back, test result.

Bio.PDB.ic_rebuild.**structure_rebuild_test**(*entity*, *verbose: bool = False*, *quick: bool = False*) → dict

Test rebuild PDB structure from internal coordinates.

Generates internal coordinates for entity and writes to a .pic file in memory, then generates XYZ coordinates from the .pic file and compares the resulting entity against the original.

See `IC_Residue.pic_accuracy` to vary numeric accuracy of the intermediate .pic file if the only issue is small differences in coordinates.

Note that with default settings, deuterated initial structures will fail the comparison, as will structures loaded with alternate *IC_Residue.accept_atoms* settings. Use *quick=True* and/or variations on *AtomKey.d2h* and *IC_Residue.accept_atoms* settings.

Parameters

- **entity** (Entity) – Biopython Structure, Model or Chain. Structure to test
- **verbose** (*bool*) – default False. print extra messages
- **quick** (*bool*) – default False. only check the internal coords atomArrays are identical

Returns
dict comparison dict from *compare_residues()*

Bio.PDB.ic_rebuild.**report_IC**(*entity:* Structure | Model | Chain | Residue, *reportDict: dict[str, Any] | None = None*, *verbose: bool = False*) → dict[str, Any]

Generate dict with counts of ic data elements for each entity level.

reportDict entries are:

- idcode : PDB ID
- hdr : PDB header lines
- mdl : models
- chn : chains
- res : residue objects
- res_e : residues with dihedra and/or hedra
- dih : dihedra
- hed : hedra

Parameters
entity (Entity) – Biopython PDB Entity object: S, M, C or R

Raises
- ***PDBException*** – if entity level not S, M, C, or R
- **Exception** – if entity does not have .level attribute

Returns
dict with counts of IC data elements

Bio.PDB.ic_rebuild.**IC_duplicate**(*entity*) → *Structure*

Duplicate structure entity with IC data, no atom coordinates.

Employs *write_PIC()*, *read_PIC()* with StringIO buffer. Calls *Chain.atom_to_internal_coordinates()* if needed.

Parameters
entity (Entity) – Biopython PDB Entity (will fail for Atom)

Returns
Biopython PDBStructure, no Atom objects except initial coords

Bio.PDB.ic_rebuild.**compare_residues**(*e0:* Structure | Model | Chain, *e1:* Structure | Model | Chain, *verbose: bool = False*, *quick: bool = False*, *rtol: float | None = None*, *atol: float | None = None*) → dict[str, Any]

Compare full IDs and atom coordinates for 2 Biopython PDB entities.

Skip DNA and HETATMs.

Parameters
- **e0,e1** (Entity) – Biopython PDB Entity objects (S, M or C). Structures, Models or Chains to be compared
- **verbose** (*bool*) – Whether to print mismatch info, default False
- **quick** (*bool*) – default False. Only check atomArrays are identical, aCoordMatchCount=0 if different

- **atol** (*float rtol,*) – default 1e-03, 1e-03 or round to 3 places. NumPy allclose parameters; default is to round atom coordinates to 3 places and test equal. For 'quick' will use defaults above for comparing atomArrays

Returns dict
Result counts for Residues, Full ID match Residues, Atoms, Full ID match atoms, and Coordinate match atoms; report string; error status (bool)

Bio.PDB.ic_rebuild.**write_PDB**(*entity:* Structure, *file: str*, *pdbid: str | None = None*, *chainid: str | None = None*) → None

Write PDB file with HEADER and TITLE if available.

Bio.PDB.internal_coords module

Classes to support internal coordinates for protein structures.

Internal coordinates comprise Psi, Omega and Phi dihedral angles along the protein backbone, Chi angles along the sidechains, and all 3-atom angles and bond lengths defining a protein chain. These routines can compute internal coordinates from atom XYZ coordinates, and compute atom XYZ coordinates from internal coordinates.

Secondary benefits include the ability to align and compare residue environments in 3D structures, support for 2D atom distance plots, converting a distance plot plus chirality information to a structure, generating an OpenSCAD description of a structure for 3D printing, and reading/writing structures as internal coordinate data files.

Usage:

```
from Bio.PDB.PDBParser import PDBParser
from Bio.PDB.Chain import Chain
from Bio.PDB.internal_coords import *
from Bio.PDB.PICIO import write_PIC, read_PIC, read_PIC_seq
from Bio.PDB.ic_rebuild import write_PDB, IC_duplicate, structure_rebuild_test
from Bio.PDB.SCADIO import write_SCAD
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.PDB.PDBIO import PDBIO
import numpy as np

# load a structure as normal, get first chain
parser = PDBParser()
myProtein = parser.get_structure("7rsa", "pdb7rsa.ent")
myChain = myProtein[0]["A"]

# compute bond lengths, angles, dihedral angles
myChain.atom_to_internal_coordinates(verbose=True)

# check myChain makes sense (can get angles and rebuild same structure)
resultDict = structure_rebuild_test(myChain)
assert resultDict['pass'] == True

# get residue 1 chi2 angle
r1 = next(myChain.get_residues())
r1chi2 = r1.internal_coord.get_angle("chi2")

# rotate residue 1 chi2 angle by 120 degrees (loops w/in +/-180)
r1.internal_coord.set_angle("chi2", r1chi2 + 120.0)
```

(continues on next page)

(continued from previous page)

```
# or
r1.internal_coord.bond_rotate("chi2", 120.0)
# update myChain XYZ coordinates with chi2 changed
myChain.internal_to_atom_coordinates()
# write new conformation with PDBIO
write_PDB(myProtein, "myChain.pdb")
# or just the ATOM records without headers:
io = PDBIO()
io.set_structure(myProtein)
io.save("myChain2.pdb")

# write chain as 'protein internal coordinates' (.pic) file
write_PIC(myProtein, "myChain.pic")
# read .pic file
myProtein2 = read_PIC("myChain.pic")

# create default structure for random sequence by reading as .pic file
myProtein3 = read_PIC_seq(
    SeqRecord(
        Seq("GAVLIMFPSTCNQYWDEHKR"),
        id="1RND",
        description="my random sequence",
    )
)
myProtein3.internal_to_atom_coordinates()
write_PDB(myProtein3, "myRandom.pdb")

# access the all-dihedrals array for the chain, e.g. residue 1 chi2 angle:
r1chi2_obj = r1.internal_coord.pick_angle("chi2")
# or same thing: r1chi2_obj = r1.internal_coord.pick_angle("CA:CB:CG:CD")
r1chi2_key = r1chi2_obj.atomkeys
# r1chi2_key is tuple of AtomKeys (1_K_CA, 1_K_CB, 1_K_CG, 1_K_CD)
r1chi2_index = myChain.internal_coord.dihedraNdx[r1chi2_key]
# or same thing: r1chi2_index = r1chi2_obj.ndx
r1chi2_value = myChain.internal_coord.dihedraAngle[r1chi2_index]
# also true: r1chi2_obj == myChain.internal_coord.dihedra[r1chi2_index]

# access the array of all atoms for the chain, e.g. residue 1 C-beta
r1_cBeta_index = myChain.internal_coord.atomArrayIndex[AtomKey("1_K_CB")]
r1_cBeta_coords = myChain.internal_coord.atomArray[r1_cBeta_index]
# r1_cBeta_coords = [ x, y, z, 1.0 ]

# the Biopython Atom coord array is now a view into atomArray, so
assert r1_cBeta_coords[1] == r1["CB"].coord[1]
r1_cBeta_coords[1] += 1.0  # change the Y coord 1 angstrom
assert r1_cBeta_coords[1] == r1["CB"].coord[1]
# they are always the same (they share the same memory)
r1_cBeta_coords[1] -= 1.0  # restore

# create a selector to filter just the C-alpha atoms from the all atom array
atmNameNdx = AtomKey.fields.atm
atomArrayIndex = myChain.internal_coord.atomArrayIndex
```

(continues on next page)

(continued from previous page)

```
CaSelect = [
    atomArrayIndex.get(k) for k in atomArrayIndex.keys() if k.akl[atmNameNdx] == "CA"
]
# now the ordered array of C-alpha atom coordinates is:
CA_coords = myChain.internal_coord.atomArray[CaSelect]
# note this uses Numpy fancy indexing, so CA_coords is a new copy

# create a C-alpha distance plot
caDistances = myChain.internal_coord.distance_plot(CaSelect)
# display with e.g. MatPlotLib:
# import matplotlib.pyplot as plt
# plt.imshow(caDistances, cmap="hot", interpolation="nearest")
# plt.show()

# build structure from distance plot:
## create the all-atom distance plot
distances = myChain.internal_coord.distance_plot()
## get the sign of the dihedral angles
chirality = myChain.internal_coord.dihedral_signs()
## get new, empty data structure : copy data structure from myChain
myChain2 = IC_duplicate(myChain)[0]["A"]
cic2 = myChain2.internal_coord
## clear the new atomArray and di/hedra value arrays, just for proof
cic2.atomArray = np.zeros((cic2.AAsiz, 4), dtype=np.float64)
cic2.dihedraAngle[:] = 0.0
cic2.hedraAngle[:] = 0.0
cic2.hedraL12[:] = 0.0
cic2.hedraL23[:] = 0.0
## copy just the first N-Ca-C coords so structures will superimpose:
cic2.copy_initNCaCs(myChain.internal_coord)
## copy distances to chain arrays:
cic2.distplot_to_dh_arrays(distances, chirality)
## compute angles and dihedral angles from distances:
cic2.distance_to_internal_coordinates()
## generate XYZ coordinates from internal coordinates:
myChain2.internal_to_atom_coordinates()
## confirm result atomArray matches original structure:
assert np.allclose(cic2.atomArray, myChain.internal_coord.atomArray)

# superimpose all phe-phe pairs - quick hack just to demonstrate concept
# for analyzing pairwise residue interactions.  Generates PDB ATOM records
# placing each PHE at origin and showing all other PHEs in environment
## shorthand for key variables:
cic = myChain.internal_coord
resNameNdx = AtomKey.fields.resname
aaNdx = cic.atomArrayIndex
## select just PHE atoms:
pheAtomSelect = [aaNdx.get(k) for k in aaNdx.keys() if k.akl[resNameNdx] == "F"]
aaF = cic.atomArray[ pheAtomSelect ]  # numpy fancy indexing makes COPY not view

for ric in cic.ordered_aa_ic_list:  # internal_coords version of get_residues()
    if ric.rbase[2] == "F":  # if PHE, get transform matrices for chi1 dihedral
```

(continues on next page)

(continued from previous page)

```
        chi1 = ric.pick_angle("N:CA:CB:CG")  # chi1 space has C-alpha at origin
        cst = np.transpose(chi1.cst)  # transform TO chi1 space
        # rcst = np.transpose(chi1.rcst)  # transform FROM chi1 space
        cic.atomArray[pheAtomSelect] = aaF.dot(cst)  # transform just the PHEs
        for res in myChain.get_residues():  # print PHEs in new coordinate space
            if res.resname in ["PHE"]:
                print(res.internal_coord.pdb_residue_string())
        cic.atomArray[pheAtomSelect] = aaF  # restore coordinate space from copy

# write OpenSCAD program of spheres and cylinders to 3d print myChain backbone
## set atom load filter to accept backbone only:
IC_Residue.accept_atoms = IC_Residue.accept_backbone
## delete existing data to force re-read of all atoms:
myChain.internal_coord = None
write_SCAD(myChain, "myChain.scad", scale=10.0)
```

See the *''Internal coordinates module"* section of the *Biopython Tutorial and Cookbook* for further discussion.

Terms and key data structures: Internal coordinates are defined on sequences of atoms which span residues or follow accepted nomenclature along sidechains. To manage these sequences and support Biopython's disorder mechanisms, *AtomKey* specifiers are implemented to capture residue, atom and variant identification in a single object. A *Hedron* object is specified as three sequential AtomKeys, comprising two bond lengths and the bond angle between them. A *Dihedron* consists of four sequential AtomKeys, linking two Hedra with a dihedral angle between them.

Algorithmic overview: The Internal Coordinates module combines a specification of connected atoms as hedra and dihedra in the *ic_data* file with routines here to transform XYZ coordinates of these atom sets between a local coordinate system and the world coordinates supplied in e.g. a PDB or mmCif data file. The local coordinate system places the center atom of a hedron at the origin (0,0,0), one leg on the +Z axis, and the other leg on the XZ plane (see *Hedron*). Measurement and creation or manipulation of hedra and dihedra in the local coordinate space is straightforward, and the calculated transformation matrices enable assembling these subunits into a protein chain starting from supplied (PDB) coordinates for the initial N-Ca-C atoms.

Psi and Phi angles are defined on atoms from adjacent residues in a protein chain, see e.g. *pick_angle()* and *ic_data* for the relevant mapping between residues and backbone dihedral angles.

Transforms to and from the dihedron local coordinate space described above are accessible via *IC_Chain.dCoordSpace* and *Dihedron* attributes .cst and .rcst, and may be applied in the alignment and comparison of residues and their environments with code along the lines of:

```
chi1 = ric0.pick_angle("chi1") # chi1 space defined with CA at origin
cst = np.transpose(chi1.cst) # transform TO chi1 local space
newAtomCoords = oldAtomCoords.dot(cst)
```

The core algorithms were developed independently during 1993-4 for ''Development and Application of a Three-dimensional Description of Amino Acid Environments in Protein," Miller, Douthart, and Dunker, Advances in Molecular Bioinformatics, IOS Press, 1994, ISBN 90 5199 172 x, pp. 9-30.

A Protein Internal Coordinate (.pic) file format is defined to capture sufficient detail to reproduce a PDB file from chain starting coordinates (first residue N, Ca, C XYZ coordinates) and remaining internal coordinates. These files are used internally to verify that a given structure can be regenerated from its internal coordinates. See *PICIO* for reading and writing .pic files and *structure_rebuild_test()* to determine if a specific PDB or mmCif datafile has sufficient information to interconvert between cartesian and internal coordinates.

Internal coordinates may also be exported as OpenSCAD data arrays for generating 3D printed protein models. OpenSCAD software is provided as a starting point and proof-of-concept for generating such models. See *SCADIO* and this Thingiverse project for a more advanced example.

Refer to *distance_plot()* and *distance_to_internal_coordinates()* for converting structure data to/from 2D distance plots.

The following classes comprise the core functionality for processing internal coordinates and are sufficiently related and coupled to place them together in this module:

***IC_Chain*: Extends Biopython Chain on .internal_coord attribute.**
Manages connected sequence of residues and chain breaks; holds numpy arrays for all atom coordinates and bond geometries. For 'parallel' processing IC_Chain methods operate on these arrays with single numpy commands.

***IC_Residue*: Extends Biopython Residue on .internal_coord attribute.**
Access for per residue views on internal coordinates and methods for serial (residue by residue) assembly.

***Dihedron*: four joined atoms forming a dihedral angle.**
Dihedral angle, homogeneous atom coordinates in local coordinate space, references to relevant Hedra and IC_Residue. Getter methods for residue dihedral angles, bond angles and bond lengths.

***Hedron*: three joined atoms forming a plane.**
Contains homogeneous atom coordinates in local coordinate space as well as bond lengths and angle between them.

***Edron*: base class for Hedron and Dihedron classes.**
Tuple of AtomKeys comprising child, string ID, mainchain membership boolean and other routines common for both Hedra and Dihedra. Implements rich comparison.

***AtomKey*: keys (dictionary and string) for referencing atom sequences.**
Capture residue and disorder/occupancy information, provides a no-whitespace key for .pic files, and implements rich comparison.

Custom exception classes: *HedronMatchError* and *MissingAtomError*

class Bio.PDB.internal_coords.**IC_Chain**(*parent*, *verbose: bool = False*)

Bases: object

Class to extend Biopython Chain with internal coordinate data.

Attributes

chain: object reference
The Biopython *Bio.PDB.Chain.Chain* object this extends

MaxPeptideBond: float
Class attribute to detect chain breaks. Override for fully contiguous chains with some very long bonds - e.g. for 3D printing (OpenSCAD output) a structure with missing residues. *MaxPeptideBond*

ParallelAssembleResidues: bool
Class attribute affecting internal_to_atom_coords. Short (50 residue and less) chains are faster to assemble without the overhead of creating numpy arrays, and the algorithm is easier to understand and trace processing a single residue at a time. Clearing (set to False) this flag will switch to the serial algorithm

ordered_aa_ic_list: list
IC_Residue objects internal_coords algorithms can process (e.g. no waters)

initNCaC: List of N, Ca, C AtomKey tuples (NCaCKeys).
NCaCKeys start chain segments (first residue or after chain break). These 3 atoms define the coordinate space for a contiguous chain segment, as initially specified by PDB or mmCIF file.

AAsiz = int
AtomArray size, number of atoms in this chain

atomArray: numpy array
homogeneous atom coords ([x,, y, z, 1.0]) for every atom in chain

atomArrayIndex: dict
maps AtomKeys to atomArray indexes

hedra: dict
Hedra forming residues in this chain; indexed by 3-tuples of AtomKeys.

hedraLen: int
length of hedra dict

hedraNdx: dict
maps hedra AtomKeys to numeric index into hedra data arrays e.g. hedraL12 below

a2ha_map: [hedraLen x 3]
atom indexes in hedraNdx order

dihedra: dict
Dihedra forming residues in this chain; indexed by 4-tuples of AtomKeys.

dihedraLen: int
length of dihedra dict

dihedraNdx: dict
maps dihedra AtomKeys to dihedra data arrays e.g. dihedraAngle

a2da_map
[[dihedraLen x 4]] AtomNdx's in dihedraNdx order

d2a_map
[[dihedraLen x [4]]] AtomNdx's for each dihedron (reshaped a2da_map)

Numpy arrays for vector processing of chain di/hedra:
hedraL12: numpy array
bond length between hedron 1st and 2nd atom

hedraAngle: numpy array
bond angle for each hedron, in degrees

hedraL23: numpy array
bond length between hedron 2nd and 3rd atom

id3_dh_index: dict
maps hedron key to list of dihedra starting with hedron, used by assemble and bond_rotate to find dihedra with h1 key

id32_dh_index: dict
like id3_dh_index, find dihedra from h2 key

hAtoms: numpy array
homogeneous atom coordinates (3x4) of hedra, central atom at origin

hAtomsR: numpy array
hAtoms in reverse orientation

hAtoms_needs_update: numpy array of bool
indicates whether hAtoms represent hedraL12/A/L23

dihedraAngle: numpy array
dihedral angles (degrees) for each dihedron

dAtoms: numpy array
homogeneous atom coordinates (4x4) of dihedra, second atom at origin

dAtoms_needs_update: numpy array of bool
indicates whether dAtoms represent dihedraAngle

dCoordSpace: numpy array
forward and reverse transform matrices standardising positions of first hedron. See *dCoordSpace*.

dcsValid: bool
indicates dCoordSpace up to date

See also attributes generated by :meth:`build_edraArrays` for indexing di/hedra data elements.

Methods

internal_to_atom_coor	Process ic data to Residue/Atom coordinates; calls assemble_residues()
assemble_residues:	Generate IC_Chain atom coords from internal coordinates (parallel)
assemble_residues_ser:	Generate IC_Residue atom coords from internal coordinates (serial)
atom_to_internal_	Calculate dihedrals, angles, bond lengths (internal coordinates) for Atom data
write_SCAD:	Write OpenSCAD matrices for internal coordinate data comprising chain; this is a support routine, see *SCADIO.write_SCAD()* to generate OpenSCAD description of a protein chain.
distance_plot:	Generate 2D plot of interatomic distances with optional filter
distance_to_internal	Compute internal coordinates from distance plot and array of dihedral angle signs.
make_extended:	Arbitrarily sets all psi and phi backbone angles to 123 and -104 degrees.

MaxPeptideBond = 1.4

Larger C-N distance than this will be chain break

ParallelAssembleResidues = True

Enable parallel internal_to_atom algorithm, is slower for short chains

AAsiz = 0

Number of atoms in this chain (size of atomArray)

atomArray: array = None

AAsiz x [4] of float np.float64 homogeneous atom coordinates, all atoms in chain.

dCoordSpace = None

[2][dihedraLen][4][4] : 2 arrays of 4x4 coordinate space transforms for each dihedron. The first [0] converts TO standard space with first atom on the XZ plane, the second atom at the origin, the third on the +Z axis, and the fourth placed according to the dihedral angle. The second [1] transform returns FROM the standard space to world coordinates (PDB file input or whatever is current). Also accessible as .cst (forward transform) and .rcst (reverse transform) in *Dihedron*.

dcsValid = None

True if dCoordSpace is up to date. Use *update_dCoordSpace()* if needed.

__init__(*parent*, *verbose: bool = False*) → None

Initialize IC_Chain object, with or without residue/Atom data.

Parameters
parent (*Bio.PDB.Chain*) – Biopython Chain object Chain object this extends

__deepcopy__(*memo*) → *IC_Chain*

Implement deepcopy for IC_Chain.

clear_ic()

Clear residue internal_coord settings for this chain.

build_atomArray() → None

Build *IC_Chain* numpy coordinate array from biopython atoms.

See also *init_edra()* for more complete initialization of IC_Chain.

Inputs:

self.akset
[set] *AtomKey* s in this chain

Generates:

self.AAsiz
[int] number of atoms in chain (len(akset))

self.aktuple
[AAsiz x AtomKeys] sorted akset AtomKeys

self.atomArrayIndex
[[AAsiz] of int] numerical index for each AtomKey in aktuple

self.atomArrayValid
[AAsiz x bool] atomArray coordinates current with internal coordinates if True

self.atomArray
[AAsiz x np.float64[4]] homogeneous atom coordinates; Biopython *Atom* coordinates are view into this array after execution

rak_cache
[dict] lookup cache for AtomKeys for each residue

build_edraArrays() → None

Build chain level hedra and dihedra arrays.

Used by *init_edra()* and **_hedraDict2chain()**. Should be private method but exposed for documentation.

Inputs:

self.dihedraLen
[int] number of dihedra needed

self.hedraLen
[int] number of hedra needed

self.AAsiz
[int] length of atomArray

self.hedraNdx
[dict] maps hedron keys to range(hedraLen)

self.dihedraNdx
[dict] maps dihedron keys to range(dihedraLen)

self.hedra
[dict] maps Hedra keys to Hedra for chain

self.atomArray
[AAsiz x np.float64[4]] homogeneous atom coordinates for chain

self.atomArrayIndex
[dict] maps AtomKeys to atomArray

self.atomArrayValid
[AAsiz x bool] indicates coord is up-to-date

Generates:

self.dCoordSpace
[[2][dihedraLen][4][4]] transforms to/from dihedron coordinate space

self.dcsValid
[dihedraLen x bool] indicates dCoordSpace is current

self.hAtoms
[hedraLen x 3 x np.float64[4]] atom coordinates in hCoordSpace

self.hAtomsR
[hedraLen x 3 x np.float64[4]] hAtoms in reverse order (trading space for time)

self.hAtoms_needs_update
[hedraLen x bool] indicates hAtoms, hAtoms current

self.a2h_map
[AAsiz x [int ...]] maps atomArrayIndex to hedraNdx's with that atom

self.a2ha_map
[[hedraLen x 3]] AtomNdx's in hedraNdx order

self.h2aa
[hedraLen x [int ...]] maps hedraNdx to atomNdx's in hedron (reshaped later)

Hedron.ndx
[int] self.hedraNdx value stored inside Hedron object

self.dRev
[dihedraLen x bool] dihedron reversed if true

self.dH1ndx, dH2ndx
[[dihedraLen]] hedraNdx's for 1st and 2nd hedra

self.h1d_map
[hedraLen x []] hedraNdx -> [dihedra using hedron]

Dihedron.h1key, h2key
[[AtomKey ...]] hedron keys for dihedron, reversed as needed

Dihedron.hedron1, hedron2
[Hedron] references inside dihedron to hedra

Dihedron.ndx
[int] self.dihedraNdx info inside Dihedron object

Dihedron.cst, rcst
[np.float64p4][4]] dCoordSpace references inside Dihedron

self.a2da_map
[[dihedraLen x 4]] AtomNdx's in dihedraNdx order

self.d2a_map
[[dihedraLen x [4]]] AtomNdx's for each dihedron (reshaped a2da_map)

self.dFwd
[bool] dihedron is not Reversed if True

self.a2d_map
[AAsiz x [[dihedraNdx]] [atom ndx 0-3 of atom in dihedron]], maps atom indexes to dihedra and atoms in them

self.dAtoms_needs_update
[dihedraLen x bool] atoms in h1, h2 are current if False

assemble_residues(*verbose: bool = False*) → None

Generate atom coords from internal coords (vectorised).

This is the 'Numpy parallel' version of *assemble_residues_ser()*.

Starting with dihedra already formed by *init_atom_coords()*, transform each from dihedron local coordinate space into protein chain coordinate space. Iterate until all dependencies satisfied.

Does not update *dCoordSpace* as *assemble_residues_ser()* does. Call *update_dCoordSpace()* if needed. Faster to do in single operation once all atom coordinates finished.

Parameters
verbose (*bool*) – default False. Report number of iterations to compute changed dihedra

generates:

self.dSet: AAsiz x dihedraLen x 4
maps atoms in dihedra to atomArray

self.dSetValid
[[dihedraLen][4] of bool] map of valid atoms into dihedra to detect 3 or 4 atoms valid

Output coordinates written to *atomArray*. Biopython *Bio.PDB.Atom* coordinates are a view on this data.

assemble_residues_ser(*verbose: bool = False*, *start: int | None = None*, *fin: int | None = None*) → None

Generate IC_Residue atom coords from internal coordinates (serial).

See *assemble_residues()* for 'numpy parallel' version.

Filter positions between start and fin if set, find appropriate start coordinates for each residue and pass to *assemble()*

Parameters

- **verbose** (*bool*) – default False. Describe runtime problems
- **start,fin** (*int*) – default None. Sequence position for begin, end of subregion to generate coords for.

init_edra(*verbose: bool = False*) → None

Create chain and residue di/hedra structures, arrays, atomArray.

Inputs:
self.ordered_aa_ic_list : list of IC_Residue

Generates:

- edra objects, self.di/hedra (executes `_create_edra()`)
- atomArray and support (executes *build_atomArray()*)

- self.hedraLen : number of hedra in structure
- hedraL12 : numpy arrays for lengths, angles (empty)
- hedraAngle ..
- hedraL23 ..
- self.hedraNdx : dict mapping hedrakeys to hedraL12 etc
- self.dihedraLen : number of dihedra in structure
- dihedraAngle ..
- dihedraAngleRads : np arrays for angles (empty)
- self.dihedraNdx : dict mapping dihedrakeys to dihedraAngle

init_atom_coords() → None

Set chain level di/hedra initial coords from angles and distances.

Initializes atom coordinates in local coordinate space for hedra and dihedra, will be transformed appropriately later by *dCoordSpace* matrices for assembly.

update_dCoordSpace(*workSelector: ndarray | None = None*) → None

Compute/update coordinate space transforms for chain dihedra.

Requires all atoms updated so calls *assemble_residues()* (returns immediately if all atoms already assembled).

Parameters

workSelector (*[bool]*) – Optional mask to select dihedra for update

propagate_changes() → None

Track through di/hedra to invalidate dependent atoms.

internal_to_atom_coordinates(*verbose: bool = False*, *start: int | None = None*, *fin: int | None = None*) → None

Process IC data to Residue/Atom coords.

Parameters

- **verbose** (*bool*) – default False. Describe runtime problems
- **start,fin** (*int*) – Optional sequence positions for begin, end of subregion to process.

Note: Setting start or fin activates serial *assemble_residues_ser()* instead of (Numpy parallel) *assemble_residues()*. Start C-alpha will be at origin.

See also:

ParallelAssembleResidues

atom_to_internal_coordinates(*verbose: bool = False*) → None

Calculate dihedrals, angles, bond lengths for Atom data.

Generates atomArray (through init_edra), value arrays for hedra and dihedra, and coordinate space transforms for dihedra.

Generates Gly C-beta if specified, see *IC_Residue.gly_Cbeta*

Parameters

verbose (*bool*) – default False. describe runtime problems

distance_plot(*filter: ndarray | None = None*) → ndarray

Generate 2D distance plot from atomArray.

Default is to calculate distances for all atoms. To generate the classic C-alpha distance plot, pass a boolean mask array like:

```
atmNameNdx = internal_coords.AtomKey.fields.atm
CaSelect = [
    atomArrayIndex.get(k)
    for k in atomArrayIndex.keys()
    if k.akl[atmNameNdx] == "CA"
]
plot = cic.distance_plot(CaSelect)
```

Alternatively, this will select all backbone atoms:

```
backboneSelect = [
    atomArrayIndex.get(k)
    for k in atomArrayIndex.keys()
    if k.is_backbone()
]
```

Parameters

filter (*[bool]*) – restrict atoms for calculation

See also:

distance_to_internal_coordinates(), which requires the default all atom distance plot.

dihedral_signs() → ndarray

Get sign array (+1/-1) for each element of chain dihedraAngle array.

Required for *distance_to_internal_coordinates()*

distplot_to_dh_arrays(*distplot: ndarray*, *dihedra_signs: ndarray*) → None

Load di/hedra distance arrays from distplot.

Fill *IC_Chain* arrays hedraL12, L23, L13 and dihedraL14 distance value arrays from input distplot, dihedra_signs array from input dihedra_signs. Distplot and di/hedra distance arrays must index according to AtomKey mappings in *IC_Chain* .hedraNdx and .dihedraNdx (created in *IC_Chain.init_edra()*)

Call *atom_to_internal_coordinates()* (or at least *init_edra()*) to generate a2ha_map and d2a_map before running this.

Explicitly removed from *distance_to_internal_coordinates()* so user may populate these chain di/hedra arrays by other methods.

distance_to_internal_coordinates(*resetAtoms: bool | None = True*) → None

Compute chain di/hedra from from distance and chirality data.

Distance properties on hedra L12, L23, L13 and dihedra L14 configured by *distplot_to_dh_arrays()* or alternative loader.

dihedraAngles result is multiplied by dihedra_signs at final step recover chirality information lost in distance plot (mirror image of structure has same distances but opposite sign dihedral angles).

Note that chain breaks will cause errors in rebuilt structure, use *copy_initNCaCs()* to avoid this

Based on Blue, the Hedronometer's answer to The dihedral angles of a tetrahedron in terms of its edge lengths on math.stackexchange.com. See also: "Heron-like Hedronometric Results for Tetrahedral Volume".

Other values from that analysis included here as comments for completeness:

- oa = hedron1 L12 if reverse else hedron1 L23
- ob = hedron1 L23 if reverse else hedron1 L12
- ac = hedron2 L12 if reverse else hedron2 L23
- ab = hedron1 L13 = law of cosines on OA, OB (hedron1 L12, L23)
- oc = hedron2 L13 = law of cosines on OA, AC (hedron2 L12, L23)
- bc = dihedron L14

target is OA, the dihedral angle along edge oa.

Parameters

resetAtoms (*bool*) – default True. Mark all atoms in di/hedra and atomArray for updating by *internal_to_atom_coordinates()*. Alternatively set this to False and manipulate *atomArrayValid*, *dAtoms_needs_update* and *hAtoms_needs_update* directly to reduce computation.

copy_initNCaCs(*other:* IC_Chain) → None

Copy atom coordinates for initNCaC atoms from other IC_Chain.

Copies the coordinates and sets atomArrayValid flags True for initial NCaC and after any chain breaks.

Needed for *distance_to_internal_coordinates()* if target has chain breaks (otherwise each fragment will start at origin).

Also useful if copying internal coordinates from another chain.

N.B. *IC_Residue.set_angle()* and *IC_Residue.set_length()* invalidate their relevant atoms, so apply them before calling this function.

make_extended()

Set all psi and phi angles to extended conformation (123, -104).

__annotations__ = {'atomArray': <built-in function array>, 'atomArrayIndex': "dict['AtomKey', int]", 'bpAtomArray': "list['Atom']", 'ordered_aa_ic_list': 'list[IC_Residue]'}

class Bio.PDB.internal_coords.**IC_Residue**(*parent:* Residue)

Bases: object

Class to extend Biopython Residue with internal coordinate data.

Parameters

parent: biopython Residue object this class extends

Attributes

no_altloc: bool default False
Class variable, disable processing of ALTLOC atoms if True, use only selected atoms.

accept_atoms: tuple
Class variable *accept_atoms*, list of PDB atom names to use when generating internal coordinates. Default is:

```
accept_atoms = accept_mainchain + accept_hydrogens
```

to exclude hydrogens in internal coordinates and generated PDB files, override as:

```
IC_Residue.accept_atoms = IC_Residue.accept_mainchain
```

to get only mainchain atoms plus amide proton, use:

```
IC_Residue.accept_atoms = IC_Residue.accept_mainchain + ('H',)
```

to convert D atoms to H, set *AtomKey.d2h* = True and use:

```
IC_Residue.accept_atoms = (
    accept_mainchain + accept_hydrogens + accept_deuteriums
)
```

Note that *accept_mainchain* = *accept_backbone* + *accept_sidechain*. Thus to generate sequence-agnostic conformational data for e.g. structure alignment in dihedral angle space, use:

```
IC_Residue.accept_atoms = accept_backbone
```

or set gly_Cbeta = True and use:

```
IC_Residue.accept_atoms = accept_backbone + ('CB',)
```

Changing accept_atoms will cause the default *structure_rebuild_test* in *ic_rebuild* to fail if some atoms are filtered (obviously). Use the *quick=True* option to test only the coordinates of filtered atoms to avoid this.

There is currently no option to output internal coordinates with D instead of H.

accept_resnames: tuple
Class variable *accept_resnames*, list of 3-letter residue names for HETATMs to accept when generating internal coordinates from atoms. HETATM sidechain will be ignored, but normal backbone atoms (N, CA, C, O, CB) will be included. Currently only CYG, YCM and UNK; override at your own risk. To generate sidechain, add appropriate entries to *ic_data_sidechains* in *ic_data* and support in *IC_Chain.atom_to_internal_coordinates()*.

gly_Cbeta: bool default False
Class variable *gly_Cbeta*, override to True to generate internal coordinates for glycine CB atoms in *IC_Chain.atom_to_internal_coordinates()*

```
IC_Residue.gly_Cbeta = True
```

pic_accuracy: str default "17.13f"
Class variable *pic_accuracy* sets accuracy for numeric values (angles, lengths) in .pic files. Default set high to support mmCIF file accuracy in rebuild tests. If you find rebuild tests fail with 'ERROR -COORDINATES-' and verbose=True shows only small discrepancies, try raising this value (or lower it to 9.5 if only working with PDB format files).

```
IC_Residue.pic_accuracy = "9.5f"
```

residue: Biopython Residue object reference
The *Residue* object this extends

hedra: dict indexed by 3-tuples of AtomKeys
Hedra forming this residue

dihedra: dict indexed by 4-tuples of AtomKeys
Dihedra forming (overlapping) this residue

rprev, rnext: lists of IC_Residue objects
References to adjacent (bonded, not missing, possibly disordered) residues in chain

atom_coords: AtomKey indexed dict of numpy [4] arrays
removed Use AtomKeys and atomArrayIndex to build if needed

ak_set: set of AtomKeys in dihedra
AtomKeys in all dihedra overlapping this residue (see __contains__())

alt_ids: list of char
AltLoc IDs from PDB file

bfactors: dict
AtomKey indexed B-factors as read from PDB file

NCaCKey: List of tuples of AtomKeys
List of tuples of N, Ca, C backbone atom AtomKeys; usually only 1 but more if backbone altlocs.

is20AA: bool
True if residue is one of 20 standard amino acids, based on Residue resname

isAccept: bool
True if is20AA or in accept_resnames below

rbase: tuple
residue position, insert code or none, resname (1 letter if standard amino acid)

cic: IC_Chain default None
parent chain *IC_Chain* object

scale: optional float
used for OpenSCAD output to generate gly_Cbeta bond length

Methods

assemble(atomCoordsIn, resetLocation, verbose)	Compute atom coordinates for this residue from internal coordinates
get_angle()	Return angle for passed key
get_length()	Return bond length for specified pair
pick_angle()	Find Hedron or Dihedron for passed key
pick_length()	Find hedra for passed AtomKey pair
set_angle()	Set angle for passed key (no position updates)
set_length()	Set bond length in all relevant hedra for specified pair
bond_rotate(delta)	adjusts related dihedra angles by delta, e.g. rotating psi (N-Ca-C-N) will adjust the adjacent N-Ca-C-O by the same amount to avoid clashes
bond_set(angle)	uses bond_rotate to set specified dihedral to angle and adjust related dihedra accordingly
rak(atom info)	cached AtomKeys for this residue

accept_resnames = ('CYG', 'YCM', 'UNK')

Add 3-letter residue name here for non-standard residues with normal backbone. CYG included for test case 4LGY (1305 residue contiguous chain). Safe to add more names for N-CA-C-O backbones, any more complexity will need additions to *accept_atoms*, *ic_data_sidechains* in *ic_data* and support in *IC_Chain.atom_to_internal_coordinates()*

no_altloc: bool = False

Set True to filter altloc atoms on input and only work with Biopython default Atoms

gly_Cbeta: bool = False

Create beta carbons on all Gly residues.

Setting this to True will generate internal coordinates for Gly C-beta carbons in `atom_to_internal_coordinates()`.

Data averaged from Sep 2019 Dunbrack cullpdb_pc20_res2.2_R1.0 restricted to structures with amide protons. Please see

PISCES: A Protein Sequence Culling Server

'G. Wang and R. L. Dunbrack, Jr. PISCES: a protein sequence culling server. Bioinformatics, 19:1589-1591, 2003.'

Ala avg rotation of OCCACB from NCACO query:

```
select avg(g.rslt) as avg_rslt, stddev(g.rslt) as sd_rslt, count(*)
from
(select f.d1d, f.d2d,
(case when f.rslt > 0 then f.rslt-360.0 else f.rslt end) as rslt
from (select d1.angle as d1d, d2.angle as d2d,
(d2.angle - d1.angle) as rslt from dihedron d1,
dihedron d2 where d1.re_class='AOACACAACB' and
d2.re_class='ANACAACAO' and d1.pdb=d2.pdb and d1.chn=d2.chn
and d1.res=d2.res) as f) as g
```

results:

```
| avg_rslt          | sd_rslt          | count   |
| -122.682194862932 | 5.044030405139l9 | 14098   |
```

pic_accuracy: str = '17.13f'

accept_backbone = ('N', 'CA', 'C', 'O', 'OXT')

accept_sidechain = ('CB', 'CG', 'CG1', 'OG1', 'OG', 'SG', 'CG2', 'CD', 'CD1', 'SD', 'OD1', 'ND1', 'CD2', 'ND2', 'CE', 'CE1', 'NE', 'OE1', 'NE1', 'CE2', 'OE2', 'NE2', 'CE3', 'CZ', 'NZ', 'CZ2', 'CZ3', 'OD2', 'OH', 'CH2', 'NH1', 'NH2')

accept_mainchain = ('N', 'CA', 'C', 'O', 'OXT', 'CB', 'CG', 'CG1', 'OG1', 'OG', 'SG', 'CG2', 'CD', 'CD1', 'SD', 'OD1', 'ND1', 'CD2', 'ND2', 'CE', 'CE1', 'NE', 'OE1', 'NE1', 'CE2', 'OE2', 'NE2', 'CE3', 'CZ', 'NZ', 'CZ2', 'CZ3', 'OD2', 'OH', 'CH2', 'NH1', 'NH2')

accept_hydrogens = ('H', 'H1', 'H2', 'H3', 'HA', 'HA2', 'HA3', 'HB', 'HB1', 'HB2', 'HB3', 'HG2', 'HG3', 'HD2', 'HD3', 'HE2', 'HE3', 'HZ1', 'HZ2', 'HZ3', 'HG11', 'HG12', 'HG13', 'HG21', 'HG22', 'HG23', 'HZ', 'HD1', 'HE1', 'HD11', 'HD12', 'HD13', 'HG', 'HG1', 'HD21', 'HD22', 'HD23', 'NH1', 'NH2', 'HE', 'HH11', 'HH12', 'HH21', 'HH22', 'HE21', 'HE22', 'HE2', 'HH', 'HH2')

accept_deuteriums = ('D', 'D1', 'D2', 'D3', 'DA', 'DA2', 'DA3', 'DB', 'DB1', 'DB2', 'DB3', 'DG2', 'DG3', 'DD2', 'DD3', 'DE2', 'DE3', 'DZ1', 'DZ2', 'DZ3', 'DG11', 'DG12', 'DG13', 'DG21', 'DG22', 'DG23', 'DZ', 'DD1', 'DE1', 'DD11', 'DD12', 'DD13', 'DG', 'DG1', 'DD21', 'DD22', 'DD23', 'ND1', 'ND2', 'DE', 'DH11', 'DH12', 'DH21', 'DH22', 'DE21', 'DE22', 'DE2', 'DH', 'DH2')

accept_atoms = ('N', 'CA', 'C', 'O', 'OXT', 'CB', 'CG', 'CG1', 'OG1', 'OG', 'SG', 'CG2', 'CD', 'CD1', 'SD', 'OD1', 'ND1', 'CD2', 'ND2', 'CE', 'CE1', 'NE', 'OE1', 'NE1', 'CE2', 'OE2', 'NE2', 'CE3', 'CZ', 'NZ', 'CZ2', 'CZ3', 'OD2', 'OH', 'CH2', 'NH1', 'NH2', 'H', 'H1', 'H2', 'H3', 'HA', 'HA2', 'HA3', 'HB', 'HB1', 'HB2', 'HB3', 'HG2', 'HG3', 'HD2', 'HD3', 'HE2', 'HE3', 'HZ1', 'HZ2', 'HZ3', 'HG11', 'HG12', 'HG13', 'HG21', 'HG22', 'HG23', 'HZ', 'HD1', 'HE1', 'HD11', 'HD12', 'HD13', 'HG', 'HG1', 'HD21', 'HD22', 'HD23', 'NH1', 'NH2', 'HE', 'HH11', 'HH12', 'HH21', 'HH22', 'HE21', 'HE22', 'HE2', 'HH', 'HH2')

Change accept_atoms to restrict atoms processed. See *IC_Residue* for usage.

__init__(*parent:* Residue) → None

Initialize IC_Residue with parent Biopython Residue.

Parameters
parent (Residue) – Biopython Residue object. The Biopython Residue this object extends

__deepcopy__(*memo*)

Deep copy implementation for IC_Residue.

__contains__(*ak:* AtomKey) → bool

Return True if atomkey is in this residue.

rak(*atm: str* | Atom) → *AtomKey*

Cache calls to AtomKey for this residue.

__repr__() → str

Print string is parent Residue ID.

pretty_str() → str

Nice string for residue ID.

set_flexible() → None

For OpenSCAD, mark N-CA and CA-C bonds to be flexible joints.

See *SCADIO.write_SCAD()*

set_hbond() → None

For OpenSCAD, mark H-N and C-O bonds to be hbonds (magnets).

See *SCADIO.write_SCAD()*

clear_transforms()

Invalidate dihedra coordinate space attributes before assemble().

Coordinate space attributes are Dihedron.cst and .rcst, and *IC_Chain.dCoordSpace*

assemble(*resetLocation: bool = False*, *verbose: bool = False*) → dict[*AtomKey*, array] | dict[tuple[*AtomKey*, *AtomKey*, *AtomKey*], array] | None

Compute atom coordinates for this residue from internal coordinates.

This is the IC_Residue part of the *assemble_residues_ser()* serial version, see *assemble_residues()* for numpy vectorized approach which works at the *IC_Chain* level.

Join prepared dihedra starting from N-CA-C and N-CA-CB hedrons, computing protein space coordinates for backbone and sidechain atoms

Sets forward and reverse transforms on each Dihedron to convert from protein coordinates to dihedron space coordinates for first three atoms (see *IC_Chain.dCoordSpace*)

Call *init_atom_coords()* to update any modified di/hedra before coming here, this only assembles dihedra into protein coordinate space.

Algorithm

Form double-ended queue, start with c-ca-n, o-c-ca, n-ca-cb, n-ca-c.

if resetLocation=True, use initial coords from generating dihedron for n-ca-c initial positions (result in dihedron coordinate space)

while queue not empty
get 3-atom hedron key

for each dihedron starting with hedron key (1st hedron of dihedron)

if have coordinates for all 4 atoms already
add 2nd hedron key to back of queue

else if have coordinates for 1st 3 atoms
compute forward and reverse transforms to take 1st 3 atoms to/from dihedron initial coordinate space

use reverse transform to get position of 4th atom in current coordinates from dihedron initial coordinates

add 2nd hedron key to back of queue

else
ordering failed, put hedron key at back of queue and hope next time we have 1st 3 atom positions (should not happen)

loop terminates (queue drains) as hedron keys which do not start any dihedra are removed without action

Parameters
resetLocation (*bool*) – default False. - Option to ignore start location and orient so initial N-Ca-C hedron at origin.

Returns

Dict of AtomKey -> homogeneous atom coords for residue in protein space relative to previous residue

Also directly updates *IC_Chain.atomArray* as *assemble_residues()* does.

split_akl(*lst: tuple[*AtomKey*, ...] | list[*AtomKey*], missingOK: bool = False*) → list[tuple[*AtomKey*, ...]]

Get AtomKeys for this residue (ak_set) for generic list of AtomKeys.

Changes and/or expands a list of 'generic' AtomKeys (e.g. 'N, C, C') to be specific to this Residue's altlocs etc., e.g. '(N-Ca_A_0.3-C, N-Ca_B_0.7-C)'

Given a list of AtomKeys for a Hedron or Dihedron,

return:

list of matching atomkeys that have id3_dh in this residue (ak may change if occupancy != 1.00)

or
multiple lists of matching atomkeys expanded for all atom altlocs

or
empty list if any of atom_coord(ak) missing and not missingOK

Parameters

- **lst** (*list*) – list[3] or [4] of AtomKeys. Non-altloc AtomKeys to match to specific AtomKeys for this residue
- **missingOK** (*bool*) – default False, see above.

atom_sernum = None

atom_chain = None

pdb_residue_string() → str

Generate PDB ATOM records for this residue as string.

Convenience method for functionality not exposed in PDBIO.py. Increments *IC_Residue.atom_sernum* if not None

Parameters

- **IC_Residue.atom_sernum** – Class variable default None. Override and increment atom serial number if not None
- **IC_Residue.atom_chain** – Class variable. Override atom chain id if not None

---

Todo: move to PDBIO

---

pic_flags = (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 496, 525, 1021, 1024, 2048, 4096, 7168, 8192, 16384)

Used by *PICIO.write_PIC()* to control classes of values to be defaulted.

picFlagsDefault = 31744

Default is all dihedra + initial tau atoms + bFactors.

picFlagsDict = {'all': 7168, 'bFactors': 16384, 'chi': 496, 'chi1': 16, 'chi2': 32, 'chi3': 64, 'chi4': 128, 'chi5': 256, 'classic': 1021, 'classic_b': 525, 'hedra': 1024, 'initAtoms': 8192, 'omg': 2, 'phi': 4, 'pomg': 512, 'primary': 2048, 'psi': 1, 'secondary': 4096, 'tau': 8}

Dictionary of pic_flags values to use as needed.

pick_angle(*angle_key: tuple[*AtomKey, AtomKey, AtomKey*] | tuple[*AtomKey, AtomKey, AtomKey, AtomKey*] | str*) → *Hedron* | *Dihedron* | None

Get Hedron or Dihedron for angle_key.

Parameters
angle_key –

- tuple of 3 or 4 AtomKeys
- string of atom names ('CA') separated by :'s
- string of [-1, 0, 1]<atom name> separated by ':'s. -1 is previous residue, 0 is this residue, 1 is next residue
- psi, phi, omg, omega, chi1, chi2, chi3, chi4, chi5
- tau (N-CA-C angle) see Richardson1981

- tuples of AtomKeys is only access for alternate disordered atoms

Observe that a residue's phi and omega dihedrals, as well as the hedra comprising them (including the N:Ca:C *tau* hedron), are stored in the n-1 di/hedra sets; this overlap is handled here, but may be an issue if accessing directly.

The following print commands are equivalent (except for sidechains with non-carbon atoms for chi2):

```
ric = r.internal_coord
print(
    r,
    ric.get_angle("psi"),
    ric.get_angle("phi"),
    ric.get_angle("omg"),
    ric.get_angle("tau"),
    ric.get_angle("chi2"),
)
print(
    r,
    ric.get_angle("N:CA:C:1N"),
    ric.get_angle("-1C:N:CA:C"),
    ric.get_angle("-1CA:-1C:N:CA"),
    ric.get_angle("N:CA:C"),
    ric.get_angle("CA:CB:CG:CD"),
)
```

See ic_data.py for detail of atoms in the enumerated sidechain angles and the backbone angles which do not span the peptide bond. Using 's' for current residue ('self') and 'n' for next residue, the spanning (overlapping) angles are:

```
(sN, sCA, sC, nN)   # psi
(sCA, sC, nN, nCA)  # omega i+1
(sC, nN, nCA, nC)   # phi i+1
(sCA, sC, nN)
(sC, nN, nCA)
(nN, nCA, nC)       # tau i+1
```

Returns
Matching Hedron, Dihedron, or None.

get_angle(*angle_key: tuple[*AtomKey, AtomKey, AtomKey*] | tuple[*AtomKey, AtomKey, AtomKey, AtomKey*] | str*) → float | None

Get dihedron or hedron angle for specified key.

See *pick_angle()* for key specifications.

set_angle(*angle_key: tuple[*AtomKey, AtomKey, AtomKey*] | tuple[*AtomKey, AtomKey, AtomKey, AtomKey*] | str*, *v: float*, *overlap=True*)

Set dihedron or hedron angle for specified key.

If angle is a *Dihedron* and *overlap* is True (default), overlapping dihedra are also changed as appropriate. The overlap is a result of protein chain definitions in *ic_data* and `_create_edra()` (e.g. psi overlaps N-CA-C-O).

The default overlap=True is probably what you want for: *set_angle("chi1", val)*

The default is probably NOT what you want when processing all dihedrals in a chain or residue (such as copying from another structure), as the overlapping dihedra will likely be in the set as well.

N.B. setting e.g. PRO chi2 is permitted without error or warning!

See *pick_angle()* for angle_key specifications. See *bond_rotate()* to change a dihedral by a number of degrees

Parameters

- **angle_key** – angle identifier.
- **v** (*float*) – new angle in degrees (result adjusted to +/-180).
- **overlap** (*bool*) – default True. Modify overlapping dihedra as needed

bond_rotate(*angle_key: tuple[*AtomKey, AtomKey, AtomKey*] | tuple[*AtomKey, AtomKey, AtomKey, AtomKey*] | str*, *delta: float*)

Rotate set of overlapping dihedrals by delta degrees.

Changes a dihedral angle by a given delta, i.e. new_angle = current_angle + delta Values are adjusted so new_angle will be within +/-180.

Changes overlapping dihedra as in *set_angle()*

See *pick_angle()* for key specifications.

bond_set(*angle_key: tuple[*AtomKey, AtomKey, AtomKey*] | tuple[*AtomKey, AtomKey, AtomKey, AtomKey*] | str*, *val: float*)

Set dihedron to val, update overlapping dihedra by same amount.

Redundant to *set_angle()*, retained for compatibility. Unlike *set_angle()* this is for dihedra only and no option to not update overlapping dihedra.

See *pick_angle()* for key specifications.

pick_length(*ak_spec: str | tuple[*AtomKey, AtomKey*]*) → tuple[list[*Hedron*] | None, tuple[*AtomKey*, *AtomKey*] | None]

Get list of hedra containing specified atom pair.

Parameters

ak_spec –

- tuple of two AtomKeys
- string: two atom names separated by ':', e.g. 'N:CA' with optional position specifier relative to self, e.g. '-1C:N' for preceding peptide bond. Position specifiers are -1, 0, 1.

The following are equivalent:

```
ric = r.internal_coord
print(
    r,
    ric.get_length("0C:1N"),
)
print(
    r,
    None
    if not ric.rnext
    else ric.get_length((ric.rak("C"), ric.rnext[0].rak("N"))),
)
```

If atom not found on current residue then will look on rprev[0] to handle cases like Gly N:CA. For finer control please access *IC_Chain.hedra* directly.

Returns
list of hedra containing specified atom pair as tuples of AtomKeys

get_length(*ak_spec: str | tuple[*AtomKey, AtomKey*]*) → float | None

Get bond length for specified atom pair.

See *pick_length()* for ak_spec and details.

set_length(*ak_spec: str | tuple[*AtomKey, AtomKey*]*, *val: float*) → None

Set bond length for specified atom pair.

See *pick_length()* for ak_spec.

applyMtx(*mtx: array*) → None

Apply matrix to atom_coords for this IC_Residue.

```
__annotations__ = {'_AllBonds': <class 'bool'>, 'ak_set': 'set[AtomKey]', 'akc':
'dict[Union[str, Atom], AtomKey]', 'alt_ids': 'Union[list[str], None]', 'bfactors':
'dict[str, float]', 'cic': 'IC_Chain', 'dihedra': 'dict[DKT, Dihedron]',
'gly_Cbeta': <class 'bool'>, 'hedra': 'dict[HKT, Hedron]', 'no_altloc': <class
'bool'>, 'pic_accuracy': <class 'str'>, 'rnext': 'list[IC_Residue]', 'rprev':
'list[IC_Residue]'}
```

class Bio.PDB.internal_coords.**Edron**(**args: list[*AtomKey*] | tuple[*AtomKey, AtomKey, AtomKey*] | tuple[*AtomKey, AtomKey, AtomKey, AtomKey*], **kwargs: str*)

Bases: `object`

Base class for Hedron and Dihedron classes.

Supports rich comparison based on lists of AtomKeys.

Attributes

atomkeys: tuple
3 (hedron) or 4 (dihedron) *AtomKey* s defining this di/hedron

id: str
':'-joined string of AtomKeys for this di/hedron

needs_update: bool
indicates di/hedron local atom_coords do NOT reflect current di/hedron angle and length values in hedron local coordinate space

e_class: str
sequence of atoms (no position or residue) comprising di/hedron for statistics

re_class: str
sequence of residue, atoms comprising di/hedron for statistics

cre_class: str
sequence of covalent radii classes comprising di/hedron for statistics

edron_re: compiled regex (Class Attribute)
A compiled regular expression matching string IDs for Hedron and Dihedron objects

cic: IC_Chain reference
Chain internal coords object containing this hedron

ndx: int
index into IC_Chain level numpy data arrays for di/hedra. Set in *IC_Chain.init_edra()*

rc: int
number of residues involved in this edron

Methods

gen_key([AtomKey, ...] or AtomKey, ...) (Static Method)	generate a ':'-joined string of AtomKey Ids
is_backbone()	Return True if all atomkeys atoms are N, Ca, C or O

edron_re = re.compile('^(?P<pdbid>\\w+)?\\s(?P<chn>[\\w|\\s])?\\s(?P<a1>[\\w\\-\\.]+):(?P<a2>[\\w\\-\\.]+):(?P<a3>[\\w\\-\\.]+)(:(?P<a4>[\\w\\-\\.]+))?\\s+(((?P<len12>\\S+)\\s+(?P<angle>\\S+)\\s+(?P<len23>\\S+)\\s*$)|((?P<)

A compiled regular expression matching string IDs for Hedron and Dihedron objects

static gen_key(*lst: list[*AtomKey*]*) → str

Generate string of ':'-joined AtomKey strings from input.

Generate '2_A_C:3_P_N:3_P_CA' from (2_A_C, 3_P_N, 3_P_CA) :param list lst: list of AtomKey objects

static gen_tuple(*akstr: str*) → tuple

Generate AtomKey tuple for ':'-joined AtomKey string.

Generate (2_A_C, 3_P_N, 3_P_CA) from '2_A_C:3_P_N:3_P_CA' :param str akstr: string of ':'-separated AtomKey strings

__init__(**args: list[*AtomKey*] | tuple[*AtomKey*, *AtomKey*, *AtomKey*] | tuple[*AtomKey*, *AtomKey*, *AtomKey*, *AtomKey*], **kwargs: str*) → None

Initialize Edron with sequence of AtomKeys.

Acceptable input:

[AtomKey, ...] : list of AtomKeys AtomKey, ... : sequence of AtomKeys as args {'a1': str, 'a2': str, ... } : dict of AtomKeys as 'a1', 'a2' ...

__deepcopy__(*memo*)

Deep copy implementation for Edron.

__contains__(*ak:* AtomKey) → bool

Return True if atomkey is in this edron.

is_backbone() → bool

Report True for contains only N, C, CA, O, H atoms.

__repr__() → str

Tuple of AtomKeys is default repr string.

__hash__() → int

Hash calculated at init from atomkeys tuple.

__eq__(*other: object*) → bool

Test for equality.

__ne__(*other: object*) → bool

Test for inequality.

__gt__(*other: object*) → bool
Test greater than.

__ge__(*other: object*) → bool
Test greater or equal.

__lt__(*other: object*) → bool
Test less than.

__le__(*other: object*) → bool
Test less or equal.

class Bio.PDB.internal_coords.**Hedron**(**args: list[AtomKey] | tuple[AtomKey, AtomKey, AtomKey]*, ***kwargs: str*)

Bases: *Edron*

Class to represent three joined atoms forming a plane.

Contains atom coordinates in local coordinate space: central atom at origin, one terminal atom on XZ plane, and the other on the +Z axis. Stored in two orientations, with the 3rd (forward) or first (reversed) atom on the +Z axis. See *Dihedron* for use of forward and reverse orientations.

Attributes

len12: float
distance between first and second atoms

len23: float
distance between second and third atoms

angle: float
angle (degrees) formed by three atoms in hedron

xrh_class: string
only for hedron spanning 2 residues, will have 'X' for residue contributing only one atom

Methods

get_length()	get bond length for specified atom pair
set_length()	set bond length for specified atom pair
angle(), len12(), len23()	setters for relevant attributes (angle in degrees)

__init__(**args: list[AtomKey] | tuple[AtomKey, AtomKey, AtomKey]*, ***kwargs: str*) → None
Initialize Hedron with sequence of AtomKeys, kwargs.

Acceptable input:
As for Edron, plus optional 'len12', 'angle', 'len23' keyworded values.

__repr__() → str
Print string for Hedron object.

property angle: float
Get this hedron angle.

property len12
Get first length for Hedron.

property len23: float

Get second length for Hedron.

get_length(*ak_tpl: tuple[*AtomKey, AtomKey*]*) → float | None

Get bond length for specified atom pair.

Parameters
ak_tpl (*tuple*) – tuple of AtomKeys. Pair of atoms in this Hedron

set_length(*ak_tpl: tuple[*AtomKey, AtomKey*]*, *newLength: float*)

Set bond length for specified atom pair; sets needs_update.

Parameters
.ak_tpl (*tuple*) – tuple of AtomKeys Pair of atoms in this Hedron

__annotations__ = {}

class Bio.PDB.internal_coords.**Dihedron**(**args: list[*AtomKey*]* | *tuple[*AtomKey, AtomKey, AtomKey, AtomKey*]*, ***kwargs: str*)

Bases: *Edron*

Class to represent four joined atoms forming a dihedral angle.

Attributes

angle: float
Measurement or specification of dihedral angle in degrees; prefer *IC_Residue.bond_set()* to set

hedron1, hedron2: Hedron object references
The two hedra which form the dihedral angle

h1key, h2key: tuples of AtomKeys
Hash keys for hedron1 and hedron2

id3,id32: tuples of AtomKeys
First 3 and second 3 atoms comprising dihedron; hxkey orders may differ

ric: IC_Residue object reference
IC_Residue object containing this dihedral

reverse: bool
Indicates order of atoms in dihedron is reversed from order of atoms in hedra

primary: bool
True if this is psi, phi, omega or a sidechain chi angle

pclass: string (primary angle class)
re_class with X for adjacent residue according to nomenclature (psi, omega, phi)

cst, rcst: numpy [4][4] arrays
transformations to (cst) and from (rcst) Dihedron coordinate space defined with atom 2 (Hedron 1 center atom) at the origin. Views on *IC_Chain.dCoordSpace*.

Methods

angle()	getter/setter for dihdral angle in degrees; prefer *IC_Residue.bond_set()*
bits()	return *IC_Residue.pic_flags* bitmask for dihedron psi, omega, etc

__init__(**args: list[*AtomKey*] | tuple[*AtomKey*, *AtomKey*, *AtomKey*, *AtomKey*], **kwargs: str*) → None

Init Dihedron with sequence of AtomKeys and optional dihedral angle.

Acceptable input:
As for Edron, plus optional 'dihedral' keyworded angle value.

__repr__() → str

Print string for Dihedron object.

property angle: float

Get dihedral angle.

static angle_dif(*a1: float | ndarray*, *a2: float | ndarray*)

Get angle difference between two +/- 180 angles.

https://stackoverflow.com/a/36001014/2783487

static angle_avg(*alst: list*, *in_rads: bool = False*, *out_rads: bool = False*)

Get average of list of +/-180 angles.

Parameters

- **alst** (*List*) – list of angles to average
- **in_rads** (*bool*) – input values are in radians
- **out_rads** (*bool*) – report result in radians

static angle_pop_sd(*alst: list*, *avg: float*)

Get population standard deviation for list of +/-180 angles.

should be sample std dev but avoid len(alst)=1 -> div by 0

difference(*other:* Dihedron) → float

Get angle difference between this and other +/- 180 angles.

bits() → int

Get *IC_Residue.pic_flags* bitmasks for self is psi, omg, phi, pomg, chiX.

__annotations__ = {}

class Bio.PDB.internal_coords.**AtomKey**(**args:* IC_Residue | Atom | *list* | *dict* | *str*, ***kwargs: str*)

Bases: object

Class for dict keys to reference atom coordinates.

AtomKeys capture residue and disorder information together, and provide a no-whitespace string key for .pic files.

Supports rich comparison and multiple ways to instantiate.

AtomKeys contain:
residue position (respos), insertion code (icode), 1 or 3 char residue name (resname), atom name (atm), altloc (altloc), and occupancy (occ)

Use *AtomKey.fields* to get the index to the component of interest by name:

Get C-alpha atoms from IC_Chain atomArray and atomArrayIndex with AtomKeys:

```
atmNameNdx = internal_coords.AtomKey.fields.atm
CaSelection = [
    atomArrayIndex.get(k)
    for k in atomArrayIndex.keys()
    if k.akl[atmNameNdx] == "CA"
]
AtomArrayCa = atomArray[CaSelection]
```

Get all phenylalanine atoms in a chain:

```
resNameNdx = internal_coords.AtomKey.fields.resname
PheSelection = [
    atomArrayIndex.get(k)
    for k in atomArrayIndex.keys()
    if k.akl[resNameNdx] == "F"
]
AtomArrayPhe = atomArray[PheSelection]
```

'resname' will be the uppercase 1-letter amino acid code if one of the 20 standard residues, otherwise the supplied 3-letter code. Supplied as input or read from .rbase attribute of *IC_Residue*.

Attributes

akl: tuple
All six fields of AtomKey

fieldNames: tuple (Class Attribute)
Mapping of key index positions to names

fields: namedtuple (Class Attribute)
Mapping of field names to index positions.

id: str
'_'-joined AtomKey fields, excluding 'None' fields

atom_re: compiled regex (Class Attribute)
A compiled regular expression matching the string form of the key

d2h: bool (Class Attribute) default False
Convert D atoms to H on input if True; must also modify *IC_Residue.accept_atoms*

missing: bool default False
AtomKey __init__'d from string is probably missing, set this flag to note the issue. Set by *IC_Residue.rak()*

ric: IC_Residue default None
If initialised with IC_Residue, this references the IC_residue

Methods

alt-loc_match(other)	Returns True if this AtomKey matches other AtomKey excluding altloc and occupancy fields
is_backbone()	Returns True if atom is N, CA, C, O or H
atm()	Returns atom name, e.g. N, CA, CB, etc.
cr_class()	Returns covalent radii class e.g. Csb

atom_re = re.compile('^(?P<respos>-?\\d+)(?P<icode>[A-Za-z])?_(?P<resname>[a-zA-Z]+)_(?P<atm>[A-Za-z0-9]+)(?:_(?P<altloc>\\w))?(?:_(?P<occ>-?\\d\\.\\d+?))?$')

Pre-compiled regular expression to match an AtomKey string.

fieldNames = ('respos', 'icode', 'resname', 'atm', 'altloc', 'occ')

fields = (0, 1, 2, 3, 4, 5)

Use this namedtuple to access AtomKey fields. See *AtomKey*

d2h = False

Set True to convert D Deuterium to H Hydrogen on input.

__init__(**args:* IC_Residue | Atom | *list* | *dict* | *str*, ***kwargs: str*) → None

Initialize AtomKey with residue and atom data.

Examples of acceptable input:

```
(<IC_Residue>, 'CA', ...)    : IC_Residue with atom info
(<IC_Residue>, <Atom>)       : IC_Residue with Biopython Atom
([52, None, 'G', 'CA', ...])  : list of ordered data fields
(52, None, 'G', 'CA', ...)    : multiple ordered arguments
({respos: 52, icode: None, atm: 'CA', ...}) : dict with fieldNames
(respos: 52, icode: None, atm: 'CA', ...) : kwargs with fieldNames
52_G_CA, 52B_G_CA, 52_G_CA_0.33, 52_G_CA_B_0.33  : id strings
```

__deepcopy__(*memo*)

Deep copy implementation for AtomKey.

__repr__() → str

Repr string from id.

__hash__() → int

Hash calculated at init from akl tuple.

altloc_match(*other:* AtomKey) → bool

Test AtomKey match to other discounting occupancy and altloc.

is_backbone() → bool

Return True if is N, C, CA, O, or H.

atm() → str

Return atom name : N, CA, CB, O etc.

cr_class() → str | None

Return covalent radii class for atom or None.

__ne__(*other: object*) → bool
Test for inequality.

__eq__(*other: object*) → bool
Test for equality.

__gt__(*other: object*) → bool
Test greater than.

__ge__(*other: object*) → bool
Test greater or equal.

__lt__(*other: object*) → bool
Test less than.

__le__(*other: object*) → bool
Test less or equal.

Bio.PDB.internal_coords.**set_accuracy_95**(*num: float*) → float
Reduce floating point accuracy to 9.5 (xxxx.xxxxx).

Used by *IC_Residue* class writing PIC and SCAD files.

Parameters
num (*float*) – input number

Returns
float with specified accuracy

exception Bio.PDB.internal_coords.**HedronMatchError**
Bases: Exception

Cannot find hedron in residue for given key.

exception Bio.PDB.internal_coords.**MissingAtomError**
Bases: Exception

Missing atom coordinates for hedron or dihedron.

Bio.PDB.kdtrees module

KDTree implementation for fast neighbor searches in 3D structures.

This module implements three objects: KDTree, Point, and Neighbor. Refer to their docstrings for more documentation on usage and implementation.

Bio.PDB.mmcifio module

Write an mmCIF file.

See https://www.iucr.org/resources/cif/spec/version1.1/cifsyntax for syntax.

class Bio.PDB.mmcifio.**MMCIFIO**
Bases: *StructureIO*

Write a Structure object or a mmCIF dictionary as a mmCIF file.

Examples

```
>>> from Bio.PDB import MMCIFParser
>>> from Bio.PDB.mmcifio import MMCIFIO
>>> parser = MMCIFParser()
>>> structure = parser.get_structure("1a8o", "PDB/1A8O.cif")
>>> io=MMCIFIO()
>>> io.set_structure(structure)
>>> io.save("bio-pdb-mmcifio-out.cif")
>>> import os
>>> os.remove("bio-pdb-mmcifio-out.cif")  # tidy up
```

__init__()

Initialise.

set_dict(*dic*)

Set the mmCIF dictionary to be written out.

save(*filepath*, *select=_select*, *preserve_atom_numbering=False*)

Save the structure to a file.

Parameters

- **filepath** (*string or filehandle*) – output file
- **select** (*object*) – selects which entities will be written.

Typically select is a subclass of L{Select}, it should have the following methods:

- accept_model(model)
- accept_chain(chain)
- accept_residue(residue)
- accept_atom(atom)

These methods should return 1 if the entity is to be written out, 0 otherwise.

__annotations__ = {}

Bio.PDB.parse_pdb_header module

Parse header of PDB files into a python dictionary.

Emerged from the Columba database project www.columba-db.de, original author Kristian Rother.

Bio.PDB.parse_pdb_header.**parse_pdb_header**(*infile*)

Return the header lines of a pdb file as a dictionary.

Dictionary keys are: head, deposition_date, release_date, structure_method, resolution, structure_reference, journal_reference, author and compound.

Bio.PDB.qcprot module

Structural alignment using Quaternion Characteristic Polynomial (QCP).

QCPSuperimposer finds the best rotation and translation to put two point sets on top of each other (minimizing the RMSD). This is eg. useful to superimpose crystal structures. QCP stands for Quaternion Characteristic Polynomial, which is used in the algorithm.

Algorithm and original code described in:

Theobald DL. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. Acta Crystallogr A. 2005 Jul;61(Pt 4):478-80. doi: 10.1107/S0108767305015266. Epub 2005 Jun 23. PMID: 15973002.

Bio.PDB.qcprot.**qcp**(*coords1*, *coords2*, *natoms*)

Implement the QCP code in Python.

Input coordinate arrays must be centered at the origin and have shape Nx3.

Variable names match (as much as possible) the C implementation.

class Bio.PDB.qcprot.**QCPSuperimposer**

Bases: object

Quaternion Characteristic Polynomial (QCP) Superimposer.

QCPSuperimposer finds the best rotation and translation to put two point sets on top of each other (minimizing the RMSD). This is eg. useful to superimposing 3D structures of proteins.

QCP stands for Quaternion Characteristic Polynomial, which is used in the algorithm.

Reference:

Douglas L Theobald (2005), "Rapid calculation of RMSDs using a quaternion-based characteristic polynomial.", Acta Crystallogr A 61(4):478-480

__init__()

Initialize the class.

set_atoms(*fixed*, *moving*)

Prepare alignment between two atom lists.

Put (translate/rotate) the atoms in fixed on the atoms in moving, in such a way that the RMSD is minimized.

Parameters

- **fixed** – list of (fixed) atoms
- **moving** – list of (moving) atoms

apply(*atom_list*)

Apply the QCP rotation matrix/translation vector to a set of atoms.

set(*reference_coords*, *coords*)

Set the coordinates to be superimposed.

coords will be put on top of reference_coords.

- reference_coords: an NxDIM array
- coords: an NxDIM array

DIM is the dimension of the points, N is the number of points to be superimposed.

run()

Superimpose the coordinate sets.

get_transformed()

Get the transformed coordinate set.

get_rotran()

Return right multiplying rotation matrix and translation vector.

get_init_rms()

Return the root mean square deviation of untransformed coordinates.

get_rms()

Root mean square deviation of superimposed coordinates.

Bio.PDB.vectors module

Vector class, including rotation-related functions.

Bio.PDB.vectors.**m2rotaxis**(*m*)

Return angles, axis pair that corresponds to rotation matrix m.

The case where `m` is the identity matrix corresponds to a singularity where any rotation axis is valid. In that case, `Vector([1, 0, 0])`, is returned.

Bio.PDB.vectors.**vector_to_axis**(*line*, *point*)

Vector to axis method.

Return the vector between a point and the closest point on a line (ie. the perpendicular projection of the point on the line).

Parameters

- **line** (*L{Vector}*) – vector defining a line
- **point** (*L{Vector}*) – vector defining the point

Bio.PDB.vectors.**rotaxis2m**(*theta*, *vector*)

Calculate left multiplying rotation matrix.

Calculate a left multiplying rotation matrix that rotates theta rad around vector.

Parameters

- **theta** (*float*) – the rotation angle
- **vector** (*L{Vector}*) – the rotation axis

Returns

The rotation matrix, a 3x3 NumPy array.

Examples

```
>>> from numpy import pi
>>> from Bio.PDB.vectors import rotaxis2m
>>> from Bio.PDB.vectors import Vector
>>> m = rotaxis2m(pi, Vector(1, 0, 0))
>>> Vector(1, 2, 3).left_multiply(m)
<Vector 1.00, -2.00, -3.00>
```

Bio.PDB.vectors.**rotaxis**(*theta*, *vector*)

Calculate left multiplying rotation matrix.

Calculate a left multiplying rotation matrix that rotates theta rad around vector.

Parameters

- **theta** (*float*) – the rotation angle
- **vector** (*L{Vector}*) – the rotation axis

Returns

The rotation matrix, a 3x3 NumPy array.

Examples

```
>>> from numpy import pi
>>> from Bio.PDB.vectors import rotaxis2m
>>> from Bio.PDB.vectors import Vector
>>> m = rotaxis2m(pi, Vector(1, 0, 0))
>>> Vector(1, 2, 3).left_multiply(m)
<Vector 1.00, -2.00, -3.00>
```

Bio.PDB.vectors.**refmat**(*p*, *q*)

Return a (left multiplying) matrix that mirrors p onto q.

Returns

The mirror operation, a 3x3 NumPy array.

Examples

```
>>> from Bio.PDB.vectors import refmat
>>> p, q = Vector(1, 2, 3), Vector(2, 3, 5)
>>> mirror = refmat(p, q)
>>> qq = p.left_multiply(mirror)
>>> print(q)
<Vector 2.00, 3.00, 5.00>
>>> print(qq)
<Vector 1.21, 1.82, 3.03>
```

Bio.PDB.vectors.**rotmat**(*p*, *q*)

Return a (left multiplying) matrix that rotates p onto q.

Parameters

- **p** (*L{Vector}*) – moving vector

- **q** (*L{Vector}*) – fixed vector

Returns
rotation matrix that rotates p onto q

Return type
3x3 NumPy array

Examples

```
>>> from Bio.PDB.vectors import rotmat
>>> p, q = Vector(1, 2, 3), Vector(2, 3, 5)
>>> r = rotmat(p, q)
>>> print(q)
<Vector 2.00, 3.00, 5.00>
>>> print(p)
<Vector 1.00, 2.00, 3.00>
>>> p.left_multiply(r)
<Vector 1.21, 1.82, 3.03>
```

Bio.PDB.vectors.**calc_angle**(*v1*, *v2*, *v3*)

Calculate angle method.

Calculate the angle between 3 vectors representing 3 connected points.

Parameters
v3 (*v1, v2,*) – the tree points that define the angle

Returns
angle

Return type
float

Bio.PDB.vectors.**calc_dihedral**(*v1*, *v2*, *v3*, *v4*)

Calculate dihedral angle method.

Calculate the dihedral angle between 4 vectors representing 4 connected points. The angle is in]-pi, pi].

Parameters
v4 (*v1, v2, v3,*) – the four points that define the dihedral angle

class Bio.PDB.vectors.**Vector**(*x*, *y=None*, *z=None*)

Bases: object

3D vector.

__init__(*x*, *y=None*, *z=None*)

Initialize the class.

__repr__()

Return vector 3D coordinates.

__neg__()

Return Vector(-x, -y, -z).

__add__(*other*)

Return Vector+other Vector or scalar.

__sub__(*other*)
Return Vector-other Vector or scalar.

__mul__(*other*)
Return Vector.Vector (dot product).

__truediv__(*x*)
Return Vector(coords/a).

__pow__(*other*)
Return VectorxVector (cross product) or Vectorxscalar.

__getitem__(*i*)
Return value of array index i.

__setitem__(*i*, *value*)
Assign values to array index i.

__contains__(*i*)
Validate if i is in array.

norm()
Return vector norm.

normsq()
Return square of vector norm.

normalize()
Normalize the Vector object.

Changes the state of `self` and doesn't return a value. If you need to chain function calls or create a new object use the `normalized` method.

normalized()
Return a normalized copy of the Vector.

To avoid allocating new objects use the `normalize` method.

angle(*other*)
Return angle between two vectors.

get_array()
Return (a copy of) the array of coordinates.

left_multiply(*matrix*)
Return Vector=Matrix x Vector.

right_multiply(*matrix*)
Return Vector=Vector x Matrix.

copy()
Return a deep copy of the Vector.

Bio.PDB.vectors.**homog_rot_mtx**(*angle_rads: float*, *axis: str*) → ndarray
Generate a 4x4 single-axis NumPy rotation matrix.

Parameters

- **angle_rads** (*float*) – the desired rotation angle in radians
- **axis** (*char*) – character specifying the rotation axis

Bio.PDB.vectors.**set_Z_homog_rot_mtx**(*angle_rads: float*, *mtx: ndarray*)

Update existing Z rotation matrix to new angle.

Bio.PDB.vectors.**set_Y_homog_rot_mtx**(*angle_rads: float*, *mtx: ndarray*)

Update existing Y rotation matrix to new angle.

Bio.PDB.vectors.**set_X_homog_rot_mtx**(*angle_rads: float*, *mtx: ndarray*)

Update existing X rotation matrix to new angle.

Bio.PDB.vectors.**homog_trans_mtx**(*x: float*, *y: float*, *z: float*) → ndarray

Generate a 4x4 NumPy translation matrix.

Parameters

z (*x, y,*) – translation in each axis

Bio.PDB.vectors.**set_homog_trans_mtx**(*x: float*, *y: float*, *z: float*, *mtx: ndarray*)

Update existing translation matrix to new values.

Bio.PDB.vectors.**homog_scale_mtx**(*scale: float*) → ndarray

Generate a 4x4 NumPy scaling matrix.

Parameters

scale (*float*) – scale multiplier

Bio.PDB.vectors.**get_spherical_coordinates**(*xyz: ndarray*) → tuple[float, float, float]

Compute spherical coordinates (r, azimuth, polar_angle) for X,Y,Z point.

Parameters

xyz (*array*) – column vector (3 row x 1 column NumPy array)

Returns

tuple of r, azimuth, polar_angle for input coordinate

Bio.PDB.vectors.**coord_space**(*a0: ndarray*, *a1: ndarray*, *a2: ndarray*, *rev: bool = False*) → tuple[ndarray, ndarray | None]

Generate transformation matrix to coordinate space defined by 3 points.

New coordinate space will have:

acs[0] on XZ plane acs[1] origin acs[2] on +Z axis

Parameters

- **acs** (*NumPy column array x3*) – X,Y,Z column input coordinates x3
- **rev** (*bool*) – if True, also return reverse transformation matrix (to return from coord_space)

Returns

4x4 NumPy array, x2 if rev=True

Bio.PDB.vectors.**multi_rot_Z**(*angle_rads: ndarray*) → ndarray

Create [entries] NumPy Z rotation matrices for [entries] angles.

Parameters

- **entries** – int number of matrices generated.
- **angle_rads** – NumPy array of angles

Returns

entries x 4 x 4 homogeneous rotation matrices

Bio.PDB.vectors.**multi_rot_Y**(*angle_rads: ndarray*) → ndarray

Create [entries] NumPy Y rotation matrices for [entries] angles.

Parameters

- **entries** – int number of matrices generated.
- **angle_rads** – NumPy array of angles

Returns

entries x 4 x 4 homogeneous rotation matrices

Bio.PDB.vectors.**multi_coord_space**(*a3: ndarray*, *dLen: int*, *rev: bool = False*) → ndarray

Generate [dLen] transform matrices to coord space defined by 3 points.

New coordinate space will have:

acs[0] on XZ plane acs[1] origin acs[2] on +Z axis

:param NumPy array [entries]x3x3 [entries] XYZ coords for 3 atoms :param bool rev: if True, also return reverse transformation matrix (to return from coord_space) :returns: [entries] 4x4 NumPy arrays, x2 if rev=True

Module contents

Classes that deal with macromolecular crystal structures.

Includes: PDB and mmCIF parsers, a Structure class, a module to keep a local copy of the PDB up-to-date, selective IO of PDB files, etc.

Original Author: Thomas Hamelryck. Contributions by: - Peter Cock - Joe Greener - Rob Miller - Lenna X. Peterson - Joao Rodrigues - Kristian Rother - Eric Talevich - and many others.

28.1.22 Bio.Pathway package

Subpackages

Bio.Pathway.Rep package

Submodules

Bio.Pathway.Rep.Graph module

get/set abstraction for graph representation.

class Bio.Pathway.Rep.Graph.**Graph**(*nodes=()*)

Bases: object

A directed graph abstraction with labeled edges.

__init__(*nodes=()*)

Initialize a new Graph object.

__eq__(*g*)

Return true if g is equal to this graph.

__repr__()

Return a unique string representation of this graph.

__str__()
Return a concise string description of this graph.

add_node(*node*)
Add a node to this graph.

add_edge(*source*, *to*, *label=None*)
Add an edge to this graph.

child_edges(*parent*)
Return a list of (child, label) pairs for parent.

children(*parent*)
Return a list of unique children for parent.

edges(*label*)
Return a list of all the edges with this label.

labels()
Return a list of all the edge labels in this graph.

nodes()
Return a list of the nodes in this graph.

parent_edges(*child*)
Return a list of (parent, label) pairs for child.

parents(*child*)
Return a list of unique parents for child.

remove_node(*node*)
Remove node and all edges connected to it.

remove_edge(*parent*, *child*, *label*)
Remove edge (NOT IMPLEMENTED).

__hash__ = None

Bio.Pathway.Rep.MultiGraph module

get/set abstraction for multi-graph representation.

class Bio.Pathway.Rep.MultiGraph.**MultiGraph**(*nodes=()*)
Bases: object

A directed multigraph abstraction with labeled edges.

__init__(*nodes=()*)
Initialize a new MultiGraph object.

__eq__(*g*)
Return true if g is equal to this graph.

__repr__()
Return a unique string representation of this graph.

__str__()
Return a concise string description of this graph.

add_node(*node*)
Add a node to this graph.

add_edge(*source*, *to*, *label=None*)
Add an edge to this graph.

child_edges(*parent*)
Return a list of (child, label) pairs for parent.

children(*parent*)
Return a list of unique children for parent.

edges(*label*)
Return a list of all the edges with this label.

labels()
Return a list of all the edge labels in this graph.

nodes()
Return a list of the nodes in this graph.

parent_edges(*child*)
Return a list of (parent, label) pairs for child.

parents(*child*)
Return a list of unique parents for child.

remove_node(*node*)
Remove node and all edges connected to it.

remove_edge(*parent*, *child*, *label*)
Remove edge (NOT IMPLEMENTED).

__hash__ = None

Bio.Pathway.Rep.MultiGraph.**df_search**(*graph*, *root=None*)
Depth first search of g.

Returns a list of all nodes that can be reached from the root node in depth-first order.

If root is not given, the search will be rooted at an arbitrary node.

Bio.Pathway.Rep.MultiGraph.**bf_search**(*graph*, *root=None*)
Breadth first search of g.

Returns a list of all nodes that can be reached from the root node in breadth-first order.

If root is not given, the search will be rooted at an arbitrary node.

Module contents

BioPython Pathway support module.

Bio.Pathway.Rep is a collection of general data abstractions used in the implementation of the Bio.Pathway module. These abstractions are not intended for direct use, but if they fit your needs there's no reason to reinvent the wheel.

Module contents

BioPython Pathway module.

Bio.Pathway is a lightweight class library designed to support the following tasks:

- Data interchange and preprocessing between pathway databases and analysis software.
- Quick prototyping of pathway analysis algorithms

The basic object in the Bio.Pathway model is Interaction, which represents an arbitrary interaction between any number of biochemical species.

Network objects are used to represent the connectivity between species in pathways and reaction networks.

For applications where it is not necessary to explicitly represent network connectivity, the specialized classes Reaction and System should be used in place of Interacton and Network.

The Bio.Pathway classes, especially Interaction, are intentionally designed to be very flexible. Their intended use are as wrappers around database specific records, such as BIND objects. The value-added in this module is a framework for representing collections of reactions in a way that supports graph theoretic and numeric analysis.

Note: This module should be regarded as a prototype only. API changes are likely.
Comments and feature requests are most welcome.

`class` Bio.Pathway.**Reaction**(*reactants=None*, *catalysts=()*, *reversible=0*, *data=None*)

Bases: `object`

Abstraction for a biochemical transformation.

This class represents a (potentially reversible) biochemical transformation of the type:

```
a S1 + b S2 + ... --> c P1 + d P2 + ...
```

where:

- a, b, c, d ... are positive numeric stochiometric coefficients,
- S1, S2, ... are substrates
- P1, P2, ... are products

A Reaction should be viewed as the net result of one or more individual reaction steps, where each step is potentially facilitated by a different catalyst. Support for 'Reaction algebra' will be added at some point in the future.

Attributes:

- reactants – dict of involved species to their stochiometric coefficients: reactants[S] = stochiometric constant for S
- catalysts – list/tuple of tuples of catalysts required for this reaction
- reversible – true iff reaction is reversible

- data – reference to arbitrary additional data

Invariants:

- for all S in reactants: reactants[S] != 0
- for all C in catalysts: catalysts[C] != 0

__init__(*reactants=None*, *catalysts=()*, *reversible=0*, *data=None*)

Initialize a new Reaction object.

__eq__(*r*)

Return true iff self is equal to r.

__hash__()

Return a hashcode for self.

__repr__()

Return a debugging string representation of self.

__str__()

Return a string representation of self.

reverse()

Return a new Reaction that is the reverse of self.

species()

Return a list of all Species involved in self.

class Bio.Pathway.**System**(*reactions=()*)

Bases: object

Abstraction for a collection of reactions.

This class is used in the Bio.Pathway framework to represent an arbitrary collection of reactions without explicitly defined links.

Attributes:

- None

__init__(*reactions=()*)

Initialize a new System object.

__repr__()

Return a debugging string representation of self.

__str__()

Return a string representation of self.

add_reaction(*reaction*)

Add reaction to self.

remove_reaction(*reaction*)

Remove reaction from self.

reactions()

Return a list of the reactions in this system.

Note the order is arbitrary!

species()

Return a list of the species in this system.

stochiometry()

Compute the stoichiometry matrix for self.

Returns (species, reactions, stoch) where:

- species = ordered list of species in this system
- reactions = ordered list of reactions in this system
- stoch = 2D array where stoch[i][j] is coef of the jth species in the ith reaction, as defined by species and reactions above

class Bio.Pathway.**Interaction**

Bases: object

An arbitrary interaction between any number of species.

This class definition is intended solely as a minimal wrapper interface that should be implemented and extended by more specific abstractions.

Attributes:

- data – reference to arbitrary additional data

__hash__()

Return a hashcode for self.

__repr__()

Return a debugging string representation of self.

__str__()

Return a string representation of self.

class Bio.Pathway.**Network**(*species=()*)

Bases: object

A set of species that are explicitly linked by interactions.

The network is a directed multigraph with labeled edges. The nodes in the graph are the biochemical species involved. The edges represent an interaction between two species, and the edge label is a reference to the associated Interaction object.

Attributes:

- None

__init__(*species=()*)

Initialize a new Network object.

__repr__()

Return a debugging string representation of this network.

__str__()

Return a string representation of this network.

add_species(*species*)

Add species to this network.

add_interaction(*source*, *sink*, *interaction*)

Add interaction to this network.

source(*species*)

Return list of unique sources for species.

source_interactions(*species*)

Return list of (source, interaction) pairs for species.

sink(*species*)

Return list of unique sinks for species.

sink_interactions(*species*)

Return list of (sink, interaction) pairs for species.

species()

Return list of the species in this network.

interactions()

Return list of the unique interactions in this network.

28.1.23 Bio.Phylo package

Subpackages

Bio.Phylo.Applications package

Module contents

Phylogenetics command line tool wrappers (OBSOLETE).

We have decided to remove this module in future, and instead recommend building your command and invoking it via the subprocess module directly.

class Bio.Phylo.Applications.**PhymlCommandline**(*cmd='phyml'*, ***kwargs*)

Bases: *AbstractCommandline*

Command-line wrapper for the tree inference program PhyML.

Homepage: http://www.atgc-montpellier.fr/phyml

References

Guindon S, Gascuel O. A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. Systematic Biology, 2003 Oct;52(5):696-704. PubMed PMID: 14530136.

Guindon S, Dufayard JF, Lefort V, Anisimova M, Hordijk W, Gascuel O. New Algorithms and Methods to Estimate Maximum-Likelihood Phylogenies: Assessing the Performance of PhyML 3.0. Systematic Biology, 2010 59(3):307-21.

__init__(*cmd='phyml'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property alpha

Distribution of the gamma distribution shape parameter.

Can be a fixed positive value, or 'e' to get the maximum-likelihood estimate.

This controls the addition of the -a parameter and its associated value. Set this property to the argument value required.

property bootstrap

Number of bootstrap replicates, if value is > 0.

Otherwise:

0: neither approximate likelihood ratio test nor bootstrap values are computed.

-1: approximate likelihood ratio test returning aLRT statistics.

-2: approximate likelihood ratio test returning Chi2-based parametric branch supports.

-4: SH-like branch supports alone.

This controls the addition of the -b parameter and its associated value. Set this property to the argument value required.

property datatype

Datatype 'nt' for nucleotide (default) or 'aa' for amino-acids.

This controls the addition of the -d parameter and its associated value. Set this property to the argument value required.

property frequencies

Character frequencies.

-f e, m, or "fA fC fG fT"

e : Empirical frequencies, determined as follows :

- Nucleotide sequences: (Empirical) the equilibrium base frequencies are estimated by counting the occurrence of the different bases in the alignment.
- Amino-acid sequences: (Empirical) the equilibrium amino-acid frequencies are estimated by counting the occurrence of the different amino-acids in the alignment.

m : ML/model-based frequencies, determined as follows :

- Nucleotide sequences: (ML) the equilibrium base frequencies are estimated using maximum likelihood
- Amino-acid sequences: (Model) the equilibrium amino-acid frequencies are estimated using the frequencies defined by the substitution model.

"fA fC fG fT" : only valid for nucleotide-based models. fA, fC, fG and fT are floating-point numbers that correspond to the frequencies of A, C, G and T, respectively.

This controls the addition of the -f parameter and its associated value. Set this property to the argument value required.

property input

PHYLIP format input nucleotide or amino-acid sequence filenam.

This controls the addition of the -i parameter and its associated value. Set this property to the argument value required.

property input_tree

Starting tree filename. The tree must be in Newick format.

This controls the addition of the -u parameter and its associated value. Set this property to the argument value required.

property model

Substitution model name.

Nucleotide-based models:

HKY85 (default) | JC69 | K80 | F81 | F84 | TN93 | GTR | custom

For the custom option, a string of six digits identifies the model. For instance, 000000 corresponds to F81 (or JC69, provided the distribution of nucleotide frequencies is uniform). 012345 corresponds to GTR. This option can be used for encoding any model that is a nested within GTR.

Amino-acid based models:

LG (default) | WAG | JTT | MtREV | Dayhoff | DCMut | RtREV | CpREV | VT | Blosum62 | MtMam | MtArt | HIVw | HIVb | custom

This controls the addition of the -m parameter and its associated value. Set this property to the argument value required.

property multiple

Number of data sets to analyse (integer).

This controls the addition of the -n parameter and its associated value. Set this property to the argument value required.

property n_rand_starts

Number of initial random trees to be used.

Only valid if SPR searches are to be performed.

This controls the addition of the –n_rand_starts parameter and its associated value. Set this property to the argument value required.

property nclasses

Number of relative substitution rate categories.

Default 1. Must be a positive integer.

This controls the addition of the -c parameter and its associated value. Set this property to the argument value required.

property optimize

Specific parameter optimisation.

tlr : tree topology (t), branch length (l) and rate parameters (r) are optimised.

tl : tree topology and branch length are optimised.

lr : branch length and rate parameters are optimised.

l : branch length are optimised.

r : rate parameters are optimised.

n : no parameter is optimised.

This controls the addition of the -o parameter and its associated value. Set this property to the argument value required.

property pars

Use a minimum parsimony starting tree.

This option is taken into account when the '-u' option is absent and when tree topology modifications are to be done.

This property controls the addition of the -p switch, treat this property as a boolean.

property print_site_lnl

Print the likelihood for each site in file *_phyml_lk.txt.

This property controls the addition of the –print_site_lnl switch, treat this property as a boolean.

property print_trace

Print each phylogeny explored during the tree search process in file *_phyml_trace.txt.

This property controls the addition of the –print_trace switch, treat this property as a boolean.

property prop_invar

Proportion of invariable sites.

Can be a fixed value in the range [0,1], or 'e' to get the maximum-likelihood estimate.

This controls the addition of the -v parameter and its associated value. Set this property to the argument value required.

property quiet

No interactive questions (for running in batch mode).

This property controls the addition of the –quiet switch, treat this property as a boolean.

property r_seed

Seed used to initiate the random number generator.

Must be an integer.

This controls the addition of the –r_seed parameter and its associated value. Set this property to the argument value required.

property rand_start

Sets the initial tree to random.

Only valid if SPR searches are to be performed.

This property controls the addition of the –rand_start switch, treat this property as a boolean.

property run_id

Append the given string at the end of each PhyML output file.

This option may be useful when running simulations involving PhyML.

This controls the addition of the –run_id parameter and its associated value. Set this property to the argument value required.

property search

Tree topology search operation option.

Can be one of:

NNI : default, fast

SPR : a bit slower than NNI

BEST : best of NNI and SPR search

This controls the addition of the -s parameter and its associated value. Set this property to the argument value required.

property sequential

Changes interleaved format (default) to sequential format.

This property controls the addition of the -q switch, treat this property as a boolean.

property ts_tv_ratio

Transition/transversion ratio. (DNA sequences only.)

Can be a fixed positive value (ex:4.0) or e to get the maximum-likelihood estimate.

This controls the addition of the -t parameter and its associated value. Set this property to the argument value required.

class Bio.Phylo.Applications.**RaxmlCommandline**(*cmd='raxmlHPC'*, ***kwargs*)

Bases: *AbstractCommandline*

Command-line wrapper for the tree inference program RAxML.

The required parameters are 'sequences' (-s), 'model' (-m) and 'name' (-n). The parameter 'parsimony_seed' (-p) must also be set for RAxML, but if you do not specify it, this wrapper will set the seed to 10000 for you.

References

Stamatakis A. RAxML-VI-HPC: Maximum Likelihood-based Phylogenetic Analyses with Thousands of Taxa and Mixed Models. Bioinformatics 2006, 22(21):2688-2690.

Homepage: http://sco.h-its.org/exelixis/software.html

Examples

```
>>> from Bio.Phylo.Applications import RaxmlCommandline
>>> raxml_cline = RaxmlCommandline(sequences="Tests/Phylip/interlaced2.phy",
...                                model="PROTCATWAG", name="interlaced2")
>>> print(raxml_cline)
raxmlHPC -m PROTCATWAG -n interlaced2 -p 10000 -s Tests/Phylip/interlaced2.phy
```

You would typically run the command line with raxml_cline() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='raxmlHPC'*, ***kwargs*)

Initialize the class.

property parsimony_seed

Random number seed for the parsimony inferences. This allows you to reproduce your results and will help developers debug the program. This option HAS NO EFFECT in the parallel MPI version.

This controls the addition of the -p parameter and its associated value. Set this property to the argument value required.

__annotations__ = {}

property algorithm

Select algorithm:

a: Rapid Bootstrap analysis and search for best-scoring ML tree in one program run.

b: Draw bipartition information on a tree provided with '-t' based on multiple trees (e.g. form a bootstrap) in a file specified by '-z'.

c: Check if the alignment can be properly read by RAxML.

d: New rapid hill-climbing (DEFAULT).

e: Optimize model+branch lengths for given input tree under GAMMA/GAMMAI only.

g: Compute per site log Likelihoods for one or more trees passed via '-z' and write them to a file that can be read by CONSEL.

h: Compute log likelihood test (SH-test) between best tree passed via '-t' and a bunch of other trees passed via '-z'.

i: Perform a really thorough bootstrap, refinement of final bootstrap tree under GAMMA and a more exhaustive algorithm.

j: Generate a bunch of bootstrapped alignment files from an original alignment file.

m: Compare bipartitions between two bunches of trees passed via '-t' and '-z' respectively. This will return the Pearson correlation between all bipartitions found in the two tree files. A file called RAxML_bipartitionFrequencies.outputFileName will be printed that contains the pair-wise bipartition frequencies of the two sets.

n: Compute the log likelihood score of all trees contained in a tree file provided by '-z' under GAMMA or GAMMA+P-Invar.

o: Old and slower rapid hill-climbing.

p: Perform pure stepwise MP addition of new sequences to an incomplete starting tree.

s: Split up a multi-gene partitioned alignment into the respective subalignments.

t: Do randomized tree searches on one fixed starting tree.

w: Compute ELW test on a bunch of trees passed via '-z'.

x: Compute pair-wise ML distances, ML model parameters will be estimated on an MP starting tree or a user-defined tree passed via '-t', only allowed for GAMMA-based models of rate heterogeneity.

This controls the addition of the -f parameter and its associated value. Set this property to the argument value required.

property binary_constraint

File name of a binary constraint tree. This tree does not need to be comprehensive, i.e. contain all taxa.

This controls the addition of the -r parameter and its associated value. Set this property to the argument value required.

property bipartition_filename

Name of a file containing multiple trees, e.g. from a bootstrap run, that shall be used to draw bipartition values onto a tree provided with '-t'. It can also be used to compute per-site log likelihoods in combination with '-f g', and to read a bunch of trees for a couple of other options ('-f h', '-f m', '-f n').

This controls the addition of the -z parameter and its associated value. Set this property to the argument value required.

property bootstrap_branch_lengths

Print bootstrapped trees with branch lengths. The bootstraps will run a bit longer, because model parameters will be optimized at the end of each run. Use with CATMIX/PROTMIX or GAMMA/GAMMAI.

This property controls the addition of the -k switch, treat this property as a boolean.

property bootstrap_seed

Random seed for bootstrapping.

This controls the addition of the -b parameter and its associated value. Set this property to the argument value required.

property checkpoints

Write checkpoints (intermediate tree topologies).

This property controls the addition of the -j switch, treat this property as a boolean.

property cluster_threshold

Threshold for sequence similarity clustering. RAxML will then print out an alignment to a file called sequenceFileName.reducedBy.threshold that only contains sequences <= the specified threshold that must be between 0.0 and 1.0. RAxML uses the QT-clustering algorithm to perform this task. In addition, a file called RAxML_reducedList.outputFileName will be written that contains clustering information.

This controls the addition of the -l parameter and its associated value. Set this property to the argument value required.

property cluster_threshold_fast

Same functionality as '-l', but uses a less exhaustive and thus faster clustering algorithm. This is intended for very large datasets with more than 20,000-30,000 sequences.

This controls the addition of the -L parameter and its associated value. Set this property to the argument value required.

property epsilon

Set model optimization precision in log likelihood units for final optimization of tree topology under MIX/MIXI or GAMMA/GAMMAI.Default: 0.1 for models not using proportion of invariant sites estimate; 0.001 for models using proportion of invariant sites estimate.

This controls the addition of the -e parameter and its associated value. Set this property to the argument value required.

property exclude_filename

An exclude file name, containing a specification of alignment positions you wish to exclude. Format is similar to Nexus, the file shall contain entries like '100-200 300-400'; to exclude a single column write, e.g., '100-100'. If you use a mixed model, an appropriately adapted model file will be written.

This controls the addition of the -E parameter and its associated value. Set this property to the argument value required.

property grouping_constraint

File name of a multifurcating constraint tree. this tree does not need to be comprehensive, i.e. contain all taxa.

This controls the addition of the -g parameter and its associated value. Set this property to the argument value required.

property model

Model of Nucleotide or Amino Acid Substitution:

NUCLEOTIDES:

GTRCAT : GTR + Optimization of substitution rates + Optimization of site-specific evolutionary rates which are categorized into numberOfCategories distinct rate categories for greater computational efficiency if you do a multiple analysis with '-#' or '-N' but without bootstrapping the program will use GTRMIX instead

GTRGAMMA : GTR + Optimization of substitution rates + GAMMA model of rate heterogeneity (alpha parameter will be estimated)

GTRMIX : Inference of the tree under GTRCAT and thereafter evaluation of the final tree topology under GTRGAMMA

GTRCAT_GAMMA : Inference of the tree with site-specific evolutionary rates. However, here rates are categorized using the 4 discrete GAMMA rates. Evaluation of the final tree topology under GTRGAMMA

GTRGAMMAI : Same as GTRGAMMA, but with estimate of proportion of invariable sites

GTRMIXI : Same as GTRMIX, but with estimate of proportion of invariable sites

GTRCAT_GAMMAI : Same as GTRCAT_GAMMA, but with estimate of proportion of invariable sites

AMINO ACIDS:

PROTCATmatrixName[F] : specified AA matrix + Optimization of substitution rates + Optimization of site-specific evolutionary rates which are categorized into numberOfCategories distinct rate categories for greater computational efficiency if you do a multiple analysis with '-#' or '-N' but without bootstrapping the program will use PROTMIX... instead

PROTGAMMAmatrixName[F] : specified AA matrix + Optimization of substitution rates + GAMMA model of rate heterogeneity (alpha parameter will be estimated)

PROTMIXmatrixName[F] : Inference of the tree under specified AA matrix + CAT and thereafter evaluation of the final tree topology under specified AA matrix + GAMMA

PROTCAT_GAMMAmatrixName[F] : Inference of the tree under specified AA matrix and site-specific evolutionary rates. However, here rates are categorized using the 4 discrete GAMMA rates. Evaluation of the final tree topology under specified AA matrix + GAMMA

PROTGAMMAImatrixName[F] : Same as PROTGAMMAmatrixName[F], but with estimate of proportion of invariable sites

PROTMIXImatrixName[F] : Same as PROTMIXmatrixName[F], but with estimate of proportion of invariable sites

PROTCAT_GAMMAImatrixName[F] : Same as PROTCAT_GAMMAmatrixName[F], but with estimate of proportion of invariable sites

Available AA substitution models: DAYHOFF, DCMUT, JTT, MTREV, WAG, RTREV, CPREV, VT, BLOSUM62, MTMAM, GTR With the optional 'F' appendix you can specify if you want to use empirical base frequencies Please not that for mixed models you can in addition specify the per-gene AA model in the mixed model file (see manual for details)

This controls the addition of the -m parameter and its associated value. Set this property to the argument value required.

property name

Name used in the output files.

This controls the addition of the -n parameter and its associated value. Set this property to the argument value required.

property num_bootstrap_searches

Number of multiple bootstrap searches per replicate. Use this to obtain better ML trees for each replicate. Default: 1 ML search per bootstrap replicate.

This controls the addition of the -u parameter and its associated value. Set this property to the argument value required.

property num_categories

Number of distinct rate categories for RAxML when evolution model is set to GTRCAT or GTRMIX.Individual per-site rates are categorized into this many rate categories to accelerate computations. Default: 25.

This controls the addition of the -c parameter and its associated value. Set this property to the argument value required.

property num_replicates

Number of alternative runs on distinct starting trees. In combination with the '-b' option, this will invoke a multiple bootstrap analysis. DEFAULT: 1 single analysis.Note that '-N' has been added as an alternative since '-#' sometimes caused problems with certain MPI job submission systems, since '-#' is often used to start comments.

This controls the addition of the -N parameter and its associated value. Set this property to the argument value required.

property outgroup

Name of a single outgroup or a comma-separated list of outgroups, eg '-o Rat' or '-o Rat,Mouse'. In case that multiple outgroups are not monophyletic the first name in the list will be selected as outgroup. Don't leave spaces between taxon names!

This controls the addition of the -o parameter and its associated value. Set this property to the argument value required.

property parsimony

Only compute a parsimony starting tree, then exit.

This property controls the addition of the -y switch, treat this property as a boolean.

property partition_branch_lengths

Switch on estimation of individual per-partition branch lengths. Only has effect when used in combination with 'partition_filename' ('-q'). Branch lengths for individual partitions will be printed to separate files. A weighted average of the branch lengths is computed by using the respective partition lengths.

This property controls the addition of the -M switch, treat this property as a boolean.

property partition_filename

File name containing the assignment of models to alignment partitions for multiple models of substitution. For the syntax of this file please consult the RAxML manual.

This controls the addition of the -q parameter and its associated value. Set this property to the argument value required.

property protein_model

File name of a user-defined AA (Protein) substitution model. This file must contain 420 entries, the first 400 being the AA substitution rates (this must be a symmetric matrix) and the last 20 are the empirical base frequencies.

This controls the addition of the -P parameter and its associated value. Set this property to the argument value required.

property random_starting_tree

Start ML optimization from random starting tree.

This property controls the addition of the -d switch, treat this property as a boolean.

property rapid_bootstrap_seed

Random seed for rapid bootstrapping.

This controls the addition of the -x parameter and its associated value. Set this property to the argument value required.

property rearrangements

Initial rearrangement setting for the subsequent application of topological changes phase.

This controls the addition of the -i parameter and its associated value. Set this property to the argument value required.

property sequences

Name of the alignment data file, in PHYLIP format.

This controls the addition of the -s parameter and its associated value. Set this property to the argument value required.

property starting_tree

File name of a user starting tree, in Newick format.

This controls the addition of the -t parameter and its associated value. Set this property to the argument value required.

property threads

Number of threads to run. PTHREADS VERSION ONLY! Make sure to set this at most the number of CPUs you have on your machine, otherwise, there will be a huge performance decrease!

This controls the addition of the -T parameter and its associated value. Set this property to the argument value required.

property version

Display version information.

This property controls the addition of the -v switch, treat this property as a boolean.

property weight_filename

Name of a column weight file to assign individual weights to each column of the alignment. Those weights must be integers separated by any type and number of whitespaces within a separate file.

This controls the addition of the -a parameter and its associated value. Set this property to the argument value required.

property working_dir

Name of the working directory where RAxML will write its output files. Default: current directory.

This controls the addition of the -w parameter and its associated value. Set this property to the argument value required.

class Bio.Phylo.Applications.**FastTreeCommandline**(*cmd='fasttree'*, ***kwargs*)

Bases: *AbstractCommandline*

Command-line wrapper for FastTree.

Only the `input` and `out` parameters are mandatory.

From the terminal command line use `fasttree.exe -help` or `fasttree.exe -expert` for more explanation of usage options.

Homepage: http://www.microbesonline.org/fasttree/

References

Price, M.N., Dehal, P.S., and Arkin, A.P. (2010) FastTree 2 – Approximately Maximum-Likelihood Trees for Large Alignments. PLoS ONE, 5(3):e9490. https://doi.org/10.1371/journal.pone.0009490.

Examples

This is an example on Windows:

```
import _Fasttree
fasttree_exe = r"C:\FasttreeWin32\fasttree.exe"
cmd = _Fasttree.FastTreeCommandline(fasttree_exe,
...                                 input=r'C:\Input\ExampleAlignment.fsa',
...                                 out=r'C:\Output\ExampleTree.tree')
print(cmd)
out, err = cmd()
print(out)
print(err)
```

__init__(*cmd='fasttree'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property bionj

Join options: weighted joins as in BIONJ.

FastTree will also weight joins during NNIs.

This property controls the addition of the -bionj switch, treat this property as a boolean.

property boot

Specify the number of resamples for support values.

Support value options: By default, FastTree computes local support values by resampling the site likelihoods 1,000 times and the Shimodaira Hasegawa test. If you specify -nome, it will compute minimum-evolution bootstrap supports instead In either case, the support values are proportions ranging from 0 to 1

Use -nosupport to turn off support values or -boot 100 to use just 100 resamples.

This controls the addition of the -boot parameter and its associated value. Set this property to the argument value required.

property cat

Maximum likelihood model options.

Specify the number of rate categories of sites (default 20).

This controls the addition of the -cat parameter and its associated value. Set this property to the argument value required.

property close

Modify the close heuristic for the top-hit list

> Top-hit heuristics: By default, FastTree uses a top-hit list to speed up search -close 0.75 – modify the close heuristic, lower is more conservative.

This controls the addition of the -close parameter and its associated value. Set this property to the argument value required.

property constraintWeight

Weight strength of constraints in topology searching.

> Constrained topology search options: -constraintWeight – how strongly to weight the constraints. A value of 1 means a penalty of 1 in tree length for violating a constraint Default: 100.0

This controls the addition of the -constraintWeight parameter and its associated value. Set this property to the argument value required.

property constraints

Specifies an alignment file for use with constrained topology searching

> Constrained topology search options: -constraints alignmentfile – an alignment with values of 0, 1, and - Not all sequences need be present. A column of 0s and 1s defines a constrained split. Some constraints may be violated (see 'violating constraints:' in standard error).

This controls the addition of the -constraints parameter and its associated value. Set this property to the argument value required.

property expert

Show the expert level help.

This property controls the addition of the -expert switch, treat this property as a boolean.

property fastest

Search the visible set (the top hit for each node) only.

> Searching for the best join: By default, FastTree combines the 'visible set' of fast neighbor-joining with local hill-climbing as in relaxed neighbor-joining -fastest – search the visible set (the top hit for each node) only Unlike the original fast neighbor-joining, -fastest updates visible(C) after joining A and B if join(AB,C) is better than join(C,visible(C)) -fastest also updates out-distances in a very lazy way, -fastest sets -2nd on as well, use -fastest -no2nd to avoid this

This property controls the addition of the -fastest switch, treat this property as a boolean.

property gamma

Report the likelihood under the discrete gamma model.

> Maximum likelihood model options: -gamma – after the final round of optimizing branch lengths with the CAT model, report the likelihood under the discrete gamma model with the same number of categories. FastTree uses the same branch lengths but optimizes the gamma shape parameter and the scale of the lengths. The final tree will have rescaled lengths. Used with -log, this also generates per-site likelihoods for use with CONSEL, see GammaLogToPaup.pl and documentation on the FastTree web site.

This property controls the addition of the -gamma switch, treat this property as a boolean.

property gtr

Maximum likelihood model options.

> Use generalized time-reversible instead of (default) Jukes-Cantor (nt only)

This property controls the addition of the -gtr switch, treat this property as a boolean.

property gtrfreq

-gtrfreq A C G T

This controls the addition of the -gtrfreq parameter and its associated value. Set this property to the argument value required.

property gtrrates

-gtrrates ac ag at cg ct gt

This controls the addition of the -gtrrates parameter and its associated value. Set this property to the argument value required.

property help

Show the help.

This property controls the addition of the -help switch, treat this property as a boolean.

property input

Enter <input file>

An input file of sequence alignments in fasta or phylip format is needed. By default FastTree expects protein alignments, use -nt for nucleotides.

This controls the addition of the input parameter and its associated value. Set this property to the argument value required.

property intree

-intree newickfile – read the starting tree in from newickfile.

Any branch lengths in the starting trees are ignored. -intree with -n will read a separate starting tree for each alignment.

This controls the addition of the -intree parameter and its associated value. Set this property to the argument value required.

property intree1

intree1 newickfile – read the same starting tree for each alignment.

This controls the addition of the -intree1 parameter and its associated value. Set this property to the argument value required.

property log

Create log files of data such as intermediate trees and per-site rates

-log logfile – save intermediate trees so you can extract the trees and restart long-running jobs if they crash -log also reports the per-site rates (1 means slowest category).

This controls the addition of the -log parameter and its associated value. Set this property to the argument value required.

property makematrix

-makematrix [alignment]

This controls the addition of the -makematrix parameter and its associated value. Set this property to the argument value required.

property matrix

Specify a matrix for nucleotide or amino acid distances

Distances: Default: For protein sequences, log-corrected distances and an amino acid dissimilarity matrix derived from BLOSUM45 or for nucleotide sequences, Jukes-Cantor distances To specify a different matrix, use -matrix FilePrefix or -nomatrix

This controls the addition of the -matrix parameter and its associated value. Set this property to the argument value required.

property mlacc

Option for optimization of branches at each NNI.

Topology refinement: By default, FastTree tries to improve the tree with up to 4*log2(N) rounds of minimum-evolution nearest-neighbor interchanges (NNI), where N is the number of unique sequences, 2 rounds of subtree-prune-regraft (SPR) moves (also min. evo.), and up to 2*log(N) rounds of maximum-likelihood NNIs. Use -nni to set the number of rounds of min. evo. NNIs, and -spr to set the rounds of SPRs. Use -mlacc 2 or -mlacc 3 to always optimize all 5 branches at each NNI, and to optimize all 5 branches in 2 or 3 rounds.

This controls the addition of the -mlacc parameter and its associated value. Set this property to the argument value required.

property mllen

Optimize branch lengths on a fixed topology.

Topology refinement: By default, FastTree tries to improve the tree with up to 4*log2(N) rounds of minimum-evolution nearest-neighbor interchanges (NNI), where N is the number of unique sequences, 2 rounds of subtree-prune-regraft (SPR) moves (also min. evo.), and up to 2*log(N) rounds of maximum-likelihood NNIs. Use -nni to set the number of rounds of min. evo. NNIs, and -spr to set the rounds of SPRs. Use -mllen to optimize branch lengths without ML NNIs Use -mllen -nome with -intree to optimize branch lengths on a fixed topology.

This property controls the addition of the -mllen switch, treat this property as a boolean.

property mlnni

Set the number of rounds of maximum-likelihood NNIs.

Topology refinement: By default, FastTree tries to improve the tree with up to 4*log2(N) rounds of minimum-evolution nearest-neighbor interchanges (NNI), where N is the number of unique sequences, 2 rounds of subtree-prune-regraft (SPR) moves (also min. evo.), and up to 2*log(N) rounds of maximum-likelihood NNIs. Use -nni to set the number of rounds of min. evo. NNIs, and -spr to set the rounds of SPRs. Use -mlnni to set the number of rounds of maximum-likelihood NNIs.

This controls the addition of the -mlnni parameter and its associated value. Set this property to the argument value required.

property n

-n – read N multiple alignments in.

This only works with phylip interleaved format. For example, you can use it with the output from phylip's seqboot. If you use -n, FastTree will write 1 tree per line to standard output.

This controls the addition of the -n parameter and its associated value. Set this property to the argument value required.

property nj

Join options: regular (unweighted) neighbor-joining (default)

This property controls the addition of the -nj switch, treat this property as a boolean.

property nni

Set the rounds of minimum-evolution nearest-neighbor interchanges

Topology refinement: By default, FastTree tries to improve the tree with up to 4*log2(N) rounds of minimum-evolution nearest-neighbor interchanges (NNI), where N is the number of unique

sequences, 2 rounds of subtree-prune-regraft (SPR) moves (also min. evo.), and up to 2*log(N) rounds of maximum-likelihood NNIs. Use -nni to set the number of rounds of min. evo. NNIs.

This controls the addition of the -nni parameter and its associated value. Set this property to the argument value required.

property no2nd

Turn 2nd-level top hits heuristic off.

Top-hit heuristics: By default, FastTree uses a top-hit list to speed up search Use -notop (or -slow) to turn this feature off and compare all leaves to each other, and all new joined nodes to each other

-2nd or -no2nd to turn 2nd-level top hits heuristic on or off This reduces memory usage and running time but may lead to marginal reductions in tree quality. (By default, -fastest turns on -2nd.)

This property controls the addition of the -no2nd switch, treat this property as a boolean.

property nocat

Maximum likelihood model options: No CAT model (just 1 category)

This property controls the addition of the -nocat switch, treat this property as a boolean.

property nomatrix

Specify that no matrix should be used for nucleotide or amino acid distances

Distances: Default: For protein sequences, log-corrected distances and an amino acid dissimilarity matrix derived from BLOSUM45 or for nucleotide sequences, Jukes-Cantor distances To specify a different matrix, use -matrix FilePrefix or -nomatrix

This property controls the addition of the -nomatrix switch, treat this property as a boolean.

property nome

Changes support values calculation to a minimum-evolution bootstrap method.

Topology refinement: By default, FastTree tries to improve the tree with up to 4*log2(N) rounds of minimum-evolution nearest-neighbor interchanges (NNI), where N is the number of unique sequences, 2 rounds of subtree-prune-regraft (SPR) moves (also min. evo.), and up to 2*log(N) rounds of maximum-likelihood NNIs. Use -nni to set the number of rounds of min. evo. NNIs, and -spr to set the rounds of SPRs. Use -mllen to optimize branch lengths without ML NNIs Use -mllen -nome with -intree to optimize branch lengths on a fixed topology

Support value options: By default, FastTree computes local support values by resampling the site likelihoods 1,000 times and the Shimodaira Hasegawa test. If you specify -nome, it will compute minimum-evolution bootstrap supports instead In either case, the support values are proportions ranging from 0 to 1.

This property controls the addition of the -nome switch, treat this property as a boolean.

property noml

Deactivate min-evo NNIs and SPRs.

Topology refinement: By default, FastTree tries to improve the tree with up to 4*log2(N) rounds of minimum-evolution nearest-neighbor interchanges (NNI), where N is the number of unique sequences, 2 rounds of subtree-prune-regraft (SPR) moves (also min. evo.), and up to 2*log(N) rounds of maximum-likelihood NNIs. Use -nni to set the number of rounds of min. evo. NNIs, and -spr to set the rounds of SPRs. Use -noml to turn off both min-evo NNIs and SPRs (useful if refining an approximately maximum-likelihood tree with further NNIs).

This property controls the addition of the -noml switch, treat this property as a boolean.

property nopr

-nopr – do not write the progress indicator to stderr.

This property controls the addition of the -nopr switch, treat this property as a boolean.

property nosupport

Turn off support values.

Support value options: By default, FastTree computes local support values by resampling the site likelihoods 1,000 times and the Shimodaira Hasegawa test. If you specify -nome, it will compute minimum-evolution bootstrap supports instead In either case, the support values are proportions ranging from 0 to 1

Use -nosupport to turn off support values or -boot 100 to use just 100 resamples.

This property controls the addition of the -nosupport switch, treat this property as a boolean.

property notop

Turn off top-hit list to speed up search

Top-hit heuristics: By default, FastTree uses a top-hit list to speed up search Use -notop (or -slow) to turn this feature off and compare all leaves to each other, and all new joined nodes to each other.

This property controls the addition of the -notop switch, treat this property as a boolean.

property nt

By default FastTree expects protein alignments, use -nt for nucleotides

This property controls the addition of the -nt switch, treat this property as a boolean.

property out

Enter <output file>

The path to a Newick Tree output file needs to be specified.

This controls the addition of the -out parameter and its associated value. Set this property to the argument value required.

property pseudo

-pseudo [weight] – Pseudocounts are used with sequence distance estimation.

Use pseudocounts to estimate distances between sequences with little or no overlap. (Off by default.) Recommended if analyzing the alignment has sequences with little or no overlap. If the weight is not specified, it is 1.0

This controls the addition of the -pseudo parameter and its associated value. Set this property to the argument value required.

property quiet

-quiet – do not write to standard error during normal operation

(no progress indicator, no options summary, no likelihood values, etc.)

This property controls the addition of the -quiet switch, treat this property as a boolean.

property quote

-quote – add quotes to sequence names in output.

Quote sequence names in the output and allow spaces, commas, parentheses, and colons in them but not ‘ characters (fasta files only).

This property controls the addition of the -quote switch, treat this property as a boolean.

property rawdist

Turn off or adjust log-correction in AA or NT distances.

Use -rawdist to turn the log-correction off or to use %different instead of Jukes-Cantor in AA or NT distances

Distances: Default: For protein sequences, log-corrected distances and an amino acid dissimilarity matrix derived from BLOSUM45 or for nucleotide sequences, Jukes-Cantor distances To specify a different matrix, use -matrix FilePrefix or -nomatrix

This property controls the addition of the -rawdist switch, treat this property as a boolean.

property refresh

Parameter for conditions that joined nodes are compared to other nodes

Top-hit heuristics: By default, FastTree uses a top-hit list to speed up search -refresh 0.8 – compare a joined node to all other nodes if its top-hit list is less than 80% of the desired length, or if the age of the top-hit list is log2(m) or greater.

This controls the addition of the -refresh parameter and its associated value. Set this property to the argument value required.

property second

Turn 2nd-level top hits heuristic on.

Top-hit heuristics: By default, FastTree uses a top-hit list to speed up search Use -notop (or -slow) to turn this feature off and compare all leaves to each other, and all new joined nodes to each other

-2nd or -no2nd to turn 2nd-level top hits heuristic on or off This reduces memory usage and running time but may lead to marginal reductions in tree quality. (By default, -fastest turns on -2nd.)

This property controls the addition of the -2nd switch, treat this property as a boolean.

property seed

Use -seed to initialize the random number generator.

Support value options: By default, FastTree computes local support values by resampling the site likelihoods 1,000 times and the Shimodaira Hasegawa test. If you specify -nome, it will compute minimum-evolution bootstrap supports instead In either case, the support values are proportions ranging from 0 to 1.

This controls the addition of the -seed parameter and its associated value. Set this property to the argument value required.

property slow

Use an exhaustive search.

Searching for the best join: By default, FastTree combines the 'visible set' of fast neighbor-joining with local hill-climbing as in relaxed neighbor-joining -slow – exhaustive search (like NJ or BIONJ, but different gap handling) -slow takes half an hour instead of 8 seconds for 1,250 proteins

This property controls the addition of the -slow switch, treat this property as a boolean.

property slownni

Turn off heuristics to avoid constant subtrees with NNIs.

Topology refinement: By default, FastTree tries to improve the tree with up to 4*log2(N) rounds of minimum-evolution nearest-neighbor interchanges (NNI), where N is the number of unique sequences, 2 rounds of subtree-prune-regraft (SPR) moves (also min. evo.), and up to 2*log(N)

rounds of maximum-likelihood NNIs. Use -nni to set the number of rounds of min. evo. NNIs, and -spr to set the rounds of SPRs. Use -slownni to turn off heuristics to avoid constant subtrees (affects both ML and ME NNIs).

This property controls the addition of the -slownni switch, treat this property as a boolean.

property spr

Set the rounds of subtree-prune-regraft moves

Topology refinement: By default, FastTree tries to improve the tree with up to 4*log2(N) rounds of minimum-evolution nearest-neighbor interchanges (NNI), where N is the number of unique sequences, 2 rounds of subtree-prune-regraft (SPR) moves (also min. evo.), and up to 2*log(N) rounds of maximum-likelihood NNIs. Use -nni to set the number of rounds of min. evo. NNIs, and -spr to set the rounds of SPRs.

This controls the addition of the -spr parameter and its associated value. Set this property to the argument value required.

property sprlength

Set maximum SPR move length in topology refinement (default 10).

Topology refinement: By default, FastTree tries to improve the tree with up to 4*log2(N) rounds of minimum-evolution nearest-neighbor interchanges (NNI), where N is the number of unique sequences, 2 rounds of subtree-prune-regraft (SPR) moves (also min. evo.), and up to 2*log(N) rounds of maximum-likelihood NNIs. Use -nni to set the number of rounds of min. evo. NNIs, and -spr to set the rounds of SPRs.

This controls the addition of the -sprlength parameter and its associated value. Set this property to the argument value required.

property top

Top-hit list to speed up search

Top-hit heuristics: By default, FastTree uses a top-hit list to speed up search Use -notop (or -slow) to turn this feature off and compare all leaves to each other, and all new joined nodes to each other.

This property controls the addition of the -top switch, treat this property as a boolean.

property topm

Change the top hits calculation method

Top-hit heuristics: By default, FastTree uses a top-hit list to speed up search -topm 1.0 – set the top-hit list size to parameter*sqrt(N) FastTree estimates the top m hits of a leaf from the top 2*m hits of a 'close' neighbor, where close is defined as d(seed,close) < 0.75 * d(seed, hit of rank 2*m), and updates the top-hits as joins proceed.

This controls the addition of the -topm parameter and its associated value. Set this property to the argument value required.

property wag

Maximum likelihood model options.

Whelan-And-Goldman 2001 model instead of (default) Jones-Taylor-Thorton 1992 model (a.a. only)

This property controls the addition of the -wag switch, treat this property as a boolean.

Bio.Phylo.PAML package

Submodules

Bio.Phylo.PAML.baseml module

Classes for the support of baseml.

Maximum likelihood analysis of nucleotide sequences.

exception Bio.Phylo.PAML.baseml.**BasemlError**

Bases: OSError

BASEML failed. Run with verbose=True to view BASEML's error message.

class Bio.Phylo.PAML.baseml.**Baseml**(*alignment=None*, *tree=None*, *working_dir=None*, *out_file=None*)

Bases: Paml

An interface to BASEML, part of the PAML package.

__init__(*alignment=None*, *tree=None*, *working_dir=None*, *out_file=None*)

Initialize the Baseml instance.

The user may optionally pass in strings specifying the locations of the input alignment and tree files, the working directory and the final output file.

write_ctl_file()

Dynamically build a BASEML control file from the options.

The control file is written to the location specified by the ctl_file property of the baseml class.

read_ctl_file(*ctl_file*)

Parse a control file and load the options into the Baseml instance.

run(*ctl_file=None*, *verbose=False*, *command='baseml'*, *parse=True*)

Run baseml using the current configuration.

Check that the tree attribute is specified and exists, and then run baseml. If parse is True then read and return the result, otherwise return none.

The arguments may be passed as either absolute or relative paths, despite the fact that BASEML requires relative paths.

Bio.Phylo.PAML.baseml.**read**(*results_file*)

Parse a BASEML results file.

Bio.Phylo.PAML.chi2 module

Methods to calculate p-values from a Chi-squared cumulative distribution function.

for likelihood ratio tests.

Bio.Phylo.PAML.chi2.**cdf_chi2**(*df*, *stat*)

Compute p-value, from distribution function and test statistics.

Bio.Phylo.PAML.codeml module

Classes for the support of CODEML.

Maximum likelihood analysis using codon substitution models.

exception Bio.Phylo.PAML.codeml.**CodemlError**

Bases: OSError

CODEML failed. Run with verbose=True to view CODEML's error message.

class Bio.Phylo.PAML.codeml.**Codeml**(*alignment=None*, *tree=None*, *working_dir=None*, *out_file=None*)

Bases: Paml

An interface to CODEML, part of the PAML package.

__init__(*alignment=None*, *tree=None*, *working_dir=None*, *out_file=None*)

Initialize the codeml instance.

The user may optionally pass in strings specifying the locations of the input alignment and tree files, the working directory and the final output file. Other options found in the CODEML control have typical settings by default to run site class models 0, 1 and 2 on a nucleotide alignment.

write_ctl_file()

Dynamically build a CODEML control file from the options.

The control file is written to the location specified by the ctl_file property of the codeml class.

read_ctl_file(*ctl_file*)

Parse a control file and load the options into the Codeml instance.

print_options()

Print out all of the options and their current settings.

run(*ctl_file=None*, *verbose=False*, *command='codeml'*, *parse=True*)

Run codeml using the current configuration.

If parse is True then read and return the results, otherwise return None.

The arguments may be passed as either absolute or relative paths, despite the fact that CODEML requires relative paths.

__annotations__ = {}

Bio.Phylo.PAML.codeml.**read**(*results_file*)

Parse a CODEML results file.

Bio.Phylo.PAML.yn00 module

Classes for the support of yn00.

Yang and Nielsen 2000, estimating synonymous and nonsynonymous substitution rates in pairwise comparison of protein-coding DNA sequences.

exception Bio.Phylo.PAML.yn00.**Yn00Error**

Bases: OSError

yn00 failed. Run with verbose=True to view yn00's error message.

class Bio.Phylo.PAML.yn00.**Yn00**(*alignment=None*, *working_dir=None*, *out_file=None*)

Bases: Paml

An interface to yn00, part of the PAML package.

__init__(*alignment=None*, *working_dir=None*, *out_file=None*)

Initialize the Yn00 instance.

The user may optionally pass in strings specifying the locations of the input alignment, the working directory and the final output file.

write_ctl_file()

Dynamically build a yn00 control file from the options.

The control file is written to the location specified by the ctl_file property of the yn00 class.

read_ctl_file(*ctl_file*)

Parse a control file and load the options into the yn00 instance.

run(*ctl_file=None*, *verbose=False*, *command='yn00'*, *parse=True*)

Run yn00 using the current configuration.

If parse is True then read and return the result, otherwise return None.

__annotations__ = {}

Bio.Phylo.PAML.yn00.**read**(*results_file*)

Parse a yn00 results file.

Module contents

Support for PAML.

Submodules

Bio.Phylo.BaseTree module

Base classes for Bio.Phylo objects.

All object representations for phylogenetic trees should derive from these base classes in order to use the common methods defined on them.

class Bio.Phylo.BaseTree.**TreeElement**

Bases: object

Base class for all Bio.Phylo classes.

__repr__() → str

Show this object's constructor with its primitive arguments.

__str__() → str

Return str(self).

class Bio.Phylo.BaseTree.**TreeMixin**

Bases: `object`

Methods for Tree- and Clade-based classes.

This lets `Tree` and `Clade` support the same traversal and searching operations without requiring Clade to inherit from Tree, so Clade isn't required to have all of Tree's attributes – just `root` (a Clade instance) and `is_terminal`.

find_any(**args*, ***kwargs*)

Return the first element found by find_elements(), or None.

This is also useful for checking whether any matching element exists in the tree, and can be used in a conditional expression.

find_elements(*target=None*, *terminal=None*, *order='preorder'*, ***kwargs*)

Find all tree elements matching the given attributes.

The arbitrary keyword arguments indicate the attribute name of the sub-element and the value to match: string, integer or boolean. Strings are evaluated as regular expression matches; integers are compared directly for equality, and booleans evaluate the attribute's truth value (True or False) before comparing. To handle nonzero floats, search with a boolean argument, then filter the result manually.

If no keyword arguments are given, then just the class type is used for matching.

The result is an iterable through all matching objects, by depth-first search. (Not necessarily the same order as the elements appear in the source file!)

Parameters

target
[TreeElement instance, type, dict, or callable] Specifies the characteristics to search for. (The default, TreeElement, matches any standard Bio.Phylo type.)

terminal
[bool] A boolean value to select for or against terminal nodes (a.k.a. leaf nodes). True searches for only terminal nodes, False excludes terminal nodes, and the default, None, searches both terminal and non-terminal nodes, as well as any tree elements lacking the `is_terminal` method.

order
[{'preorder', 'postorder', 'level'}] Tree traversal order: 'preorder' (default) is depth-first search, 'postorder' is DFS with child nodes preceding parents, and 'level' is breadth-first search.

Examples

```
>>> from Bio import Phylo
>>> phx = Phylo.PhyloXMLIO.read('PhyloXML/phyloxml_examples.xml')
>>> matches = phx.phylogenies[5].find_elements(code='OCTVU')
>>> next(matches)
Taxonomy(code='OCTVU', scientific_name='Octopus vulgaris')
```

find_clades(*target=None*, *terminal=None*, *order='preorder'*, ***kwargs*)

Find each clade containing a matching element.

That is, find each element as with find_elements(), but return the corresponding clade object. (This is usually what you want.)

Returns
an iterable through all matching objects, searching depth-first (preorder) by default.

`get_path`(*target=None*, ***kwargs*)

List the clades directly between this root and the given target.

Returns
list of all clade objects along this path, ending with the given target, but excluding the root clade.

`get_nonterminals`(*order='preorder'*)

Get a list of all of this tree's nonterminal (internal) nodes.

`get_terminals`(*order='preorder'*)

Get a list of all of this tree's terminal (leaf) nodes.

`trace`(*start*, *finish*)

List of all clade object between two targets in this tree.

Excluding `start`, including `finish`.

`common_ancestor`(*targets*, **more_targets*)

Most recent common ancestor (clade) of all the given targets.

Edge cases:

- If no target is given, returns self.root
- If 1 target is given, returns the target
- If any target is not found in this tree, raises a ValueError

`count_terminals`()

Count the number of terminal (leaf) nodes within this tree.

`depths`(*unit_branch_lengths=False*)

Create a mapping of tree clades to depths (by branch length).

Parameters

unit_branch_lengths
[bool] If True, count only the number of branches (levels in the tree). By default the distance is the cumulative branch length leading to the clade.

Returns
dict of {clade: depth}, where keys are all of the Clade instances in the tree, and values are the distance from the root to each clade (including terminals).

`distance`(*target1*, *target2=None*)

Calculate the sum of the branch lengths between two targets.

If only one target is specified, the other is the root of this tree.

`is_bifurcating`()

Return True if tree downstream of node is strictly bifurcating.

I.e., all nodes have either 2 or 0 children (internal or external, respectively). The root may have 3 descendents and still be considered part of a bifurcating tree, because it has no ancestor.

is_monophyletic(*terminals*, **more_terminals*)

MRCA of terminals if they comprise a complete subclade, or False.

I.e., there exists a clade such that its terminals are the same set as the given targets.

The given targets must be terminals of the tree.

To match both `Bio.Nexus.Trees` and the other multi-target methods in Bio.Phylo, arguments to this method can be specified either of two ways: (i) as a single list of targets, or (ii) separately specified targets, e.g. is_monophyletic(t1, t2, t3) – but not both.

For convenience, this method returns the common ancestor (MCRA) of the targets if they are monophyletic (instead of the value True), and False otherwise.

Returns
common ancestor if terminals are monophyletic, otherwise False.

is_parent_of(*target=None*, ***kwargs*)

Check if target is a descendent of this tree.

Not required to be a direct descendent.

To check only direct descendents of a clade, simply use list membership testing: `if subclade in clade: ...`

is_preterminal()

Check if all direct descendents are terminal.

total_branch_length()

Calculate the sum of all the branch lengths in this tree.

collapse(*target=None*, ***kwargs*)

Delete target from the tree, relinking its children to its parent.

Returns
the parent clade.

collapse_all(*target=None*, ***kwargs*)

Collapse all the descendents of this tree, leaving only terminals.

Total branch lengths are preserved, i.e. the distance to each terminal stays the same.

For example, this will safely collapse nodes with poor bootstrap support:

```
>>> from Bio import Phylo
>>> tree = Phylo.read('PhyloXML/apaf.xml', 'phyloxml')
>>> print("Total branch length %0.2f" % tree.total_branch_length())
Total branch length 20.44
>>> tree.collapse_all(lambda c: c.confidence is not None and c.confidence < 70)
>>> print("Total branch length %0.2f" % tree.total_branch_length())
Total branch length 21.37
```

This implementation avoids strange side-effects by using level-order traversal and testing all clade properties (versus the target specification) up front. In particular, if a clade meets the target specification in the original tree, it will be collapsed. For example, if the condition is:

```
>>> from Bio import Phylo
>>> tree = Phylo.read('PhyloXML/apaf.xml', 'phyloxml')
>>> print("Total branch length %0.2f" % tree.total_branch_length())
Total branch length 20.44
```

(continues on next page)

(continued from previous page)

```
>>> tree.collapse_all(lambda c: c.branch_length < 0.1)
>>> print("Total branch length %0.2f" % tree.total_branch_length())
Total branch length 21.13
```

Collapsing a clade's parent node adds the parent's branch length to the child, so during the execution of collapse_all, a clade's branch_length may increase. In this implementation, clades are collapsed according to their properties in the original tree, not the properties when tree traversal reaches the clade. (It's easier to debug.) If you want the other behavior (incremental testing), modifying the source code of this function is straightforward.

ladderize(*reverse=False*)

Sort clades in-place according to the number of terminal nodes.

Deepest clades are last by default. Use `reverse=True` to sort clades deepest-to-shallowest.

prune(*target=None*, ***kwargs*)

Prunes a terminal clade from the tree.

If taxon is from a bifurcation, the connecting node will be collapsed and its branch length added to remaining terminal node. This might be no longer be a meaningful value.

Returns
parent clade of the pruned target

split(*n=2*, *branch_length=1.0*)

Generate n (default 2) new descendants.

In a species tree, this is a speciation event.

New clades have the given branch_length and the same name as this clade's root plus an integer suffix (counting from 0). For example, splitting a clade named "A" produces sub-clades named "A0" and "A1". If the clade has no name, the prefix "n" is used for child nodes, e.g. "n0" and "n1".

class Bio.Phylo.BaseTree.**Tree**(*root=None*, *rooted=True*, *id=None*, *name=None*)

Bases: *TreeElement*, *TreeMixin*

A phylogenetic tree, containing global info for the phylogeny.

The structure and node-specific data is accessible through the 'root' clade attached to the Tree instance.

Parameters

root
[Clade] The starting node of the tree. If the tree is rooted, this will usually be the root node.

rooted
[bool] Whether or not the tree is rooted. By default, a tree is assumed to be rooted.

id
[str] The identifier of the tree, if there is one.

name
[str] The name of the tree, in essence a label.

__init__(*root=None*, *rooted=True*, *id=None*, *name=None*)

Initialize parameter for phylogenetic tree.

classmethod from_clade(*clade*, ***kwargs*)

Create a new Tree object given a clade.

Keyword arguments are the usual `Tree` constructor parameters.

classmethod randomized(*taxa*, *branch_length=1.0*, *branch_stdev=None*)

Create a randomized bifurcating tree given a list of taxa.

Parameters
taxa – Either an integer specifying the number of taxa to create (automatically named taxon#), or an iterable of taxon names, as strings.

Returns
a tree of the same type as this class.

property clade

Return first clade in this tree (not itself).

as_phyloxml(***kwargs*)

Convert this tree to a PhyloXML-compatible Phylogeny.

This lets you use the additional annotation types PhyloXML defines, and save this information when you write this tree as 'phyloxml'.

root_with_outgroup(*outgroup_targets*, **more_targets*, *outgroup_branch_length=None*)

Reroot this tree with the outgroup clade containing outgroup_targets.

Operates in-place.

Edge cases:

- If `outgroup == self.root`, no change
- If outgroup is terminal, create new bifurcating root node with a 0-length branch to the outgroup
- If outgroup is internal, use the given outgroup node as the new trifurcating root, keeping branches the same
- If the original root was bifurcating, drop it from the tree, preserving total branch lengths

Parameters
outgroup_branch_length – length of the branch leading to the outgroup after rerooting. If not specified (None), then:

- If the outgroup is an internal node (not a single terminal taxon), then use that node as the new root.
- Otherwise, create a new root node as the parent of the outgroup.

root_at_midpoint()

Root the tree at the midpoint of the two most distant taxa.

This operates in-place, leaving a bifurcating root. The topology of the tree is otherwise retained, though no guarantees are made about the stability of clade/node/taxon ordering.

is_terminal()

Check if the root of this tree is terminal.

__format__(*format_spec*)

Serialize the tree as a string in the specified file format.

This method supports Python's `format` built-in function.

Parameters
format_spec – a lower-case string supported by `Bio.Phylo.write` as an output file format.

format(*fmt=None*)

Serialize the tree as a string in the specified file format.

Parameters

fmt – a lower-case string supported by `Bio.Phylo.write` as an output file format.

__str__() → str

Return a string representation of the entire tree.

Serialize each sub-clade recursively using `repr` to create a summary of the object structure.

__annotations__ = {}

class Bio.Phylo.BaseTree.Clade(*branch_length=None*, *name=None*, *clades=None*, *confidence=None*, *color=None*, *width=None*)

Bases: *TreeElement*, *TreeMixin*

A recursively defined sub-tree.

Parameters

branch_length
[str] The length of the branch leading to the root node of this clade.

name
[str] The clade's name (a label).

clades
[list] Sub-trees rooted directly under this tree's root.

confidence
[number] Support.

color
[BranchColor] The display color of the branch and descendents.

width
[number] The display width of the branch and descendents.

__init__(*branch_length=None*, *name=None*, *clades=None*, *confidence=None*, *color=None*, *width=None*)

Define parameters for the Clade tree.

property root

Allow TreeMixin methods to traverse clades properly.

is_terminal()

Check if this is a terminal (leaf) node.

__getitem__(*index*)

Get clades by index (integer or slice).

__iter__()

Iterate through this tree's direct descendent clades (sub-trees).

__len__()

Return the number of clades directly under the root.

__bool__()

Boolean value of an instance of this class (True).

NB: If this method is not defined, but `__len__` is, then the object is considered true if the result of `__len__()` is nonzero. We want Clade instances to always be considered True.

__str__() → str

Return name of the class instance.

property color

Branch color.

__annotations__ = {}

class Bio.Phylo.BaseTree.**BranchColor**(*red*, *green*, *blue*)

Bases: object

Indicates the color of a clade when rendered graphically.

The color should be interpreted by client code (e.g. visualization programs) as applying to the whole clade, unless overwritten by the color(s) of sub-clades.

Color values must be integers from 0 to 255.

color_names = {'aqua': (0, 255, 255), 'b': (0, 0, 255), 'black': (0, 0, 0), 'blue': (0, 0, 255), 'brown': (165, 42, 42), 'c': (0, 255, 255), 'cyan': (0, 255, 255), 'fuchsia': (255, 0, 255), 'g': (0, 128, 0), 'gold': (255, 215, 0), 'gray': (128, 128, 128), 'green': (0, 128, 0), 'grey': (128, 128, 128), 'k': (0, 0, 0), 'lime': (0, 255, 0), 'm': (255, 0, 255), 'magenta': (255, 0, 255), 'maroon': (128, 0, 0), 'navy': (0, 0, 128), 'olive': (128, 128, 0), 'orange': (255, 165, 0), 'pink': (255, 192, 203), 'purple': (128, 0, 128), 'r': (255, 0, 0), 'red': (255, 0, 0), 'salmon': (250, 128, 114), 'silver': (192, 192, 192), 'tan': (210, 180, 140), 'teal': (0, 128, 128), 'w': (255, 255, 255), 'white': (255, 255, 255), 'y': (255, 255, 0), 'yellow': (255, 255, 0)}

__init__(*red*, *green*, *blue*)

Initialize BranchColor for a tree.

classmethod from_hex(*hexstr*)

Construct a BranchColor object from a hexadecimal string.

The string format is the same style used in HTML and CSS, such as '#FF8000' for an RGB value of (255, 128, 0).

classmethod from_name(*colorname*)

Construct a BranchColor object by the color's name.

to_hex()

Return a 24-bit hexadecimal RGB representation of this color.

The returned string is suitable for use in HTML/CSS, as a color parameter in matplotlib, and perhaps other situations.

Examples

```
>>> bc = BranchColor(12, 200, 100)
>>> bc.to_hex()
'#0cc864'
```

to_rgb()

Return a tuple of RGB values (0 to 255) representing this color.

Examples

```
>>> bc = BranchColor(255, 165, 0)
>>> bc.to_rgb()
(255, 165, 0)
```

__repr__() → str

Preserve the standard RGB order when representing this object.

__str__() → str

Show the color's RGB values.

Bio.Phylo.CDAO module

Classes corresponding to CDAO trees.

See classes in `Bio.Nexus`: Trees.Tree, Trees.NodeData, and Nodes.Chain.

class Bio.Phylo.CDAO.**Tree**(*root=None*, *rooted=False*, *id=None*, *name=None*, *weight=1.0*)

Bases: *Tree*

CDAO Tree object.

__init__(*root=None*, *rooted=False*, *id=None*, *name=None*, *weight=1.0*)

Initialize value of for the CDAO tree object.

__annotations__ = {}

class Bio.Phylo.CDAO.**Clade**(*branch_length=1.0*, *name=None*, *clades=None*, *confidence=None*, *comment=None*)

Bases: *Clade*

CDAO Clade (sub-tree) object.

__init__(*branch_length=1.0*, *name=None*, *clades=None*, *confidence=None*, *comment=None*)

Initialize values for the CDAO Clade object.

__annotations__ = {}

Bio.Phylo.CDAOIO module

I/O function wrappers for the RDF/CDAO file format.

This is an RDF format that conforms to the Comparative Data Analysis Ontology (CDAO). See: http://evolutionaryontology.org/cdao

This module requires the librdf Python bindings (http://www.librdf.org)

The CDAOIO.Parser, in addition to parsing text files, can also parse directly from a triple store that implements the Redland storage interface; similarly, the CDAOIO.Writer can store triples in a triple store instead of serializing them to a file.

Bio.Phylo.CDAOIO.**qUri**(*x*)

Resolve URI for librdf.

Bio.Phylo.CDAOIO.**format_label**(*x*)

Format label for librdf.

Bio.Phylo.CDAOIO.**parse**(*handle*, ***kwargs*)

Iterate over the trees in a CDAO file handle.

Returns
generator of Bio.Phylo.CDAO.Tree objects.

Bio.Phylo.CDAOIO.**write**(*trees*, *handle*, *plain=False*, ***kwargs*)

Write a trees in CDAO format to the given file handle.

Returns
number of trees written.

class Bio.Phylo.CDAOIO.**Parser**(*handle=None*)

Bases: `object`

Parse a CDAO tree given a file handle.

__init__(*handle=None*)

Initialize CDAO tree parser.

classmethod from_string(*treetext*)

Instantiate the class from the given string.

parse(***kwargs*)

Parse the text stream this object was initialized with.

parse_handle_to_graph(*rooted=False*, *parse_format='turtle'*, *context=None*, ***kwargs*)

Parse self.handle into RDF model self.model.

parse_graph(*graph=None*, *context=None*)

Iterate over RDF model yielding CDAO.Tree instances.

new_clade(*node*)

Return a CDAO.Clade object for a given named node.

get_node_info(*graph*, *context=None*)

Create a dictionary containing information about all nodes in the tree.

parse_children(*node*)

Traverse the tree to create a nested clade structure.

Return a CDAO.Clade, and calls itself recursively for each child, traversing the entire tree and creating a nested structure of CDAO.Clade objects.

class Bio.Phylo.CDAOIO.**Writer**(*trees*)

Bases: `object`

Based on the writer in Bio.Nexus.Trees (str, to_string).

prefixes = {'cdao': 'http://purl.obolibrary.org/obo/cdao.owl#', 'obo': 'http://purl.obolibrary.org/obo/', 'owl': 'http://www.w3.org/2002/07/owl#', 'rdf': 'http://www.w3.org/1999/02/22-rdf-syntax-ns#', 'rdfs': 'http://www.w3.org/2000/01/rdf-schema#'}

__init__(*trees*)

Initialize parameters for writing a CDAO tree.

write(*handle*, *tree_uri=''*, *record_complete_ancestry=False*, *rooted=False*, ***kwargs*)

Write this instance's trees to a file handle.

add_stmt_to_handle(*handle*, *stmt*)

Add URI prefix to handle.

process_clade(*clade*, *parent=None*, *root=False*)

Recursively generate triples describing a tree of clades.

Bio.Phylo.Consensus module

Classes and methods for finding consensus trees.

This module contains a `_BitString` class to assist the consensus tree searching and some common consensus algorithms such as strict, majority rule and adam consensus.

Bio.Phylo.Consensus.**strict_consensus**(*trees*)

Search strict consensus tree from multiple trees.

Parameters

trees
[iterable] iterable of trees to produce consensus tree.

Bio.Phylo.Consensus.**majority_consensus**(*trees*, *cutoff=0*)

Search majority rule consensus tree from multiple trees.

This is a extend majority rule method, which means the you can set any cutoff between 0 ~ 1 instead of 0.5. The default value of cutoff is 0 to create a relaxed binary consensus tree in any condition (as long as one of the provided trees is a binary tree). The branch length of each consensus clade in the result consensus tree is the average length of all counts for that clade.

Parameters

trees
[iterable] iterable of trees to produce consensus tree.

Bio.Phylo.Consensus.**adam_consensus**(*trees*)

Search Adam Consensus tree from multiple trees.

Parameters

trees
[list] list of trees to produce consensus tree.

Bio.Phylo.Consensus.**get_support**(*target_tree*, *trees*, *len_trees=None*)

Calculate branch support for a target tree given bootstrap replicate trees.

Parameters

target_tree
[Tree] tree to calculate branch support for.

trees
[iterable] iterable of trees used to calculate branch support.

len_trees
[int] optional count of replicates in trees. len_trees must be provided when len(trees) is not a valid operation.

Bio.Phylo.Consensus.**bootstrap**(*msa*, *times*)

Generate bootstrap replicates from a multiple sequence alignment (OBSOLETE).

Parameters

msa
[MultipleSeqAlignment] multiple sequence alignment to generate replicates.

times
[int] number of bootstrap times.

Bio.Phylo.Consensus.**bootstrap_trees**(*alignment*, *times*, *tree_constructor*)

Generate bootstrap replicate trees from a multiple sequence alignment.

Parameters

alignment
[Alignment or MultipleSeqAlignment object] multiple sequence alignment to generate replicates.

times
[int] number of bootstrap times.

tree_constructor
[TreeConstructor] tree constructor to be used to build trees.

Bio.Phylo.Consensus.**bootstrap_consensus**(*alignment*, *times*, *tree_constructor*, *consensus*)

Consensus tree of a series of bootstrap trees for a multiple sequence alignment.

Parameters

alignment
[Alignment or MultipleSeqAlignment object] Multiple sequence alignment to generate replicates.

times
[int] Number of bootstrap times.

tree_constructor
[TreeConstructor] Tree constructor to be used to build trees.

consensus
[function] Consensus method in this module: `strict_consensus`, `majority_consensus`, `adam_consensus`.

Bio.Phylo.NeXML module

Classes corresponding to NeXML trees.

See classes in `Bio.Nexus`: Trees.Tree, Trees.NodeData, and Nodes.Chain.

class Bio.Phylo.NeXML.**Tree**(*root=None*, *rooted=False*, *id=None*, *name=None*, *weight=1.0*)

Bases: *Tree*

NeXML Tree object.

__init__(*root=None*, *rooted=False*, *id=None*, *name=None*, *weight=1.0*)

Instantiate a NeXML tree object with the given parameters.

__annotations__ = {}

class Bio.Phylo.NeXML.**Clade**(*branch_length=1.0*, *name=None*, *clades=None*, *confidence=None*, *comment=None*, ***kwargs*)

Bases: *Clade*

NeXML Clade (sub-tree) object.

__init__(*branch_length=1.0*, *name=None*, *clades=None*, *confidence=None*, *comment=None*, ***kwargs*)

Initialize parameters for NeXML Clade object.

__annotations__ = {}

Bio.Phylo.NeXMLIO module

I/O function wrappers for the NeXML file format.

See: http://www.nexml.org

Bio.Phylo.NeXMLIO.**qUri**(*s*)

Given a prefixed URI, return the full URI.

Bio.Phylo.NeXMLIO.**cdao_to_obo**(*s*)

Optionally converts a CDAO-prefixed URI into an OBO-prefixed URI.

Bio.Phylo.NeXMLIO.**matches**(*s*)

Check for matches in both CDAO and OBO namespaces.

exception Bio.Phylo.NeXMLIO.**NeXMLError**

Bases: Exception

Exception raised when NeXML object construction cannot continue.

Bio.Phylo.NeXMLIO.**parse**(*handle*, ***kwargs*)

Iterate over the trees in a NeXML file handle.

Returns
generator of Bio.Phylo.NeXML.Tree objects.

Bio.Phylo.NeXMLIO.**write**(*trees*, *handle*, *plain=False*, ***kwargs*)

Write a trees in NeXML format to the given file handle.

Returns
number of trees written.

class Bio.Phylo.NeXMLIO.**Parser**(*handle*)

Bases: object

Parse a NeXML tree given a file handle.

Based on the parser in Bio.Nexus.Trees.

__init__(*handle*)

Initialize parameters for NeXML file parser.

classmethod **from_string**(*treetext*)

Convert file handle to StringIO object.

add_annotation(*node_dict*, *meta_node*)

Add annotations for the NeXML parser.

parse(*values_are_confidence=False*, *rooted=False*)

Parse the text stream this object was initialized with.

class Bio.Phylo.NeXMLIO.**Writer**(*trees*)

Bases: `object`

Based on the writer in Bio.Nexus.Trees (str, to_string).

__init__(*trees*)

Initialize parameters for NeXML writer.

new_label(*obj_type*)

Create new labels for the NeXML writer.

write(*handle*, *cdao_to_obo=True*, ***kwargs*)

Write this instance's trees to a file handle.

Bio.Phylo.Newick module

Classes corresponding to Newick trees, also used for Nexus trees.

See classes in `Bio.Nexus`: Trees.Tree, Trees.NodeData, and Nodes.Chain.

class Bio.Phylo.Newick.**Tree**(*root=None*, *rooted=False*, *id=None*, *name=None*, *weight=1.0*)

Bases: *Tree*

Newick Tree object.

__init__(*root=None*, *rooted=False*, *id=None*, *name=None*, *weight=1.0*)

Initialize parameters for a Newick tree object.

__annotations__ = {}

class Bio.Phylo.Newick.**Clade**(*branch_length=None*, *name=None*, *clades=None*, *confidence=None*, *comment=None*)

Bases: *Clade*

Newick Clade (sub-tree) object.

__init__(*branch_length=None*, *name=None*, *clades=None*, *confidence=None*, *comment=None*)

Initialize parameters for a Newick Clade object.

__annotations__ = {}

Bio.Phylo.NewickIO module

I/O function wrappers for the Newick file format.

See: http://evolution.genetics.washington.edu/phylip/newick_doc.html

exception Bio.Phylo.NewickIO.**NewickError**

Bases: `Exception`

Exception raised when Newick object construction cannot continue.

Bio.Phylo.NewickIO.**parse**(*handle*, ***kwargs*)

Iterate over the trees in a Newick file handle.

Returns
generator of Bio.Phylo.Newick.Tree objects.

Bio.Phylo.NewickIO.**write**(*trees*, *handle*, *plain=False*, ***kwargs*)

Write a trees in Newick format to the given file handle.

Returns
number of trees written.

class Bio.Phylo.NewickIO.**Parser**(*handle*)

Bases: object

Parse a Newick tree given a file handle.

Based on the parser in Bio.Nexus.Trees.

__init__(*handle*)

Initialize file handle for the Newick Tree.

classmethod from_string(*treetext*)

Instantiate the Newick Tree class from the given string.

parse(*values_are_confidence=False*, *comments_are_confidence=False*, *rooted=False*)

Parse the text stream this object was initialized with.

new_clade(*parent=None*)

Return new Newick.Clade, optionally with temporary reference to parent.

process_clade(*clade*)

Remove node's parent and return it. Final processing of parsed clade.

class Bio.Phylo.NewickIO.**Writer**(*trees*)

Bases: object

Based on the writer in Bio.Nexus.Trees (str, to_string).

__init__(*trees*)

Initialize parameter for Tree Writer object.

write(*handle*, ***kwargs*)

Write this instance's trees to a file handle.

to_strings(*confidence_as_branch_length=False*, *branch_length_only=False*, *plain=False*, *plain_newick=True*, *ladderize=None*, *max_confidence=1.0*, *format_confidence='%1.2f'*, *format_branch_length='%1.5f'*)

Return an iterable of PAUP-compatible tree lines.

Bio.Phylo.NexusIO module

I/O function wrappers for `Bio.Nexus` trees.

`Bio.Phylo.NexusIO.`**`parse`**(*handle*)

Parse the trees in a Nexus file.

Uses the old Nexus.Trees parser to extract the trees, converts them back to plain Newick trees, and feeds those strings through the new Newick parser. This way we don't have to modify the Nexus module yet. (Perhaps we'll eventually change Nexus to use the new NewickIO parser directly.)

`Bio.Phylo.NexusIO.`**`write`**(*obj*, *handle*, ***kwargs*)

Write a new Nexus file containing the given trees.

Uses a simple Nexus template and the NewickIO writer to serialize just the trees and minimal supporting info needed for a valid Nexus file.

Bio.Phylo.PhyloXML module

Classes corresponding to phyloXML elements.

See Also

Official specification:
http://phyloxml.org/

Journal article:
Han and Zmasek (2009), https://doi.org/10.1186/1471-2105-10-356

exception `Bio.Phylo.PhyloXML.`**`PhyloXMLWarning`**

Bases: *BiopythonWarning*

Warning for non-compliance with the phyloXML specification.

`__annotations__ = {}`

class `Bio.Phylo.PhyloXML.`**`PhyloElement`**

Bases: *TreeElement*

Base class for all PhyloXML objects.

`__annotations__ = {}`

class `Bio.Phylo.PhyloXML.`**`Phyloxml`**(*attributes*, *phylogenies=None*, *other=None*)

Bases: *PhyloElement*

Root node of the PhyloXML document.

Contains an arbitrary number of Phylogeny elements, possibly followed by elements from other namespaces.

Parameters

attributes
[dict] (XML namespace definitions)

phylogenies
[list] The phylogenetic trees

other
[list] Arbitrary non-phyloXML elements, if any

`__init__`(*attributes*, *phylogenies=None*, *other=None*)

Initialize parameters for PhyloXML object.

`__getitem__`(*index*)

Get a phylogeny by index or name.

`__iter__`()

Iterate through the phylogenetic trees in this object.

`__len__`()

Return the number of phylogenetic trees in this object.

`__str__`()

Return name of phylogenies in the object.

`__annotations__ = {}`

class Bio.Phylo.PhyloXML.**Other**(*tag*, *namespace=None*, *attributes=None*, *value=None*, *children=None*)

Bases: *PhyloElement*

Container for non-phyloXML elements in the tree.

Usually, an Other object will have either a 'value' or a non-empty list of 'children', but not both. This is not enforced here, though.

Parameters

tag
[string] local tag for the XML node

namespace
[string] XML namespace for the node – should not be the default phyloXML namespace.

attributes
[dict of strings] attributes on the XML node

value
[string] text contained directly within this XML node

children
[list] child nodes, if any (also `Other` instances)

`__init__`(*tag*, *namespace=None*, *attributes=None*, *value=None*, *children=None*)

Initialize values for non-phyloXML elements.

`__iter__`()

Iterate through the children of this object (if any).

`__annotations__ = {}`

class Bio.Phylo.PhyloXML.**Phylogeny**(*root=None*, *rooted=True*, *rerootable=None*, *branch_length_unit=None*, *type=None*, *name=None*, *id=None*, *description=None*, *date=None*, *confidences=None*, *clade_relations=None*, *sequence_relations=None*, *properties=None*, *other=None*)

Bases: *PhyloElement*, *Tree*

A phylogenetic tree.

Parameters

root
[Clade] the root node/clade of this tree

rooted
[bool] True if this tree is rooted

rerootable
[bool] True if this tree is rerootable

branch_length_unit
[string] unit for branch_length values on clades

name
[string] identifier for this tree, not required to be unique

id
[Id] unique identifier for this tree

description
[string] plain-text description

date
[Date] date for the root node of this tree

confidences
[list] Confidence objects for this tree

clade_relations
[list] CladeRelation objects

sequence_relations
[list] SequenceRelation objects

properties
[list] Property objects

other
[list] non-phyloXML elements (type `Other`)

`__init__`(*root=None*, *rooted=True*, *rerootable=None*, *branch_length_unit=None*, *type=None*, *name=None*, *id=None*, *description=None*, *date=None*, *confidences=None*, *clade_relations=None*, *sequence_relations=None*, *properties=None*, *other=None*)

Initialize values for phylogenetic tree object.

`classmethod from_tree`(*tree*, ***kwargs*)

Create a new Phylogeny given a Tree (from Newick/Nexus or BaseTree).

Keyword arguments are the usual `Phylogeny` constructor parameters.

`classmethod from_clade`(*clade*, ***kwargs*)

Create a new Phylogeny given a Newick or BaseTree Clade object.

Keyword arguments are the usual `PhyloXML.Clade` constructor parameters.

`as_phyloxml`()

Return this tree, a PhyloXML-compatible Phylogeny object.

Overrides the `BaseTree` method.

`to_phyloxml_container`(***kwargs*)

Create a new Phyloxml object containing just this phylogeny.

`to_alignment()`

Construct a MultipleSeqAlignment from the aligned sequences in this tree.

`property alignment`

Construct an Alignment object from the aligned sequences in this tree.

`property confidence`

Equivalent to self.confidences[0] if there is only 1 value (PRIVATE).

See Also: `Clade.confidence`, `Clade.taxonomy`

`__annotations__ = {}`

`class Bio.Phylo.PhyloXML.Clade(`*branch_length=None*, *id_source=None*, *name=None*, *width=None*, *color=None*, *node_id=None*, *events=None*, *binary_characters=None*, *date=None*, *confidences=None*, *taxonomies=None*, *sequences=None*, *distributions=None*, *references=None*, *properties=None*, *clades=None*, *other=None*`)`

Bases: *PhyloElement*, *Clade*

Describes a branch of the current phylogenetic tree.

Used recursively, describes the topology of a phylogenetic tree.

Both `color` and `width` elements should be interpreted by client code as applying to the whole clade, including all descendents, unless overwritten in-sub clades. This module doesn't automatically assign these attributes to sub-clades to achieve this cascade – and neither should you.

Parameters

branch_length
parent branch length of this clade

id_source
link other elements to a clade (on the xml-level)

name
[string] short label for this clade

confidences
[list of Confidence objects] used to indicate the support for a clade/parent branch.

width
[float] branch width for this clade (including branch from parent)

color
[BranchColor] color used for graphical display of this clade

node_id
unique identifier for the root node of this clade

taxonomies
[list] Taxonomy objects

sequences
[list] Sequence objects

events
[Events] describe such events as gene-duplications at the root node/parent branch of this clade

binary_characters
[BinaryCharacters] binary characters

distributions
[list of Distribution objects] distribution(s) of this clade

date
[Date] a date for the root node of this clade

references
[list] Reference objects

properties
[list] Property objects

clades
[list Clade objects] Sub-clades

other
[list of Other objects] non-phyloXML objects

`__init__`(*branch_length=None*, *id_source=None*, *name=None*, *width=None*, *color=None*, *node_id=None*, *events=None*, *binary_characters=None*, *date=None*, *confidences=None*, *taxonomies=None*, *sequences=None*, *distributions=None*, *references=None*, *properties=None*, *clades=None*, *other=None*)

Initialize value for the Clade object.

`classmethod from_clade`(*clade*, ***kwargs*)

Create a new PhyloXML Clade from a Newick or BaseTree Clade object.

Keyword arguments are the usual PhyloXML Clade constructor parameters.

`to_phylogeny`(***kwargs*)

Create a new phylogeny containing just this clade.

`property confidence`

Return confidence values (PRIVATE).

`property taxonomy`

Get taxonomy list for the clade (PRIVATE).

`__annotations__ = {}`

`class` Bio.Phylo.PhyloXML.**BranchColor**(**args*, ***kwargs*)

Bases: *PhyloElement*, *BranchColor*

Manage Tree branch's color.

`__init__`(**args*, ***kwargs*)

Initialize parameters for the BranchColor object.

`__annotations__ = {}`

`class` Bio.Phylo.PhyloXML.**Accession**(*value*, *source*)

Bases: *PhyloElement*

Captures the local part in a sequence identifier.

Example: In `UniProtKB:P17304`, the Accession instance attribute `value` is 'P17304' and the `source` attribute is 'UniProtKB'.

`__init__`(*value*, *source*)

Initialize value for Accession object.

__str__()

Show the class name and an identifying attribute.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**Annotation**(*ref=None*, *source=None*, *evidence=None*, *type=None*, *desc=None*, *confidence=None*, *uri=None*, *properties=None*)

Bases: *PhyloElement*

The annotation of a molecular sequence.

It is recommended to annotate by using the optional 'ref' attribute.

Parameters

ref
[string] reference string, e.g. 'GO:0008270', 'KEGG:Tetrachloroethene degradation', 'EC:1.1.1.1'

source
[string] plain-text source for this annotation

evidence
[str] describe evidence as free text (e.g. 'experimental')

desc
[string] free text description

confidence
[Confidence] state the type and value of support (type Confidence)

properties
[list] typed and referenced annotations from external resources

uri
[Uri] link

re_ref = re.compile('[a-zA-Z0-9_]+:[a-zA-Z0-9_\\.\\-\\s]+')

__init__(*ref=None*, *source=None*, *evidence=None*, *type=None*, *desc=None*, *confidence=None*, *uri=None*, *properties=None*)

Initialize value for the Annotation object.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**BinaryCharacters**(*type=None*, *gained_count=None*, *lost_count=None*, *present_count=None*, *absent_count=None*, *gained=None*, *lost=None*, *present=None*, *absent=None*)

Bases: *PhyloElement*

Binary characters at the root of a clade.

The names and/or counts of binary characters present, gained, and lost at the root of a clade.

__init__(*type=None*, *gained_count=None*, *lost_count=None*, *present_count=None*, *absent_count=None*, *gained=None*, *lost=None*, *present=None*, *absent=None*)

Initialize values for the BinaryCharacters object.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**CladeRelation**(*type*, *id_ref_0*, *id_ref_1*, *distance=None*, *confidence=None*)

Bases: *PhyloElement*

Expresses a typed relationship between two clades.

For example, this could be used to describe multiple parents of a clade.

__init__(*type*, *id_ref_0*, *id_ref_1*, *distance=None*, *confidence=None*)

Initialize values for the CladeRelation object.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**Confidence**(*value*, *type='unknown'*)

Bases: float, *PhyloElement*

A general purpose confidence element.

For example, this can be used to express the bootstrap support value of a clade (in which case the type attribute is 'bootstrap').

Parameters

value
[float] confidence value

type
[string] label for the type of confidence, e.g. 'bootstrap'

static __new__(*cls*, *value*, *type='unknown'*)

Create and return a Confidence object with the specified value and type.

property value

Return the float value of the Confidence object.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**Date**(*value=None*, *unit=None*, *desc=None*, *minimum=None*, *maximum=None*)

Bases: *PhyloElement*

A date associated with a clade/node.

Its value can be numerical by using the 'value' element and/or free text with the 'desc' element' (e.g. 'Silurian'). If a numerical value is used, it is recommended to employ the 'unit' attribute.

Parameters

unit
[string] type of numerical value (e.g. 'mya' for 'million years ago')

value
[float] the date value

desc
[string] plain-text description of the date

minimum
[float] lower bound on the date value

maximum
[float] upper bound on the date value

__init__(*value=None*, *unit=None*, *desc=None*, *minimum=None*, *maximum=None*)

Initialize values of the Date object.

__str__()

Show the class name and the human-readable date.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**Distribution**(*desc=None*, *points=None*, *polygons=None*)

Bases: *PhyloElement*

Geographic distribution of the items of a clade (species, sequences).

Intended for phylogeographic applications.

Parameters

desc
[string] free-text description of the location

points
[list of `Point` objects] coordinates (similar to the 'Point' element in Google's KML format)

polygons
[list of `Polygon` objects] coordinate sets defining geographic regions

__init__(*desc=None*, *points=None*, *polygons=None*)

Initialize values of Distribution object.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**DomainArchitecture**(*length=None*, *domains=None*)

Bases: *PhyloElement*

Domain architecture of a protein.

Parameters

length
[int] total length of the protein sequence

domains
[list ProteinDomain objects] the domains within this protein

__init__(*length=None*, *domains=None*)

Initialize values of the DomainArchitecture object.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**Events**(*type=None*, *duplications=None*, *speciations=None*, *losses=None*, *confidence=None*)

Bases: *PhyloElement*

Events at the root node of a clade (e.g. one gene duplication).

All attributes are set to None by default, but this object can also be treated as a dictionary, in which case None values are treated as missing keys and deleting a key resets that attribute's value back to None.

ok_type = {'fusion', 'mixed', 'other', 'speciation_or_duplication', 'transfer', 'unassigned'}

__init__(*type=None*, *duplications=None*, *speciations=None*, *losses=None*, *confidence=None*)
Initialize values of the Events object.

items()
Return Event's items.

keys()
Return Event's keys.

values()
Return values from a key-value pair in an Events dict.

__len__()
Return number of Events.

__getitem__(*key*)
Get value of Event with the given key.

__setitem__(*key*, *val*)
Add item to Event dict.

__delitem__(*key*)
Delete Event with given key.

__iter__()
Iterate over the keys present in a Events dict.

__contains__(*key*)
Return True if Event dict contains key.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**Id**(*value*, *provider=None*)
Bases: *PhyloElement*

A general-purpose identifier element.

Allows to indicate the provider (or authority) of an identifier, e.g. NCBI, along with the value itself.

__init__(*value*, *provider=None*)
Initialize values for the identifier object.

__str__()
Return identifier as a string.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**MolSeq**(*value*, *is_aligned=None*)
Bases: *PhyloElement*

Store a molecular sequence.

Parameters

value
[string] the sequence itself

is_aligned
[bool] True if this sequence is aligned with the others (usually meaning all aligned seqs are the same length and gaps may be present)

re_value = re.compile('[a-zA-Z\\.\\-\\?*_]+')

__init__(*value*, *is_aligned=None*)

Initialize parameters for the MolSeq object.

__str__()

Return the value of the Molecular Sequence object.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**Point**(*geodetic_datum*, *lat*, *long*, *alt=None*, *alt_unit=None*)

Bases: *PhyloElement*

Geographic coordinates of a point, with an optional altitude.

Used by element 'Distribution'.

Parameters

geodetic_datum
[string, required] the geodetic datum (also called 'map datum'). For example, Google's KML uses 'WGS84'.

lat
[numeric] latitude

long
[numeric] longitude

alt
[numeric] altitude

alt_unit
[string] unit for the altitude (e.g. 'meter')

__init__(*geodetic_datum*, *lat*, *long*, *alt=None*, *alt_unit=None*)

Initialize value for the Point object.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**Polygon**(*points=None*)

Bases: *PhyloElement*

A polygon defined by a list of 'Points' (used by element 'Distribution').

Parameters
points – list of 3 or more points representing vertices.

__init__(*points=None*)

Initialize value for the Polygon object.

__str__()

Return list of points as a string.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**Property**(*value*, *ref*, *applies_to*, *datatype*, *unit=None*, *id_ref=None*)

Bases: *PhyloElement*

A typed and referenced property from an external resources.

Can be attached to `Phylogeny`, `Clade`, and `Annotation` objects.

Parameters

value
[string] the value of the property

ref
[string] reference to an external resource, e.g. "NOAA:depth"

applies_to
[string] indicates the item to which a property applies to (e.g. 'node' for the parent node of a clade, 'parent_branch' for the parent branch of a clade, or just 'clade').

datatype
[string] the type of a property; limited to xsd-datatypes (e.g. 'xsd:string', 'xsd:boolean', 'xsd:integer', 'xsd:decimal', 'xsd:float', 'xsd:double', 'xsd:date', 'xsd:anyURI').

unit
[string (optional)] the unit of the property, e.g. "METRIC:m"

id_ref
[Id (optional)] allows to attached a property specifically to one element (on the xml-level)

```
re_ref = re.compile('[a-zA-Z0-9_]+:[a-zA-Z0-9_\\.\\-\\s]+')
```

```
ok_applies_to = {'annotation', 'clade', 'node', 'other', 'parent_branch',
'phylogeny'}
```

```
ok_datatype = {'xsd:anyURI', 'xsd:base64Binary', 'xsd:boolean', 'xsd:byte',
'xsd:date', 'xsd:dateTime', 'xsd:decimal', 'xsd:double', 'xsd:duration',
'xsd:float', 'xsd:gDay', 'xsd:gMonth', 'xsd:gMonthDay', 'xsd:gYear',
'xsd:gYearMonth', 'xsd:hexBinary', 'xsd:int', 'xsd:integer', 'xsd:long',
'xsd:negativeInteger', 'xsd:nonNegativeInteger', 'xsd:nonPositiveInteger',
'xsd:normalizedString', 'xsd:positiveInteger', 'xsd:short', 'xsd:string',
'xsd:time', 'xsd:token', 'xsd:unsignedByte', 'xsd:unsignedInt', 'xsd:unsignedLong',
'xsd:unsignedShort'}
```

__init__(*value*, *ref*, *applies_to*, *datatype*, *unit=None*, *id_ref=None*)

Initialize value for the Property object.

```
__annotations__ = {}
```

class Bio.Phylo.PhyloXML.**ProteinDomain**(*value*, *start*, *end*, *confidence=None*, *id=None*)

Bases: *PhyloElement*

Represents an individual domain in a domain architecture.

The locations use 0-based indexing, as most Python objects including SeqFeature do, rather than the usual biological convention starting at 1. This means the start and end attributes can be used directly as slice indexes on Seq objects.

Parameters

start
[non-negative integer] start of the domain on the sequence, using 0-based indexing

end
[non-negative integer] end of the domain on the sequence

confidence
[float] can be used to store e.g. E-values

id
[string] unique identifier/name

__init__(*value*, *start*, *end*, *confidence=None*, *id=None*)
Initialize value for a ProteinDomain object.

classmethod from_seqfeature(*feat*)
Create ProteinDomain object from SeqFeature.

to_seqfeature()
Create a SeqFeature from the ProteinDomain Object.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**Reference**(*doi=None*, *desc=None*)
Bases: *PhyloElement*

Literature reference for a clade.

NB: Whenever possible, use the `doi` attribute instead of the free-text `desc` element.

re_doi = re.compile('[a-zA-Z0-9_\\.]+/[a-zA-Z0-9_\\.]+')

__init__(*doi=None*, *desc=None*)
Initialize elements of the Reference class object.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**Sequence**(*type=None*, *id_ref=None*, *id_source=None*, *symbol=None*, *accession=None*, *name=None*, *location=None*, *mol_seq=None*, *uri=None*, *domain_architecture=None*, *annotations=None*, *other=None*)
Bases: *PhyloElement*

A molecular sequence (Protein, DNA, RNA) associated with a node.

One intended use for `id_ref` is to link a sequence to a taxonomy (via the taxonomy's `id_source`) in case of multiple sequences and taxonomies per node.

Parameters

type
[{'dna', 'rna', 'protein'}] type of molecule this sequence represents

id_ref
[string] reference to another resource

id_source
[string] source for the reference

symbol
[string] short symbol of the sequence, e.g. 'ACTM' (max. 10 chars)

accession
[Accession] accession code for this sequence.

name
[string] full name of the sequence, e.g. 'muscle Actin'

location
location of a sequence on a genome/chromosome.

mol_seq
[MolSeq] the molecular sequence itself

uri
[Uri] link

annotations
[list of Annotation objects] annotations on this sequence

domain_architecture
[DomainArchitecture] protein domains on this sequence

other
[list of Other objects] non-phyloXML elements

types = {'dna', 'protein', 'rna'}

re_symbol = re.compile('\\S{1,10}')

__init__(*type=None*, *id_ref=None*, *id_source=None*, *symbol=None*, *accession=None*, *name=None*, *location=None*, *mol_seq=None*, *uri=None*, *domain_architecture=None*, *annotations=None*, *other=None*)

Initialize value for a Sequence object.

classmethod from_seqrecord(*record*, *is_aligned=None*)

Create a new PhyloXML Sequence from a SeqRecord object.

to_seqrecord()

Create a SeqRecord object from this Sequence instance.

The seqrecord.annotations dictionary is packed like so:

```
{ # Sequence attributes with no SeqRecord equivalent:
  'id_ref': self.id_ref,
  'id_source': self.id_source,
  'location': self.location,
  'uri': { 'value': self.uri.value,
                  'desc': self.uri.desc,
                  'type': self.uri.type },
  # Sequence.annotations attribute (list of Annotations)
  'annotations': [{'ref': ann.ref,
                   'source': ann.source,
                   'evidence': ann.evidence,
                   'type': ann.type,
                   'confidence': [ann.confidence.value,
                                  ann.confidence.type],
                   'properties': [{'value': prop.value,
                                   'ref': prop.ref,
                                   'applies_to': prop.applies_to,
                                   'datatype': prop.datatype,
                                   'unit': prop.unit,
                                   'id_ref': prop.id_ref}
                                  for prop in ann.properties],
                  } for ann in self.annotations],
}
```

__annotations__ = {}

class Bio.Phylo.PhyloXML.**SequenceRelation**(*type*, *id_ref_0*, *id_ref_1*, *distance=None*, *confidence=None*)

Bases: *PhyloElement*

Express a typed relationship between two sequences.

For example, this could be used to describe an orthology (in which case attribute 'type' is 'orthology').

Parameters

id_ref_0
[Id] first sequence reference identifier

id_ref_1
[Id] second sequence reference identifier

distance
[float] distance between the two sequences

type
[restricted string] describe the type of relationship

confidence
[Confidence] confidence value for this relation

ok_type = {'one_to_one_orthology', 'orthology', 'other', 'paralogy', 'super_orthology', 'ultra_paralogy', 'unknown', 'xenology'}

__init__(*type*, *id_ref_0*, *id_ref_1*, *distance=None*, *confidence=None*)

Initialize the class.

__annotations__ = {}

class Bio.Phylo.PhyloXML.**Taxonomy**(*id_source=None*, *id=None*, *code=None*, *scientific_name=None*, *authority=None*, *rank=None*, *uri=None*, *common_names=None*, *synonyms=None*, *other=None*)

Bases: *PhyloElement*

Describe taxonomic information for a clade.

Parameters

id_source
[Id] link other elements to a taxonomy (on the XML level)

id
[Id] unique identifier of a taxon, e.g. Id('6500', provider='ncbi_taxonomy') for the California sea hare

code
[restricted string] store UniProt/Swiss-Prot style organism codes, e.g. 'APLCA' for the California sea hare 'Aplysia californica'

scientific_name
[string] the standard scientific name for this organism, e.g. 'Aplysia californica' for the California sea hare

authority
[string] keep the authority, such as 'J. G. Cooper, 1863', associated with the 'scientific_name'

common_names
[list of strings] common names for this organism

synonyms
[list of strings] synonyms for this taxon?

rank
[restricted string] taxonomic rank

uri
[Uri] link

other
[list of Other objects] non-phyloXML elements

```
re_code = re.compile('[a-zA-Z0-9_]{2,10}')
```

```
ok_rank = {'branch', 'class', 'cohort', 'cultivar', 'division', 'domain', 'family',
'form', 'genus', 'infraclass', 'infracohort', 'infradivision', 'infrakingdom',
'infralegion', 'infraphylum', 'infratribe', 'kingdom', 'legion', 'microphylum',
'order', 'other', 'phylum', 'species', 'subclass', 'subcohort', 'subdivision',
'subfamily', 'subform', 'subgenus', 'subkingdom', 'sublegion', 'suborder',
'subphylum', 'subspecies', 'subtribe', 'subvariety', 'superclass', 'supercohort',
'superdivision', 'superfamily', 'superlegion', 'superorder', 'superphylum',
'superspecies', 'supertribe', 'tribe', 'unknown', 'variety'}
```

`__init__`(*id_source=None*, *id=None*, *code=None*, *scientific_name=None*, *authority=None*, *rank=None*, *uri=None*, *common_names=None*, *synonyms=None*, *other=None*)

Initialize the class.

`__str__`()

Show the class name and an identifying attribute.

```
__annotations__ = {}
```

class Bio.Phylo.PhyloXML.**Uri**(*value*, *desc=None*, *type=None*)

Bases: *PhyloElement*

A uniform resource identifier.

In general, this is expected to be an URL (for example, to link to an image on a website, in which case the `type` attribute might be 'image' and `desc` might be 'image of a California sea hare').

`__init__`(*value*, *desc=None*, *type=None*)

Initialize the class.

`__str__`()

Return string representation of Uri.

```
__annotations__ = {}
```

Bio.Phylo.PhyloXMLIO module

PhyloXML reader/parser, writer, and associated functions.

Instantiates tree elements from a parsed PhyloXML file, and constructs an XML file from a `Bio.Phylo.PhyloXML` object.

About capitalization:

- phyloXML means the file format specification
- PhyloXML means the Biopython module `Bio.Phylo.PhyloXML` and its classes

- Phyloxml means the top-level class used by PhyloXMLIO.read (but not Bio.Phylo.read!), containing a list of Phylogenies (objects derived from BaseTree.Tree)

exception Bio.Phylo.PhyloXMLIO.**PhyloXMLError**

Bases: Exception

Exception raised when PhyloXML object construction cannot continue.

XML syntax errors will be found and raised by the underlying ElementTree module; this exception is for valid XML that breaks the phyloXML specification.

Bio.Phylo.PhyloXMLIO.**read**(*file*)

Parse a phyloXML file or stream and build a tree of Biopython objects.

The children of the root node are phylogenies and possibly other arbitrary (non-phyloXML) objects.

Returns
a single Bio.Phylo.PhyloXML.Phyloxml object.

Bio.Phylo.PhyloXMLIO.**parse**(*file*)

Iterate over the phylogenetic trees in a phyloXML file.

This ignores any additional data stored at the top level, but may be more memory-efficient than the read function.

Returns
a generator of Bio.Phylo.PhyloXML.Phylogeny objects.

Bio.Phylo.PhyloXMLIO.**write**(*obj*, *file*, *encoding=DEFAULT_ENCODING*, *indent=True*)

Write a phyloXML file.

Parameters

obj
an instance of Phyloxml, Phylogeny or BaseTree.Tree, or an iterable of either of the latter two. The object will be converted to a Phyloxml object before serialization.

file
either an open handle or a file name.

class Bio.Phylo.PhyloXMLIO.**Parser**(*file*)

Bases: object

Methods for parsing all phyloXML nodes from an XML stream.

To minimize memory use, the tree of ElementTree parsing events is cleared after completing each phylogeny, clade, and top-level 'other' element. Elements below the clade level are kept in memory until parsing of the current clade is finished – this shouldn't be a problem because clade is the only recursive element, and non-clade nodes below this level are of bounded size.

__init__(*file*)

Initialize the class.

read()

Parse the phyloXML file and create a single Phyloxml object.

parse()

Parse the phyloXML file incrementally and return each phylogeny.

other(*elem*, *namespace*, *localtag*)

Create an Other object, a non-phyloXML element.

accession(*elem*)

Create accession object.

annotation(*elem*)

Create annotation object.

binary_characters(*elem*)

Create binary characters object.

clade_relation(*elem*)

Create clade relationship object.

color(*elem*)

Create branch color object.

confidence(*elem*)

Create confidence object.

date(*elem*)

Create date object.

distribution(*elem*)

Create geographic distribution object.

domain(*elem*)

Create protein domain object.

domain_architecture(*elem*)

Create domain architecture object.

events(*elem*)

Create events object.

id(*elem*)

Create identifier object.

mol_seq(*elem*)

Create molecular sequence object.

point(*elem*)

Create point object, coordinates of a point.

polygon(*elem*)

Create polygon object, list of points.

property(*elem*)

Create properties from external resources.

reference(*elem*)

Create literature reference object.

sequence_relation(*elem*)

Create sequence relationship object, relationship between two sequences.

uri(*elem*)

Create uri object, expected to be a url.

class Bio.Phylo.PhyloXMLIO.**Writer**(*phyloxml*)

Bases: object

Methods for serializing a PhyloXML object to XML.

__init__(*phyloxml*)

Build an ElementTree from a PhyloXML object.

write(*file*, *encoding=DEFAULT_ENCODING*, *indent=True*)

Write PhyloXML to a file.

phyloxml(*obj*)

Convert phyloxml to Etree element.

other(*obj*)

Convert other to Etree element.

phylogeny(*obj*)

Serialize a phylogeny and its subnodes, in order.

clade(*obj*)

Serialize a clade and its subnodes, in order.

accession(*obj*)

Serialize a accession and its subnodes, in order.

annotation(*obj*)

Serialize a annotation and its subnodes, in order.

binary_characters(*obj*)

Serialize a binary_characters node and its subnodes.

clade_relation(*obj*)

Serialize a clade_relation and its subnodes, in order.

color(*obj*)

Serialize a color and its subnodes, in order.

confidence(*obj*)

Serialize a confidence and its subnodes, in order.

date(*obj*)

Serialize a date and its subnodes, in order.

distribution(*obj*)

Serialize a distribution and its subnodes, in order.

domain(*obj*)

Serialize a domain node.

domain_architecture(*obj*)

Serialize a domain_architecture and its subnodes, in order.

events(*obj*)

Serialize a events and its subnodes, in order.

id(*obj*)

Serialize a id and its subnodes, in order.

mol_seq(*obj*)
Serialize a mol_seq and its subnodes, in order.

node_id(*obj*)
Serialize a node_id and its subnodes, in order.

point(*obj*)
Serialize a point and its subnodes, in order.

polygon(*obj*)
Serialize a polygon and its subnodes, in order.

property(*obj*)
Serialize a property and its subnodes, in order.

reference(*obj*)
Serialize a reference and its subnodes, in order.

sequence(*obj*)
Serialize a sequence and its subnodes, in order.

sequence_relation(*obj*)
Serialize a sequence_relation and its subnodes, in order.

taxonomy(*obj*)
Serialize a taxonomy and its subnodes, in order.

uri(*obj*)
Serialize a uri and its subnodes, in order.

alt(*obj*)
Serialize a simple alt node.

branch_length(*obj*)
Serialize a simple branch_length node.

lat(*obj*)
Serialize a simple lat node.

long(*obj*)
Serialize a simple long node.

maximum(*obj*)
Serialize a simple maximum node.

minimum(*obj*)
Serialize a simple minimum node.

value(*obj*)
Serialize a simple value node.

width(*obj*)
Serialize a simple width node.

blue(*obj*)
Serialize a simple blue node.

duplications(*obj*)
Serialize a simple duplications node.

green(*obj*)
Serialize a simple green node.

losses(*obj*)
Serialize a simple losses node.

red(*obj*)
Serialize a simple red node.

speciations(*obj*)
Serialize a simple speciations node.

bc(*obj*)
Serialize a simple bc node.

code(*obj*)
Serialize a simple code node.

common_name(*obj*)
Serialize a simple common_name node.

desc(*obj*)
Serialize a simple desc node.

description(*obj*)
Serialize a simple description node.

location(*obj*)
Serialize a simple location node.

name(*obj*)
Serialize a simple name node.

rank(*obj*)
Serialize a simple rank node.

scientific_name(*obj*)
Serialize a simple scientific_name node.

symbol(*obj*)
Serialize a simple symbol node.

synonym(*obj*)
Serialize a simple synonym node.

type(*obj*)
Serialize a simple type node.

Bio.Phylo.TreeConstruction module

Classes and methods for tree construction.

class Bio.Phylo.TreeConstruction.**DistanceMatrix**(*names*, *matrix=None*)

Bases: _Matrix

Distance matrix class that can be used for distance based tree algorithms.

All diagonal elements will be zero no matter what the users provide.

__init__(*names*, *matrix=None*)

Initialize the class.

__setitem__(*item*, *value*)

Set Matrix's items to values.

format_phylip(*handle*)

Write data in Phylip format to a given file-like object or handle.

The output stream is the input distance matrix format used with Phylip programs (e.g. 'neighbor'). See: http://evolution.genetics.washington.edu/phylip/doc/neighbor.html

Parameters

handle
[file or file-like object] A writeable text mode file handle or other object supporting the 'write' method, such as StringIO or sys.stdout.

class Bio.Phylo.TreeConstruction.**DistanceCalculator**(*model='identity'*, *skip_letters=None*)

Bases: object

Calculates the distance matrix from a DNA or protein sequence alignment.

This class calculates the distance matrix from a multiple sequence alignment of DNA or protein sequences, and the given name of the substitution model.

Currently only scoring matrices are used.

Parameters

model
[str] Name of the model matrix to be used to calculate distance. The attribute `dna_models` contains the available model names for DNA sequences and `protein_models` for protein sequences.

Examples

Loading a small PHYLIP alignment from which to compute distances:

```
>>> from Bio.Phylo.TreeConstruction import DistanceCalculator
>>> from Bio import AlignIO
>>> aln = AlignIO.read(open('TreeConstruction/msa.phy'), 'phylip')
>>> print(aln)
Alignment with 5 rows and 13 columns
AACGTGGCCACAT Alpha
AAGGTCGCCACAC Beta
CAGTTCGCCACAA Gamma
```

(continues on next page)

(continued from previous page)

```
GAGATTTCCGCCT Delta
GAGATCTCCGCCC Epsilon
```

DNA calculator with 'identity' model:

```
>>> calculator = DistanceCalculator('identity')
>>> dm = calculator.get_distance(aln)
>>> print(dm)
  Alpha   0.000000
  Beta    0.230769    0.000000
  Gamma   0.384615    0.230769    0.000000
  Delta   0.538462    0.538462    0.538462    0.000000
  Epsilon 0.615385    0.384615    0.461538    0.153846    0.000000
      Alpha   Beta    Gamma   Delta   Epsilon
```

Protein calculator with 'blosum62' model:

```
>>> calculator = DistanceCalculator('blosum62')
>>> dm = calculator.get_distance(aln)
>>> print(dm)
Alpha   0.000000
Beta    0.369048    0.000000
Gamma   0.493976    0.250000    0.000000
Delta   0.585366    0.547619    0.566265    0.000000
Epsilon 0.700000    0.355556    0.488889    0.222222    0.000000
    Alpha   Beta    Gamma   Delta   Epsilon
```

Same calculation, using the new Alignment object:

```
>>> from Bio.Phylo.TreeConstruction import DistanceCalculator
>>> from Bio import Align
>>> aln = Align.read('TreeConstruction/msa.phy', 'phylip')
>>> print(aln)
Alpha             0 AACGTGGCCACAT 13
Beta              0 AAGGTCGCCACAC 13
Gamma             0 CAGTTCGCCACAA 13
Delta             0 GAGATTTCCGCCT 13
Epsilon           0 GAGATCTCCGCCC 13
```

DNA calculator with 'identity' model:

```
>>> calculator = DistanceCalculator('identity')
>>> dm = calculator.get_distance(aln)
>>> print(dm)
Alpha   0.000000
Beta    0.230769    0.000000
Gamma   0.384615    0.230769    0.000000
Delta   0.538462    0.538462    0.538462    0.000000
Epsilon 0.615385    0.384615    0.461538    0.153846    0.000000
    Alpha   Beta    Gamma   Delta   Epsilon
```

Protein calculator with 'blosum62' model:

```
>>> calculator = DistanceCalculator('blosum62')
>>> dm = calculator.get_distance(aln)
>>> print(dm)
Alpha   0.000000
Beta    0.369048    0.000000
Gamma   0.493976    0.250000    0.000000
Delta   0.585366    0.547619    0.566265    0.000000
Epsilon 0.700000    0.355556    0.488889    0.222222    0.000000
    Alpha   Beta    Gamma   Delta   Epsilon
```

dna_models = ['benner22', 'benner6', 'benner74', 'blastn', 'dayhoff', 'feng', 'genetic', 'gonnet1992', 'hoxd70', 'johnson', 'jones', 'levin', 'mclachlan', 'mdm78', 'megablast', 'blastn', 'rao', 'risler', 'schneider', 'str', 'trans']

protein_models = ['blastp', 'blosum45', 'blosum50', 'blosum62', 'blosum80', 'blosum90', 'pam250', 'pam30', 'pam70']

models = ['identity', 'benner22', 'benner6', 'benner74', 'blastn', 'dayhoff', 'feng', 'genetic', 'gonnet1992', 'hoxd70', 'johnson', 'jones', 'levin', 'mclachlan', 'mdm78', 'megablast', 'blastn', 'rao', 'risler', 'schneider', 'str', 'trans', 'blastp', 'blosum45', 'blosum50', 'blosum62', 'blosum80', 'blosum90', 'pam250', 'pam30', 'pam70']

__init__(*model='identity'*, *skip_letters=None*)

Initialize with a distance model.

get_distance(*msa*)

Return a DistanceMatrix for an Alignment or MultipleSeqAlignment object.

Parameters

msa
[Alignment or MultipleSeqAlignment object representing a] DNA or protein multiple sequence alignment.

class Bio.Phylo.TreeConstruction.**TreeConstructor**

Bases: object

Base class for all tree constructor.

build_tree(*msa*)

Caller to build the tree from an Alignment or MultipleSeqAlignment object.

This should be implemented in subclass.

class Bio.Phylo.TreeConstruction.**DistanceTreeConstructor**(*distance_calculator=None*, *method='nj'*)

Bases: *TreeConstructor*

Distance based tree constructor.

Parameters

method
[str] Distance tree construction method, 'nj'(default) or 'upgma'.

distance_calculator
[DistanceCalculator] The distance matrix calculator for multiple sequence alignment. It must be provided if `build_tree` will be called.

Examples

Loading a small PHYLIP alignment from which to compute distances, and then build a upgma Tree:

```
>>> from Bio.Phylo.TreeConstruction import DistanceTreeConstructor
>>> from Bio.Phylo.TreeConstruction import DistanceCalculator
>>> from Bio import AlignIO
>>> aln = AlignIO.read(open('TreeConstruction/msa.phy'), 'phylip')
>>> constructor = DistanceTreeConstructor()
>>> calculator = DistanceCalculator('identity')
>>> dm = calculator.get_distance(aln)
>>> upgmatree = constructor.upgma(dm)
>>> print(upgmatree)
Tree(rooted=True)
    Clade(branch_length=0, name='Inner4')
        Clade(branch_length=0.18749999999999994, name='Inner1')
            Clade(branch_length=0.07692307692307693, name='Epsilon')
            Clade(branch_length=0.07692307692307693, name='Delta')
        Clade(branch_length=0.11057692307692304, name='Inner3')
            Clade(branch_length=0.038461538461538464, name='Inner2')
                Clade(branch_length=0.11538461538461536, name='Gamma')
                Clade(branch_length=0.11538461538461536, name='Beta')
            Clade(branch_length=0.15384615384615383, name='Alpha')
```

Build a NJ Tree:

```
>>> njtree = constructor.nj(dm)
>>> print(njtree)
Tree(rooted=False)
    Clade(branch_length=0, name='Inner3')
        Clade(branch_length=0.18269230769230765, name='Alpha')
        Clade(branch_length=0.04807692307692307, name='Beta')
        Clade(branch_length=0.04807692307692307, name='Inner2')
            Clade(branch_length=0.27884615384615385, name='Inner1')
                Clade(branch_length=0.051282051282051266, name='Epsilon')
                Clade(branch_length=0.10256410256410259, name='Delta')
            Clade(branch_length=0.14423076923076922, name='Gamma')
```

Same example, using the new Alignment class:

```
>>> from Bio.Phylo.TreeConstruction import DistanceTreeConstructor
>>> from Bio.Phylo.TreeConstruction import DistanceCalculator
>>> from Bio import Align
>>> aln = Align.read(open('TreeConstruction/msa.phy'), 'phylip')
>>> constructor = DistanceTreeConstructor()
>>> calculator = DistanceCalculator('identity')
>>> dm = calculator.get_distance(aln)
>>> upgmatree = constructor.upgma(dm)
>>> print(upgmatree)
Tree(rooted=True)
    Clade(branch_length=0, name='Inner4')
        Clade(branch_length=0.18749999999999994, name='Inner1')
            Clade(branch_length=0.07692307692307693, name='Epsilon')
            Clade(branch_length=0.07692307692307693, name='Delta')
```

(continues on next page)

(continued from previous page)

```
        Clade(branch_length=0.11057692307692304, name='Inner3')
            Clade(branch_length=0.038461538461538464, name='Inner2')
                Clade(branch_length=0.11538461538461536, name='Gamma')
                Clade(branch_length=0.11538461538461536, name='Beta')
            Clade(branch_length=0.15384615384615383, name='Alpha')
```

Build a NJ Tree:

```
>>> njtree = constructor.nj(dm)
>>> print(njtree)
Tree(rooted=False)
    Clade(branch_length=0, name='Inner3')
        Clade(branch_length=0.18269230769230765, name='Alpha')
        Clade(branch_length=0.04807692307692307, name='Beta')
        Clade(branch_length=0.04807692307692307, name='Inner2')
            Clade(branch_length=0.27884615384615385, name='Inner1')
                Clade(branch_length=0.051282051282051266, name='Epsilon')
                Clade(branch_length=0.10256410256410259, name='Delta')
            Clade(branch_length=0.14423076923076922, name='Gamma')
```

methods = ['nj', 'upgma']

__init__(*distance_calculator=None*, *method='nj'*)

Initialize the class.

build_tree(*msa*)

Construct and return a Tree, Neighbor Joining or UPGMA.

upgma(*distance_matrix*)

Construct and return an UPGMA tree.

Constructs and returns an Unweighted Pair Group Method with Arithmetic mean (UPGMA) tree.

Parameters

distance_matrix
[DistanceMatrix] The distance matrix for tree construction.

nj(*distance_matrix*)

Construct and return a Neighbor Joining tree.

Parameters

distance_matrix
[DistanceMatrix] The distance matrix for tree construction.

__annotations__ = {}

class Bio.Phylo.TreeConstruction.**Scorer**

Bases: `object`

Base class for all tree scoring methods.

get_score(*tree*, *alignment*)

Caller to get the score of a tree for the given alignment.

This should be implemented in subclass.

class Bio.Phylo.TreeConstruction.**TreeSearcher**

Bases: object

Base class for all tree searching methods.

search(*starting_tree*, *alignment*)

Caller to search the best tree with a starting tree.

This should be implemented in subclass.

class Bio.Phylo.TreeConstruction.**NNITreeSearcher**(*scorer*)

Bases: *TreeSearcher*

Tree searching with Nearest Neighbor Interchanges (NNI) algorithm.

Parameters

scorer
[ParsimonyScorer] parsimony scorer to calculate the parsimony score of different trees during NNI algorithm.

__init__(*scorer*)

Initialize the class.

search(*starting_tree*, *alignment*)

Implement the TreeSearcher.search method.

Parameters

starting_tree
[Tree] starting tree of NNI method.

alignment
[Alignment or MultipleSeqAlignment object] multiple sequence alignment used to calculate parsimony score of different NNI trees.

__annotations__ = {}

class Bio.Phylo.TreeConstruction.**ParsimonyScorer**(*matrix=None*)

Bases: *Scorer*

Parsimony scorer with a scoring matrix.

This is a combination of Fitch algorithm and Sankoff algorithm. See ParsimonyTreeConstructor for usage.

Parameters

matrix
[_Matrix] scoring matrix used in parsimony score calculation.

__init__(*matrix=None*)

Initialize the class.

get_score(*tree*, *alignment*)

Calculate parsimony score using the Fitch algorithm.

Calculate and return the parsimony score given a tree and the MSA using either the Fitch algorithm (without a penalty matrix) or the Sankoff algorithm (with a matrix).

__annotations__ = {}

class Bio.Phylo.TreeConstruction.**ParsimonyTreeConstructor**(*searcher*, *starting_tree=None*)

Bases: *TreeConstructor*

Parsimony tree constructor.

Parameters

searcher
[TreeSearcher] tree searcher to search the best parsimony tree.

starting_tree
[Tree] starting tree provided to the searcher.

Examples

We will load an alignment, and then load various trees which have already been computed from it:

```
>>> from Bio import AlignIO, Phylo
>>> aln = AlignIO.read(open('TreeConstruction/msa.phy'), 'phylip')
>>> print(aln)
Alignment with 5 rows and 13 columns
AACGTGGCCACAT Alpha
AAGGTCGCCACAC Beta
CAGTTCGCCACAA Gamma
GAGATTTCCGCCT Delta
GAGATCTCCGCCC Epsilon
```

Load a starting tree:

```
>>> starting_tree = Phylo.read('TreeConstruction/nj.tre', 'newick')
>>> print(starting_tree)
Tree(rooted=False, weight=1.0)
    Clade(branch_length=0.0, name='Inner3')
        Clade(branch_length=0.01421, name='Inner2')
            Clade(branch_length=0.23927, name='Inner1')
                Clade(branch_length=0.08531, name='Epsilon')
                Clade(branch_length=0.13691, name='Delta')
            Clade(branch_length=0.2923, name='Alpha')
        Clade(branch_length=0.07477, name='Beta')
        Clade(branch_length=0.17523, name='Gamma')
```

Build the Parsimony tree from the starting tree:

```
>>> scorer = Phylo.TreeConstruction.ParsimonyScorer()
>>> searcher = Phylo.TreeConstruction.NNITreeSearcher(scorer)
>>> constructor = Phylo.TreeConstruction.ParsimonyTreeConstructor(searcher,␣
↪starting_tree)
>>> pars_tree = constructor.build_tree(aln)
>>> print(pars_tree)
Tree(rooted=True, weight=1.0)
    Clade(branch_length=0.0)
        Clade(branch_length=0.19732999999999998, name='Inner1')
            Clade(branch_length=0.13691, name='Delta')
            Clade(branch_length=0.08531, name='Epsilon')
        Clade(branch_length=0.04194000000000003, name='Inner2')
```

(continues on next page)

(continued from previous page)

```
            Clade(branch_length=0.01421, name='Inner3')
                Clade(branch_length=0.17523, name='Gamma')
                Clade(branch_length=0.07477, name='Beta')
            Clade(branch_length=0.2923, name='Alpha')
```

Same example, using the new Alignment class:

```
>>> from Bio import Align, Phylo
>>> alignment = Align.read(open('TreeConstruction/msa.phy'), 'phylip')
>>> print(alignment)
Alpha             0 AACGTGGCCACAT 13
Beta              0 AAGGTCGCCACAC 13
Gamma             0 CAGTTCGCCACAA 13
Delta             0 GAGATTTCCGCCT 13
Epsilon           0 GAGATCTCCGCCC 13
```

Load a starting tree:

```
>>> starting_tree = Phylo.read('TreeConstruction/nj.tre', 'newick')
>>> print(starting_tree)
Tree(rooted=False, weight=1.0)
    Clade(branch_length=0.0, name='Inner3')
        Clade(branch_length=0.01421, name='Inner2')
            Clade(branch_length=0.23927, name='Inner1')
                Clade(branch_length=0.08531, name='Epsilon')
                Clade(branch_length=0.13691, name='Delta')
            Clade(branch_length=0.2923, name='Alpha')
        Clade(branch_length=0.07477, name='Beta')
        Clade(branch_length=0.17523, name='Gamma')
```

Build the Parsimony tree from the starting tree:

```
>>> scorer = Phylo.TreeConstruction.ParsimonyScorer()
>>> searcher = Phylo.TreeConstruction.NNITreeSearcher(scorer)
>>> constructor = Phylo.TreeConstruction.ParsimonyTreeConstructor(searcher,␣
↪starting_tree)
>>> pars_tree = constructor.build_tree(alignment)
>>> print(pars_tree)
Tree(rooted=True, weight=1.0)
    Clade(branch_length=0.0)
        Clade(branch_length=0.19732999999999998, name='Inner1')
            Clade(branch_length=0.13691, name='Delta')
            Clade(branch_length=0.08531, name='Epsilon')
        Clade(branch_length=0.04194000000000003, name='Inner2')
            Clade(branch_length=0.01421, name='Inner3')
                Clade(branch_length=0.17523, name='Gamma')
                Clade(branch_length=0.07477, name='Beta')
            Clade(branch_length=0.2923, name='Alpha')
```

__annotations__ = {}

__init__(*searcher*, *starting_tree=None*)

Initialize the class.

build_tree(*alignment*)

Build the tree.

Parameters

alignment
[MultipleSeqAlignment] multiple sequence alignment to calculate parsimony tree.

Module contents

Package for working with phylogenetic trees.

See Also: http://biopython.org/wiki/Phylo

28.1.24 Bio.PopGen package

Subpackages

Bio.PopGen.GenePop package

Submodules

Bio.PopGen.GenePop.Controller module

Module to control GenePop.

class Bio.PopGen.GenePop.Controller.**GenePopController**(*genepop_dir=None*)

Bases: object

Define a class to interface with the GenePop program.

__init__(*genepop_dir=None*)

Initialize the controller.

genepop_dir is the directory where GenePop is.

The binary should be called Genepop (capital G)

test_pop_hz_deficiency(*fname*, *enum_test=True*, *dememorization=10000*, *batches=20*, *iterations=5000*)

Use Hardy-Weinberg test for heterozygote deficiency.

Returns a population iterator containing a dictionary where dictionary[locus]=(P-val, SE, Fis-WC, Fis-RH, steps).

Some loci have a None if the info is not available. SE might be none (for enumerations).

test_pop_hz_excess(*fname*, *enum_test=True*, *dememorization=10000*, *batches=20*, *iterations=5000*)

Use Hardy-Weinberg test for heterozygote deficiency.

Returns a population iterator containing a dictionary where dictionary[locus]=(P-val, SE, Fis-WC, Fis-RH, steps).

Some loci have a None if the info is not available. SE might be none (for enumerations).

test_pop_hz_prob(*fname*, *ext*, *enum_test=False*, *dememorization=10000*, *batches=20*, *iterations=5000*)

Use Hardy-Weinberg test based on probability.

Returns 2 iterators and a final tuple:

1. **Returns a loci iterator containing:**
   - A dictionary[pop_pos]=(P-val, SE, Fis-WC, Fis-RH, steps). Some pops have a None if the info is not available. SE might be none (for enumerations).
   - Result of Fisher's test (Chi2, deg freedom, prob).
2. **Returns a population iterator containing:**
   - A dictionary[locus]=(P-val, SE, Fis-WC, Fis-RH, steps). Some loci have a None if the info is not available. SE might be none (for enumerations).
   - Result of Fisher's test (Chi2, deg freedom, prob).
3. Final tuple (Chi2, deg freedom, prob).

test_global_hz_deficiency(*fname*, *enum_test=True*, *dememorization=10000*, *batches=20*, *iterations=5000*)

Use Global Hardy-Weinberg test for heterozygote deficiency.

Returns a triple with:

- An list per population containing (pop_name, P-val, SE, switches). Some pops have a None if the info is not available. SE might be none (for enumerations).
- An list per loci containing (locus_name, P-val, SE, switches). Some loci have a None if the info is not available. SE might be none (for enumerations).
- Overall results (P-val, SE, switches).

test_global_hz_excess(*fname*, *enum_test=True*, *dememorization=10000*, *batches=20*, *iterations=5000*)

Use Global Hardy-Weinberg test for heterozygote excess.

Returns a triple with:

- A list per population containing (pop_name, P-val, SE, switches). Some pops have a None if the info is not available. SE might be none (for enumerations).
- A list per loci containing (locus_name, P-val, SE, switches). Some loci have a None if the info is not available. SE might be none (for enumerations).
- Overall results (P-val, SE, switches)

test_ld(*fname*, *dememorization=10000*, *batches=20*, *iterations=5000*)

Test for linkage disequilibrium on each pair of loci in each population.

create_contingency_tables(*fname*)

Provision for creating Genotypic contingency tables.

test_genic_diff_all(*fname*, *dememorization=10000*, *batches=20*, *iterations=5000*)

Provision for Genic differentiation for all populations.

test_genic_diff_pair(*fname*, *dememorization=10000*, *batches=20*, *iterations=5000*)

Provision for Genic differentiation for all population pairs.

test_genotypic_diff_all(*fname*, *dememorization=10000*, *batches=20*, *iterations=5000*)

Provision for Genotypic differentiation for all populations.

`test_genotypic_diff_pair`(*fname*, *dememorization=10000*, *batches=20*, *iterations=5000*)

Provision for Genotypic differentiation for all population pairs.

`estimate_nm`(*fname*)

Estimate the Number of Migrants.

Parameters:

- fname - file name

Returns

- Mean sample size
- Mean frequency of private alleles
- Number of migrants for Ne=10
- Number of migrants for Ne=25
- Number of migrants for Ne=50
- Number of migrants after correcting for expected size

`calc_allele_genotype_freqs`(*fname*)

Calculate allele and genotype frequencies per locus and per sample.

Parameters:

- fname - file name

Returns tuple with 2 elements:

- Population iterator with
  - population name
  - Locus dictionary with key = locus name and content tuple as Genotype List with (Allele1, Allele2, observed, expected) (expected homozygotes, observed hm, expected heterozygotes, observed ht) Allele frequency/Fis dictionary with allele as key and (count, frequency, Fis Weir & Cockerham)
  - Totals as a pair
  - count
  - Fis Weir & Cockerham,
  - Fis Robertson & Hill
- Locus iterator with
  - Locus name
  - allele list
  - Population list with a triple
    - population name
    - list of allele frequencies in the same order as allele list above
    - number of genes

Will create a file called fname.INF

calc_diversities_fis_with_identity(*fname*)

Compute identity-base Gene diversities and Fis.

calc_diversities_fis_with_size(*fname*)

Provision to Computer Allele size-based Gene diversities and Fis.

calc_fst_all(*fname*)

Execute GenePop and gets Fst/Fis/Fit (all populations).

Parameters:

- fname - file name

Returns:

- (multiLocusFis, multiLocusFst, multiLocus Fit),
- Iterator of tuples (Locus name, Fis, Fst, Fit, Qintra, Qinter)

Will create a file called `fname.FST`.

This does not return the genotype frequencies.

calc_fst_pair(*fname*)

Estimate spatial structure from Allele identity for all population pairs.

calc_rho_all(*fname*)

Provision for estimating spatial structure from Allele size for all populations.

calc_rho_pair(*fname*)

Provision for estimating spatial structure from Allele size for all population pairs.

calc_ibd_diplo(*fname*, *stat='a'*, *scale='Log'*, *min_dist=0.00001*)

Calculate isolation by distance statistics for diploid data.

See _calc_ibd for parameter details.

Note that each pop can only have a single individual and the individual name has to be the sample coordinates.

calc_ibd_haplo(*fname*, *stat='a'*, *scale='Log'*, *min_dist=0.00001*)

Calculate isolation by distance statistics for haploid data.

See _calc_ibd for parameter details.

Note that each pop can only have a single individual and the individual name has to be the sample coordinates.

Bio.PopGen.GenePop.EasyController module

Control GenePop through an easier interface.

This interface is less efficient than the standard GenePopControler

class Bio.PopGen.GenePop.EasyController.**EasyController**(*fname*, *genepop_dir=None*)

Bases: `object`

Define a class for an easier interface with the GenePop program.

__init__(*fname*, *genepop_dir=None*)

Initialize the controller.

genepop_dir is the directory where GenePop is.

The binary should be called Genepop (capital G)

get_basic_info()

Obtain the population list and loci list from the file.

test_hw_pop(*pop_pos*, *test_type='probability'*)

Perform Hardy-Weinberg test on the given position.

test_hw_global(*test_type='deficiency'*, *enum_test=True*, *dememorization=10000*, *batches=20*, *iterations=5000*)

Perform Hardy-Weinberg global Heterozygote test.

test_ld_all_pair(*locus1*, *locus2*, *dememorization=10000*, *batches=20*, *iterations=5000*)

Test for linkage disequilibrium for each pair of loci in each population.

estimate_nm()

Estimate Nm. Just a simple bridge.

get_heterozygosity_info(*pop_pos*, *locus_name*)

Return the heterozygosity info for a certain locus on a population.

Returns (Expected homozygotes, observed homozygotes, Expected heterozygotes, observed heterozygotes)

get_genotype_count(*pop_pos*, *locus_name*)

Return the genotype counts for a certain population and locus.

get_fis(*pop_pos*, *locus_name*)

Return the Fis for a certain population and locus.

Below CW means Cockerham and Weir and RH means Robertson and Hill.

Returns a pair:

- dictionary [allele] = (repetition count, frequency, Fis CW) with information for each allele
- a triple with total number of alleles, Fis CW, Fis RH

get_alleles(*pop_pos*, *locus_name*)

Return the alleles for a certain population and locus.

get_alleles_all_pops(*locus_name*)

Return the alleles for a certain population and locus.

get_allele_frequency(*pop_pos*, *locus_name*)

Calculate the allele frequency for a certain locus on a population.

get_multilocus_f_stats()

Return the multilocus F stats.

Explain averaging. Returns Fis(CW), Fst, Fit

get_f_stats(*locus_name*)

Return F stats for a locus.

Returns Fis(CW), Fst, Fit, Qintra, Qinter

get_avg_fis()

Calculate identity-base average Fis.

get_avg_fst_pair()

Calculate Allele size-base average Fis for all population pairs.

get_avg_fst_pair_locus(*locus*)

Calculate Allele size-base average Fis for all population pairs of the given locus.

calc_ibd(*is_diplo=True*, *stat='a'*, *scale='Log'*, *min_dist=0.00001*)

Calculate isolation by distance statistics for Diploid or Haploid.

Bio.PopGen.GenePop.FileParser module

Code to parse BIG GenePop files.

The difference between this class and the standard Bio.PopGen.GenePop.Record class is that this one does not read the whole file to memory. It provides an iterator interface, slower but consuming much mess memory. Should be used with big files (Thousands of markers and individuals).

See http://wbiomed.curtin.edu.au/genepop/ , the format is documented here: http://wbiomed.curtin.edu.au/genepop/help_input.html .

Classes:

- FileRecord Holds GenePop data.

Functions:

Bio.PopGen.GenePop.FileParser.**read**(*fname*)

Parse a file containing a GenePop file.

fname is a file name that contains a GenePop record.

class Bio.PopGen.GenePop.FileParser.**FileRecord**(*fname*)

Bases: object

Hold information from a GenePop record.

Attributes: - marker_len The marker length (2 or 3 digit code per allele). - comment_line Comment line. - loci_list List of loci names.

Methods: - get_individual Returns the next individual of the current population. - skip_population Skips the current population.

skip_population skips the individuals of the current population, returns True if there are more populations.

get_individual returns an individual of the current population (or None if the list ended).

Each individual is a pair composed by individual name and a list of alleles (2 per marker or 1 for haploid data). Examples:

```
('Ind1', [(1,2),    (3,3), (200,201)]
('Ind2', [(2,None), (3,3), (None,None)]
('Other1', [(1,1),  (4,3), (200,200)]
```

__init__(*fname*)

Initialize the class.

__str__()
Return (reconstructs) a GenePop textual representation.

This might take a lot of memory. Marker length will be 3.

start_read()
Start parsing a file containing a GenePop file.

skip_header()
Skip the Header. To be done after a re-open.

seek_position(*pop*, *indiv*)
Seek a certain position in the file.

Arguments:

- pop - pop position (0 is first)
- indiv - individual in pop

skip_population()
Skip the current population. Returns true if there is another pop.

get_individual()
Get the next individual.

Returns individual information if there are more individuals in the current population. Returns True if there are no more individuals in the current population, but there are more populations. Next read will be of the following pop. Returns False if at end of file.

remove_population(*pos*, *fname*)
Remove a population (by position).

Arguments:

- pos - position
- fname - file to be created with population removed

remove_locus_by_position(*pos*, *fname*)
Remove a locus by position.

Arguments:

- pos - position
- fname - file to be created with locus removed

remove_loci_by_position(*positions*, *fname*)
Remove a set of loci by position.

Arguments:

- positions - positions
- fname - file to be created with locus removed

remove_locus_by_name(*name*, *fname*)
Remove a locus by name.

Arguments:

- name - name
- fname - file to be created with locus removed

remove_loci_by_name(*names*, *fname*)

Remove a loci list (by name).

Arguments:

- names - names
- fname - file to be created with loci removed

Bio.PopGen.GenePop.LargeFileParser module

Large file parsing of Genepop files.

The standard parser loads the whole file into memory. This parser provides an iterator over data.

Classes: - LargeRecord - Holds GenePop data.

Functions: - read - Parses a GenePop record (file) into a Record object.

Bio.PopGen.GenePop.LargeFileParser.**get_indiv**(*line*)

Get individual's data from line.

Bio.PopGen.GenePop.LargeFileParser.**read**(*handle*)

Parse a handle containing a GenePop file.

Arguments: - handle is a file-like object that contains a GenePop record.

class Bio.PopGen.GenePop.LargeFileParser.**Record**(*handle*)

Bases: object

Hold information from a GenePop record.

Members: marker_len The marker length (2 or 3 digit code per allele).

comment_line Comment line.

loci_list List of loci names.

data_generator Iterates over population data.

The generator will only work once. If you want to read a handle twice you have to re-open it!

data_generator can either be () - an empty tuple - marking a new population or an individual. An individual is something like ('Ind1', [(1,1), (3,None), (200,201)], In the case above the individual is called Ind1, has three diploid loci. For the second loci, one of the alleles is unknown.

__init__(*handle*)

Initialize the class.

data_generator()

Extract population data.

Module contents

Code to work with GenePop.

See http://wbiomed.curtin.edu.au/genepop/ , the format is documented here: http://wbiomed.curtin.edu.au/genepop/help_input.html .

Classes: Record Holds GenePop data.

Functions: read Parses a GenePop record (file) into a Record object.

Partially inspired by MedLine Code.

`Bio.PopGen.GenePop.`**`get_indiv`**`(`*`line`*`)`

Extract the details of the individual information on the line.

`Bio.PopGen.GenePop.`**`read`**`(`*`handle`*`)`

Parse a handle containing a GenePop file.

handle is a file-like object that contains a GenePop record.

`class` `Bio.PopGen.GenePop.`**`Record`**

Bases: `object`

Hold information from a GenePop record.

Members:

- marker_len The marker length (2 or 3 digit code per allele).
- comment_line Comment line.
- loci_list List of loci names.
- pop_list List of population names.
- populations List of population data.

In most genepop files, the population name is not trustable. It is strongly recommended that populations are referred by index.

populations has one element per population. Each element is itself a list of individuals, each individual is a pair composed by individual name and a list of alleles (2 per marker or 1 for haploids): Example:

```
[
    [
        ('Ind1', [(1,2),    (3,3), (200,201)],
        ('Ind2', [(2,None), (3,3), (None,None)],
    ],
    [
        ('Other1', [(1,1),  (4,3), (200,200)],
    ]
]
```

`__init__``()`

Initialize the class.

`__str__``()`

Return (reconstruct) a GenePop textual representation.

split_in_pops(*pop_names*)

Split a GP record in a dictionary with 1 pop per entry.

Given a record with n pops and m loci returns a dictionary of records (key pop_name) where each item is a record with a single pop and m loci.

Arguments: - pop_names - Population names

split_in_loci(*gp*)

Split a GP record in a dictionary with 1 locus per entry.

Given a record with n pops and m loci returns a dictionary of records (key locus name) where each item is a record with a single locus and n pops.

remove_population(*pos*)

Remove a population (by position).

remove_locus_by_position(*pos*)

Remove a locus by position.

remove_locus_by_name(*name*)

Remove a locus by name.

Module contents

PopGen: Population Genetics and Genomics library in Python.

28.1.25 Bio.Restriction package

Submodules

Bio.Restriction.PrintFormat module

Print the results of restriction enzyme analysis.

PrintFormat prints the results from restriction analysis in 3 different format: list, column or map.

The easiest way to use it is:

```
>>> from Bio.Restriction.PrintFormat import PrintFormat
>>> from Bio.Restriction.Restriction import RestrictionBatch
>>> from Bio.Seq import Seq
>>> pBs_mcs = Seq('GGTACCGGGCCCCCCCTCGAGGTCGACGGTATCGATAAGCTTGATATCGAATTC')
>>> restriction_batch = RestrictionBatch(['EcoRI', 'BamHI', 'ApaI'])
>>> result = restriction_batch.search(pBs_mcs)
>>> my_map = PrintFormat()
>>> my_map.print_that(result, 'My pBluescript mcs analysis:\n',
...                'No site:\n')
My pBluescript mcs analysis:
ApaI       :  12.
EcoRI      :  50.
No site:
BamHI

>>> my_map.sequence = pBs_mcs
```

(continues on next page)

(continued from previous page)

```
>>> my_map.print_as("map")
>>> my_map.print_that(result)
           12 ApaI
           |
           |                                          50 EcoRI
           |                                          |
GGTACCGGGCCCCCCCTCGAGGTCGACGGTATCGATAAGCTTGATATCGAATTC
||||||||||||||||||||||||||||||||||||||||||||||||||||||
CCATGGCCCGGGGGGGAGCTCCAGCTGCCATAGCTATTCGAACTATAGCTTAAG
1                                                    54


   Enzymes which do not cut the sequence.

BamHI

>>>
```

Some of the methods of PrintFormat are meant to be overridden by derived class.

Use the following parameters to control the appearance:

- **ConsoleWidth**
  [width of the console used default to 80.] should never be less than 60.
- NameWidth : space attributed to the name in PrintList method.
- Indent : Indent of the second line.
- **MaxSize**
  [Maximal size of the sequence (default=6:] -> 99 999 bp + 1 trailing ',' people are unlikely to ask for restriction map of sequences bigger than 100.000 bp. This is needed to determine the space to be reserved for sites location.
  - MaxSize = 5 => 9.999 bp
  - MaxSize = 6 => 99.999 bp
  - MaxSize = 7 => 999.999 bp

Example output:

```
<------------ ConsoleWidth --------------->
<- NameWidth ->
EcoRI         :   1, 45, 50, 300, 400, 650,
                      700, 1200, 2500.
                  <-->
                    Indent
```

class Bio.Restriction.PrintFormat.**PrintFormat**

Bases: object

PrintFormat allow the printing of results of restriction analysis.

ConsoleWidth = 80

NameWidth = 10

MaxSize = 6

Cmodulo = 0

PrefWidth = 80

Indent = 4

linesize = 70

print_as(*what='list'*)

Print the results as specified.

Valid format are:

'list' -> alphabetical order 'number' -> number of sites in the sequence 'map' -> a map representation of the sequence with the sites.

If you want more flexibility over-ride the virtual method make_format.

format_output(*dct*, *title=''*, *s1=''*)

Summarise results as a nicely formatted string.

Arguments:

- dct is a dictionary as returned by a RestrictionBatch.search()
- title is the title of the map. It must be a formatted string, i.e. you must include the line break.
- s1 is the title separating the list of enzymes that have sites from those without sites.
- s1 must be a formatted string as well.

The format of print_that is a list.

print_that(*dct*, *title=''*, *s1=''*)

Print the output of the format_output method (OBSOLETE).

Arguments:

- dct is a dictionary as returned by a RestrictionBatch.search()
- title is the title of the map. It must be a formatted string, i.e. you must include the line break.
- s1 is the title separating the list of enzymes that have sites from those without sites.
- s1 must be a formatted string as well.

This method prints the output of A.format_output() and it is here for backwards compatibility.

make_format(*cut=()*, *title=''*, *nc=()*, *s1=''*)

Virtual method used for formatting results.

Virtual method. Here to be pointed to one of the _make_* methods. You can as well create a new method and point make_format to it.

Bio.Restriction.Restriction_Dictionary module

Restriction Analysis Libraries.

Used REBASE emboss files version 404 (2024).

Module contents

Restriction Digest Enzymes.

Examples

```
>>> from Bio.Seq import Seq
>>> from Bio.Restriction import *
>>> pBs_mcs = 'GGTACCGGGCCCCCCCTCGAGGTCGACGGTATCGATAAGCTTGATATCGAATTCCTG'
>>> pBs_mcs += 'CAGCCCGGGGGATCCACTAGTTCTAGAGCGGCCGCCACCGCGGTGGAGCTC'
>>> seq = Seq(pBs_mcs)  # Multiple-cloning site of pBluescript SK(-)
>>> a = Analysis(AllEnzymes, seq)
>>> a.print_that()              # no argument -> print all the results
AbaSI      :  10, 12, 13, 16, 17, 18, 19, 20, 22, 23, 24, 25, 25, 26, 27...
BmeDI      :  7, 8, 8, 9, 9, 13, 14, 15, 16, 17, 18, 19, 19, 21, 21...
YkrI       :  10, 12, 13, 16, 16, 17, 19, 20, 21, 22, 23, 24, 25, 25, 26...
```

BmeDI : 1, 2, 7, 8, 8, 9, 9, 13, 14, 15, 16, 17, 18, 19… AccII : 98. AciI : 86, 90, 96, 98…

Enzymes which do not cut the sequence.

AspLEI BstHHI CfoI CviAII FaeI FaiI FatI GlaI HhaI Hin1II Hin6I HinP1I HpyCH4IV HpySE526I Hsp92II HspAI MaeII MseI NlaIII SaqAI TaiI Tru1I Tru9I… <BLANKLINE> >>> b = a.blunt() # Analysis with blunt enzmyes >>> a.print_that(b) # Print results for blunt cutters AccII : 98. AfaI : 4. AluBI : 40, 106. AluI : 40, 106. Bsh1236I : 98. BshFI : 10, 89. BsnI : 10, 89. BspANI : 10, 89…

Enzymes which do not cut the sequence.

FaiI GlaI CdiI MlyI SchI SspD5I AanI… <BLANKLINE>

28.1.26 Bio.SCOP package

Submodules

Bio.SCOP.Cla module

Handle the SCOP CLAssification file, which describes SCOP domains.

The file format is described in the scop "release notes.":http://scop.mrc-lmb.cam.ac.uk/scop/release-notes.html The latest CLA file can be found "elsewhere at SCOP.":http://scop.mrc-lmb.cam.ac.uk/scop/parse/

"Release 1.73": http://scop.mrc-lmb.cam.ac.uk/scop/parse/dir.cla.scop.txt_1.73 (July 2008)

class Bio.SCOP.Cla.**Record**(*line=None*)

Bases: object

Holds information for one SCOP domain.

Attributes:

- sid - SCOP identifier. e.g. d1danl2
- residues - The domain definition as a Residues object
- sccs - SCOP concise classification strings. e.g. b.1.2.1
- sunid - SCOP unique identifier for this domain
- hierarchy - A dictionary, keys are nodetype, values are sunid, describing the location of this domain in the SCOP hierarchy. See the Scop module for a description of nodetypes. This used to be a list of (key,value) tuples in older versions of Biopython (see Bug 3109).

__init__(*line=None*)

Initialize the class.

__str__()

Represent the SCOP classification record as a tab-separated string.

Bio.SCOP.Cla.**parse**(*handle*)

Iterate over a CLA file as Cla records for each line.

Arguments:

- handle - file-like object.

class Bio.SCOP.Cla.**Index**(*filename*)

Bases: dict

A CLA file indexed by SCOP identifiers for rapid random access.

__init__(*filename*)

Create CLA index.

Arguments:

- filename - The file to index

__getitem__(*key*)

Return an item from the indexed file.

Bio.SCOP.Des module

Handle the SCOP DEScription file.

The file format is described in the scop "release notes.":http://scop.berkeley.edu/release-notes-1.55.html The latest DES file can be found "elsewhere at SCOP.":http://scop.mrc-lmb.cam.ac.uk/scop/parse/

"Release 1.55":http://scop.berkeley.edu/parse/des.cla.scop.txt_1.55 (July 2001)

class Bio.SCOP.Des.**Record**(*line=None*)

Bases: object

Holds information for one node in the SCOP hierarchy.

Attributes:

- sunid - SCOP unique identifiers
- nodetype - One of 'cl' (class), 'cf' (fold), 'sf' (superfamily), 'fa' (family), 'dm' (protein), 'sp' (species), 'px' (domain). Additional node types may be added.
- sccs - SCOP concise classification strings. e.g. b.1.2.1

- name - The SCOP ID (sid) for domains (e.g. d1anu1), currently empty for other node types
- description - e.g. “All beta proteins”,”Fibronectin type III”,

__init__(*line=None*)

Initialize the class.

__str__()

Represent the SCOP description record as a tab-separated string.

Bio.SCOP.Des.**parse**(*handle*)

Iterate over a DES file as a Des record for each line.

Arguments:

- handle - file-like object

Bio.SCOP.Dom module

Handle the SCOP DOMain file.

The DOM file has been officially deprecated. For more information see the SCOP”release notes.”:http://scop.berkeley.edu/release-notes-1.55.html The DOM files for older releases can be found “elsewhere at SCOP.”:http://scop.mrc-lmb.cam.ac.uk/scop/parse/

class Bio.SCOP.Dom.**Record**(*line=None*)

Bases: `object`

Holds information for one SCOP domain.

Attributes:

- sid - The SCOP ID of the entry, e.g. d1anu1
- residues - The domain definition as a Residues object
- hierarchy - A string specifying where this domain is in the hierarchy.

__init__(*line=None*)

Initialize the class.

__str__()

Represent the SCOP domain record as a tab-separated string.

Bio.SCOP.Dom.**parse**(*handle*)

Iterate over a DOM file as a Dom record for each line.

Arguments:

- handle – file-like object.

Bio.SCOP.Hie module

Handle the SCOP HIErarchy files.

The SCOP Hierarchy files describe the SCOP hierarchy in terms of SCOP unique identifiers (sunid).

The file format is described in the SCOP release notes.

The latest HIE file can be found elsewhere at SCOP.

Release 1.55 (July 2001).

class Bio.SCOP.Hie.**Record**(*line=None*)

Bases: object

Holds information for one node in the SCOP hierarchy.

Attributes:

- sunid - SCOP unique identifiers of this node
- parent - Parents sunid
- children - Sequence of children sunids

__init__(*line=None*)

Initialize the class.

__str__()

Represent the SCOP hierarchy record as a string.

Bio.SCOP.Hie.**parse**(*handle*)

Iterate over a HIE file as Hie records for each line.

Arguments:

- handle - file-like object.

Bio.SCOP.Raf module

ASTRAL RAF (Rapid Access Format) Sequence Maps.

The ASTRAL RAF Sequence Maps record the relationship between the PDB SEQRES records (representing the sequence of the molecule used in an experiment) to the ATOM records (representing the atoms experimentally observed).

This data is derived from the Protein Data Bank CIF files. Known errors in the CIF files are corrected manually, with the original PDB file serving as the final arbiter in case of discrepancies.

Residues are referenced by residue ID. This consists of a the PDB residue sequence number (up to 4 digits) and an optional PDB insertion code (an ascii alphabetic character, a-z, A-Z). e.g. “1”, “10A”, “1010b”, “-1”

See “ASTRAL RAF Sequence Maps”:http://astral.stanford.edu/raf.html

Dictionary *protein_letters_3to1_extended* provides a mapping from the 3-letter amino acid codes found in PDB files to 1-letter codes. The 3-letter codes include chemically modified residues.

Bio.SCOP.Raf.**normalize_letters**(*one_letter_code*)

Convert RAF one-letter amino acid codes into IUPAC standard codes.

Letters are uppercased, and “.” (“Unknown”) is converted to “X”.

class Bio.SCOP.Raf.**SeqMapIndex**(*filename*)

Bases: dict

An RAF file index.

The RAF file itself is about 50 MB. This index provides rapid, random access of RAF records without having to load the entire file into memory.

The index key is a concatenation of the PDB ID and chain ID. e.g "2drcA", "155c_". RAF uses an underscore to indicate blank chain IDs.

__init__(*filename*)

Initialize the RAF file index.

Arguments:

- filename – The file to index

__getitem__(*key*)

Return an item from the indexed file.

getSeqMap(*residues*)

Get the sequence map for a collection of residues.

Arguments:

- residues – A Residues instance, or a string that can be converted into a Residues instance.

class Bio.SCOP.Raf.**SeqMap**(*line=None*)

Bases: object

An ASTRAL RAF (Rapid Access Format) Sequence Map.

This is a list like object; You can find the location of particular residues with index(), slice this SeqMap into fragments, and glue fragments back together with extend().

Attributes:

- pdbid – The PDB 4 character ID
- pdb_datestamp – From the PDB file
- version – The RAF format version. e.g. 0.01
- flags – RAF flags. (See release notes for more information.)
- res – A list of Res objects, one for each residue in this sequence map

__init__(*line=None*)

Initialize the class.

index(*resid*, *chainid='_'*)

Return the index of the SeqMap for the given resid and chainid.

__getitem__(*index*)

Extract a single Res object from the SeqMap.

append(*res*)

Append another Res object onto the list of residue mappings.

extend(*other*)

Append another SeqMap onto the end of self.

Both SeqMaps must have the same PDB ID, PDB datestamp and RAF version. The RAF flags are erased if they are inconsistent. This may happen when fragments are taken from different chains.

__iadd__(*other*)

In place addition of SeqMap objects.

__add__(*other*)

Addition of SeqMap objects.

getAtoms(*pdb_handle*, *out_handle*)

Extract all relevant ATOM and HETATOM records from a PDB file.

The PDB file is scanned for ATOM and HETATOM records. If the chain ID, residue ID (seqNum and iCode), and residue type match a residue in this sequence map, then the record is echoed to the output handle.

This is typically used to find the coordinates of a domain, or other residue subset.

Arguments:

- pdb_handle – A handle to the relevant PDB file.
- out_handle – All output is written to this file like object.

class Bio.SCOP.Raf.**Res**

Bases: object

A single residue mapping from a RAF record.

Attributes:

- chainid – A single character chain ID.
- resid – The residue ID.
- atom – amino acid one-letter code from ATOM records.
- seqres – amino acid one-letter code from SEQRES records.

__init__()

Initialize the class.

Bio.SCOP.Raf.**parse**(*handle*)

Iterate over RAF file, giving a SeqMap object for each line.

Arguments:

- handle – file-like object.

Bio.SCOP.Residues module

A collection of residues from a PDB structure.

class Bio.SCOP.Residues.**Residues**(*str=None*)

Bases: object

A collection of residues from a PDB structure.

This class provides code to work with SCOP domain definitions. These are concisely expressed as one or more chain fragments. For example, "(1bba A:10-20,B:)" indicates residue 10 through 20 (inclusive) of chain A, and

every residue of chain B in the pdb structure 1bba. The pdb id and brackets are optional. In addition "-" indicates every residue of a pbd structure with one unnamed chain.

Start and end residue ids consist of the residue sequence number and an optional single letter insertion code. e.g. "12", "-1", "1a", "1000"

pdbid – An optional PDB id, e.g. "1bba"

fragments – A sequence of tuples (chainID, startResID, endResID)

__init__(*str=None*)

Initialize the class.

__str__()

Represent the SCOP residues record as a string.

Module contents

SCOP: Structural Classification of Proteins.

The SCOP database aims to provide a manually constructed classification of all know protein structures into a hierarchy, the main levels of which are family, superfamily and fold.

- "SCOP":http://scop.mrc-lmb.cam.ac.uk/legacy/
- "Introduction":http://scop.mrc-lmb.cam.ac.uk/legacy/intro.html
- "SCOP parsable files":http://scop.mrc-lmb.cam.ac.uk/legacy/parse/

The Scop object in this module represents the entire SCOP classification. It can be built from the three SCOP parsable files, modified is so desired, and converted back to the same file formats. A single SCOP domain (represented by the Domain class) can be obtained from Scop using the domain's SCOP identifier (sid).

- **nodeCodeDict – A mapping between known 2 letter node codes and a longer**
  description. The known node types are 'cl' (class), 'cf' (fold), 'sf' (superfamily), 'fa' (family), 'dm' (domain), 'sp' (species), 'px' (domain). Additional node types may be added in the future.

This module also provides code to access SCOP over the WWW.

Functions:

- search – Access the main CGI script (DEPRECATED, no longer available)..
- _open – Internally used function.

Bio.SCOP.**cmp_sccs**(*sccs1*, *sccs2*)

Order SCOP concise classification strings (sccs).

a.4.5.1 < a.4.5.11 < b.1.1.1

A sccs (e.g. a.4.5.11) compactly represents a domain's classification. The letter represents the class, and the numbers are the fold, superfamily, and family, respectively.

Bio.SCOP.**parse_domain**(*term*)

Convert an ASTRAL header string into a Scop domain.

An ASTRAL (http://astral.stanford.edu/) header contains a concise description of a SCOP domain. A very similar format is used when a Domain object is converted into a string. The Domain returned by this method contains most of the SCOP information, but it will not be located within the SCOP hierarchy (i.e. The parent node will be None). The description is composed of the SCOP protein and species descriptions.

A typical ASTRAL header looks like – >d1tpt_1 a.46.2.1 (1-70) Thymidine phosphorylase {Escherichia coli}

class Bio.SCOP.**Scop**(*cla_handle=None*, *des_handle=None*, *hie_handle=None*, *dir_path=None*, *db_handle=None*, *version=None*)

Bases: object

The entire SCOP hierarchy.

root – The root node of the hierarchy

__init__(*cla_handle=None*, *des_handle=None*, *hie_handle=None*, *dir_path=None*, *db_handle=None*, *version=None*)

Build the SCOP hierarchy from the SCOP parsable files, or a sql backend.

If no file handles are given, then a Scop object with a single empty root node is returned.

If a directory and version are given (with dir_path=.., version=...) or file handles for each file, the whole scop tree will be built in memory.

If a MySQLdb database handle is given, the tree will be built as needed, minimising construction times. To build the SQL database to the methods write_xxx_sql to create the tables.

getRoot()

Get root node.

getDomainBySid(*sid*)

Return a domain from its sid.

getNodeBySunid(*sunid*)

Return a node from its sunid.

getDomains()

Return an ordered tuple of all SCOP Domains.

write_hie(*handle*)

Build an HIE SCOP parsable file from this object.

write_des(*handle*)

Build a DES SCOP parsable file from this object.

write_cla(*handle*)

Build a CLA SCOP parsable file from this object.

getDomainFromSQL(*sunid=None*, *sid=None*)

Load a node from the SQL backend using sunid or sid.

getAscendentFromSQL(*node*, *type*)

Get ascendents using SQL backend.

getDescendentsFromSQL(*node*, *type*)

Get descendents of a node using the database backend.

This avoids repeated iteration of SQL calls and is therefore much quicker than repeatedly calling node.getChildren().

write_hie_sql(*handle*)

Write HIE data to SQL database.

write_cla_sql(*handle*)

Write CLA data to SQL database.

write_des_sql(*handle*)

Write DES data to SQL database.

class Bio.SCOP.**Node**(*scop=None*)

Bases: object

A node in the Scop hierarchy.

Attributes:

- sunid – SCOP unique identifiers. e.g. '14986'
- parent – The parent node
- children – A list of child nodes
- sccs – SCOP concise classification string. e.g. 'a.1.1.2'
- type – A 2 letter node type code. e.g. 'px' for domains
- description – Description text.

__init__(*scop=None*)

Initialize a Node in the scop hierarchy.

If a Scop instance is provided to the constructor, this will be used to lookup related references using the SQL methods. If no instance is provided, it is assumed the whole tree exists and is connected.

__str__()

Represent the node as a string.

toHieRecord()

Return an Hie.Record.

toDesRecord()

Return a Des.Record.

getChildren()

Return a list of children of this Node.

getParent()

Return the parent of this Node.

getDescendents(*node_type*)

Return a list of all descendant nodes of the given type.

Node type can be a two letter code or longer description, e.g. 'fa' or 'family'.

getAscendent(*node_type*)

Return the ancenstor node of the given type, or None.

Node type can be a two letter code or longer description, e.g. 'fa' or 'family'.

class Bio.SCOP.**Domain**(*scop=None*)

Bases: *Node*

A SCOP domain. A leaf node in the Scop hierarchy.

Attributes:

- sid - The SCOP domain identifier. e.g. "d5hbib_"
- residues - A Residue object. It defines the collection of PDB atoms that make up this domain.

__init__(*scop=None*)

Initialize a SCOP Domain object.

__str__()

Represent the SCOP Domain as a string.

toDesRecord()

Return a Des.Record.

toClaRecord()

Return a Cla.Record.

__annotations__ = {}

class Bio.SCOP.Astral(*dir_path=None*, *version=None*, *scop=None*, *astral_file=None*, *db_handle=None*)

Bases: object

Representation of the ASTRAL database.

Abstraction of the ASTRAL database, which has sequences for all the SCOP domains, as well as clusterings by percent id or evalue.

__init__(*dir_path=None*, *version=None*, *scop=None*, *astral_file=None*, *db_handle=None*)

Initialize the astral database.

You must provide either a directory of SCOP files:

- **dir_path - string, the path to location of the scopseq-x.xx directory**
  (not the directory itself), and
- version -a version number.

or, a FASTA file:

- astral_file - string, a path to a fasta file (which will be loaded in memory)

or, a MYSQL database:

- db_handle - a database handle for a MYSQL database containing a table 'astral' with the astral data in it. This can be created using writeToSQL.

domainsClusteredByEv(*id*)

Get domains clustered by evalue.

domainsClusteredById(*id*)

Get domains clustered by percentage identity.

getAstralDomainsFromFile(*filename=None*, *file_handle=None*)

Get the scop domains from a file containing a list of sids.

getAstralDomainsFromSQL(*column*)

Load ASTRAL domains from the MySQL database.

Load a set of astral domains from a column in the astral table of a MYSQL database (which can be created with writeToSQL(...).

getSeqBySid(*domain*)

Get the seq record of a given domain from its sid.

getSeq(*domain*)

Return seq associated with domain.

hashedDomainsById(*id*)

Get domains clustered by sequence identity in a dict.

hashedDomainsByEv(*id*)

Get domains clustered by evalue in a dict.

isDomainInId(*dom*, *id*)

Return true if the domain is in the astral clusters for percent ID.

isDomainInEv(*dom*, *id*)

Return true if the domain is in the ASTRAL clusters for evalues.

writeToSQL(*db_handle*)

Write the ASTRAL database to a MYSQL database.

Bio.SCOP.**search**(*pdb=None*, *key=None*, *sid=None*, *disp=None*, *dir=None*, *loc=None*, *cgi='http://scop.mrc-lmb.cam.ac.uk/legacy/search.cgi'*, ***keywds*)

Access SCOP search and return a handle to the results (DEPRECATED).

Access search.cgi and return a handle to the results. See the online help file for an explanation of the parameters: http://scop.mrc-lmb.cam.ac.uk/legacy/help.html

Raises an IOError if there's a network error.

This function is now DEPRECATED and will be removed in a future release of Biopython because this search.cgi API is no longer available with SCOP now hosted the the EBI.

28.1.27 Bio.SVDSuperimposer package

Module contents

Align on protein structure onto another using SVD alignment.

SVDSuperimposer finds the best rotation and translation to put two point sets on top of each other (minimizing the RMSD). This is eg. useful to superimpose crystal structures. SVD stands for singular value decomposition, which is used in the algorithm.

class Bio.SVDSuperimposer.**SVDSuperimposer**

Bases: `object`

Class to run SVD alignment.

SVDSuperimposer finds the best rotation and translation to put two point sets on top of each other (minimizing the RMSD). This is eg. useful to superimpose crystal structures.

SVD stands for Singular Value Decomposition, which is used to calculate the superposition.

Reference:

Matrix computations, 2nd ed. Golub, G. & Van Loan, CF., The Johns Hopkins University Press, Baltimore, 1989

start with two coordinate sets (Nx3 arrays - float)

```
>>> from Bio.SVDSuperimposer import SVDSuperimposer
>>> from numpy import array, dot, set_printoptions
>>>
>>> x = array([[51.65, -1.90, 50.07],
...       [50.40, -1.23, 50.65],
...       [50.68, -0.04, 51.54],
```

(continues on next page)

(continued from previous page)

```
...      [50.22, -0.02, 52.85]], 'f')
>>>
>>> y = array([[51.30, -2.99, 46.54],
...      [51.09, -1.88, 47.58],
...      [52.36, -1.20, 48.03],
...      [52.71, -1.18, 49.38]], 'f')
```

start

```
>>> sup = SVDSuperimposer()
```

set the coords y will be rotated and translated on x

```
>>> sup.set(x, y)
```

do the lsq fit

```
>>> sup.run()
```

get the rmsd

```
>>> rms = sup.get_rms()
```

get rotation (right multiplying!) and the translation

```
>>> rot, tran = sup.get_rotran()
```

rotate y on x

```
>>> y_on_x1 = dot(y, rot) + tran
```

same thing

```
>>> y_on_x2 = sup.get_transformed()
```

```
>>> set_printoptions(precision=2)
>>> print(y_on_x1)
[[ 5.17e+01 -1.90e+00  5.01e+01]
 [ 5.04e+01 -1.23e+00  5.06e+01]
 [ 5.07e+01 -4.16e-02  5.15e+01]
 [ 5.02e+01 -1.94e-02  5.29e+01]]
>>> print(y_on_x2)
[[ 5.17e+01 -1.90e+00  5.01e+01]
 [ 5.04e+01 -1.23e+00  5.06e+01]
 [ 5.07e+01 -4.16e-02  5.15e+01]
 [ 5.02e+01 -1.94e-02  5.29e+01]]
>>> print("%.2f" % rms)
0.00
```

__init__()

Initialize the class.

set(*reference_coords*, *coords*)

Set the coordinates to be superimposed.

coords will be put on top of reference_coords.

- reference_coords: an NxDIM array
- coords: an NxDIM array

DIM is the dimension of the points, N is the number of points to be superimposed.

run()

Superimpose the coordinate sets.

get_transformed()

Get the transformed coordinate set.

get_rotran()

Right multiplying rotation matrix and translation.

get_init_rms()

Root mean square deviation of untransformed coordinates.

get_rms()

Root mean square deviation of superimposed coordinates.

28.1.28 Bio.SearchIO package

Subpackages

Bio.SearchIO.BlastIO package

Submodules

Bio.SearchIO.BlastIO.blast_tab module

Bio.SearchIO parser for BLAST+ tab output format, with or without comments.

class Bio.SearchIO.BlastIO.blast_tab.**BlastTabIndexer**(*filename*, *comments=False*, *fields=_DEFAULT_FIELDS*)

Bases: SearchIndexer

Indexer class for BLAST+ tab output.

__init__(*filename*, *comments=False*, *fields=_DEFAULT_FIELDS*)

Initialize the class.

__iter__()

Iterate over the file handle; yields key, start offset, and length.

get_raw(*offset*)

Return the raw bytes string of a QueryResult object from the given offset.

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.SearchIO.BlastIO.blast_tab.**BlastTabParser**(*handle*, *comments=False*, *fields=_DEFAULT_FIELDS*)

Bases: object

Parser for the BLAST tabular format.

__init__(*handle*, *comments=False*, *fields=_DEFAULT_FIELDS*)

Initialize the class.

__iter__()

Iterate over BlastTabParser, yields query results.

class Bio.SearchIO.BlastIO.blast_tab.**BlastTabWriter**(*handle*, *comments=False*, *fields=_DEFAULT_FIELDS*)

Bases: object

Writer for blast-tab output format.

__init__(*handle*, *comments=False*, *fields=_DEFAULT_FIELDS*)

Initialize the class.

write_file(*qresults*)

Write to the handle, return how many QueryResult objects were written.

Bio.SearchIO.BlastIO.blast_xml module

Bio.SearchIO parser for BLAST+ XML output formats.

class Bio.SearchIO.BlastIO.blast_xml.**BlastXmlParser**(*handle*, *use_raw_query_ids=False*, *use_raw_hit_ids=False*)

Bases: object

Parser for the BLAST XML format.

__init__(*handle*, *use_raw_query_ids=False*, *use_raw_hit_ids=False*)

Initialize the class.

__iter__()

Iterate over BlastXmlParser object yields query results.

class Bio.SearchIO.BlastIO.blast_xml.**BlastXmlIndexer**(*filename*, ***kwargs*)

Bases: SearchIndexer

Indexer class for BLAST XML output.

qstart_mark = b'<Iteration>'

qend_mark = b'</Iteration>'

block_size = 16384

__init__(*filename*, ***kwargs*)

Initialize the class.

__iter__()

Iterate over BlastXmlIndexer yields qstart_id, start_offset, block's length.

get_raw(*offset*)

Return the raw record from the file as a bytes string.

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.SearchIO.BlastIO.blast_xml.**BlastXmlWriter**(*handle*, *use_raw_query_ids=True*, *use_raw_hit_ids=True*)

Bases: object

Stream-based BLAST+ XML Writer.

__init__(*handle*, *use_raw_query_ids=True*, *use_raw_hit_ids=True*)

Initialize the class.

write_file(*qresults*)

Write the XML contents to the output handle.

Module contents

Bio.SearchIO support for BLAST+ output formats.

This module adds support for parsing BLAST+ outputs. BLAST+ is a rewrite of NCBI's legacy BLAST (Basic Local Alignment Search Tool), based on the NCBI C++ toolkit. The BLAST+ suite is available as command line programs or on NCBI's web page.

Bio.SearchIO.BlastIO was tested on the following BLAST+ flavors and versions:

- flavors: blastn, blastp, blastx, tblastn, tblastx
- versions: 2.2.22+, 2.2.26+

You should also be able to parse outputs from a local BLAST+ search or from NCBI's web interface. Although the module was not tested against all BLAST+, it should still be able to parse these other versions' outputs. Please submit a bug report if you stumble upon an unparsable file.

Some output formats from the BLAST legacy suite (BLAST+'s predecessor) may still be parsed by this module. However, results are not guaranteed. You may try to use the Bio.Blast module to parse them instead.

More information about BLAST are available through these links:

- Publication: http://www.biomedcentral.com/1471-2105/10/421
- Web interface: http://blast.ncbi.nlm.nih.gov/
- User guide: http://www.ncbi.nlm.nih.gov/books/NBK1762/

Supported Formats

Bio.SearchIO.BlastIO supports the following BLAST+ output formats:

- XML - 'blast-xml' - parsing, indexing, writing
- Tabular - 'blast-tab' - parsing, indexing, writing

blast-xml

The blast-xml parser follows the BLAST XML DTD written here: http://www.ncbi.nlm.nih.gov/dtd/NCBI_BlastOutput.mod.dtd

It provides the following attributes for each SearchIO object:

Object	Attribute	XML Element
QueryResult	target	BlastOutput_db
	program	BlastOutput_program
	reference	BlastOutput_reference
	version	BlastOutput_version[1]
	description	Iteration_query-def
	id	Iteration_query-ID
	seq_len	Iteration_query-len
	param_evalue_threshold	Parameters_expect
	param_entrez_query	Parameters_entrez-query
	param_filter	Parameters_filter
	param_gap_extend	Parameters_gap-extend
	param_gap_open	Parameters_gap-open
	param_include	Parameters_include
	param_matrix	Parameters_matrix
	param_pattern	Parameters_pattern
	param_score_match	Parameters_sc-match
	param_score_mismatch	Parameters_sc-mismatch
	stat_db_num	Statistics_db-num
	stat_db_len	Statistics_db-len
	stat_eff_space	Statistics_eff-space
	stat_entropy	Statistics_entropy
	stat_hsp_len	Statistics_hsp-len
	stat_kappa	Statistics_kappa
	stat_lambda	Statistics_lambda
Hit	accession	Hit_accession
	description	Hit_def
	id	Hit_id
	seq_len	Hit_len
HSP	bitscore	Hsp_bit-score
	density	Hsp_density
	evalue	Hsp_evalue
	gap_num	Hsp_gaps
	ident_num	Hsp_identity
	pos_num	Hsp_positive
	bitscore_raw	Hsp_score
HSPFragment (also via HSP)	aln_span	Hsp_align-len
	hit_frame	Hsp_hit-frame
	hit_start	Hsp_hit-from
	hit_end	Hsp_hit-to
	hit	Hsp_hseq
	aln_annotation	Hsp_midline
	pattern_start	Hsp_pattern-from
	pattern_end	Hsp_pattern-to
	query_frame	Hsp_query-frame

continues on next page

Table 2 – continued from previous page

Object	Attribute	XML Element
	query_start	Hsp_query-from
	query_end	Hsp_query-to
	query	Hsp_qseq

You may notice that in BLAST XML files, sometimes BLAST replaces your true sequence ID with its own generated ID. For example, the query IDs become 'Query_1', 'Query_2', and so on. While the hit IDs sometimes become 'gnl|BL_ORD_ID|1', 'gnl|BL_ORD_ID|2', and so on. In these cases, BLAST lumps the true sequence IDs together with their descriptions.

The blast-xml parser is aware of these modifications and will attempt to extract the true sequence IDs out of the descriptions. So when accessing QueryResult or Hit objects, you will use the non-BLAST-generated IDs.

This behavior on the query IDs can be disabled using the 'use_raw_query_ids' parameter while the behavior on the hit IDs can be disabled using the 'use_raw_hit_ids' parameter. Both are boolean values that can be supplied to SearchIO.read or SearchIO.parse, with the default values set to 'False'.

In any case, the raw BLAST IDs can always be accessed using the query or hit object's 'blast_id' attribute.

The blast-xml write function also accepts 'use_raw_query_ids' and 'use_raw_hit_ids' parameters. However, note that the default values for the writer are set to 'True'. This is because the writer is meant to mimic native BLAST result as much as possible.

blast-tab

The default format for blast-tab support is the variant without comments (-m 6 flag). Commented BLAST tabular files may be parsed, indexed, or written using the keyword argument 'comments' set to True:

```
>>> # blast-tab defaults to parsing uncommented files
>>> from Bio import SearchIO
>>> uncommented = 'Blast/tab_2226_tblastn_004.txt'
>>> qresult = SearchIO.read(uncommented, 'blast-tab')
>>> qresult
QueryResult(id='gi|11464971:4-101', 5 hits)
```

```
>>> # set the keyword argument to parse commented files
>>> commented = 'Blast/tab_2226_tblastn_008.txt'
>>> qresult = SearchIO.read(commented, 'blast-tab', comments=True)
>>> qresult
QueryResult(id='gi|11464971:4-101', 5 hits)
```

For uncommented files, the parser defaults to using BLAST's default column ordering: 'qseqid sseqid pident length mismatch gapopen qstart qend sstart send evalue bitscore'.

If you want to parse an uncommented file with a customized column order, you can use the 'fields' keyword argument to pass the custom column order. The names of the column follow BLAST's naming. For example, 'qseqid' is the column for the query sequence ID. These names may be passed either as a Python list or as a space-separated strings.

```
>>> # pass the custom column names as a Python list
>>> fname = 'Blast/tab_2226_tblastn_009.txt'
>>> custom_fields = ['qseqid', 'sseqid']
```

(continues on next page)

[1] may be modified

(continued from previous page)

```
>>> qresult = next(SearchIO.parse(fname, 'blast-tab', fields=custom_fields))
>>> qresult
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
```

```
>>> # pass the custom column names as a space-separated string
>>> fname = 'Blast/tab_2226_tblastn_009.txt'
>>> custom_fields = 'qseqid sseqid'
>>> qresult = next(SearchIO.parse(fname, 'blast-tab', fields=custom_fields))
>>> qresult
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
```

You may also use the 'std' field name as an alias to BLAST's default 12 columns, just like when you run a command line BLAST search.

Note that the 'fields' keyword argument will be ignored if the parsed file is commented. Commented files have their column ordering stated explicitly in the file, so there is no need to specify it again in SearchIO.

'comments' and 'fields' keyword arguments are both applicable for parsing, indexing, and writing.

blast-tab provides the following attributes for each SearchIO objects:

Object	Attribute	Column name
QueryResult	accession	qacc
	accession_version	qaccver
	gi	qgi
	seq_len	qlen
	id	qseqid
Hit	accession	sacc
	accession_version	sacc_ver
	gi	sgi
	gi_all	sallgi
	id_all	sallseqid
	seq_len	slen
	id	sseqid
HSP	bitscore	bitscore
	btop	btop
	evalue	evalue
	gapopen_num	gapopen
	gap_num	gaps
	ident_num	nident
	ident_pct	pident
	mismatch_num	mismatch
	pos_pct	ppos
	pos_num	positive
	bitscore_raw	score
HSPFragment (also via HSP)	frames	frames[2]
	aln_span	length
	query_end	qend
	query_frame	qframe
	query	qseq
	query_start	qstart
	hit_end	send

continues on next page

Table 3 – continued from previous page

Object	Attribute	Column name
	hit_frame	sframe
	hit	sseq
	hit_start	sstart

If the parsed file is commented, the following attributes may be available as well:

Object	Attribute	Value
QueryResult	description	query description
	fields	columns in the output file
	program	BLAST flavor
	rid	remote search ID
	target	target database
	version	BLAST version

Bio.SearchIO.ExonerateIO package

Submodules

Bio.SearchIO.ExonerateIO.exonerate_cigar module

Bio.SearchIO parser for Exonerate cigar output format.

class Bio.SearchIO.ExonerateIO.exonerate_cigar.**ExonerateCigarParser**(*handle*)

Bases: _BaseExonerateParser

Parser for Exonerate cigar strings.

parse_alignment_block(*header*)

Parse alignment block for cigar format, return query results, hits, hsps.

__abstractmethods__ = frozenset({})

__annotations__ = {'_ALN_MARK': 'Optional[str]'}

class Bio.SearchIO.ExonerateIO.exonerate_cigar.**ExonerateCigarIndexer**(*filename*, ***kwargs*)

Bases: *ExonerateVulgarIndexer*

Indexer class for exonerate cigar lines.

get_qresult_id(*pos*)

Return the query ID of the nearest cigar line.

__abstractmethods__ = frozenset({})

__annotations__ = {'_parser': 'Type[_BaseExonerateParser]', '_query_mark': 'Optional[bytes]'}

[2] When 'frames' is present, both `query_frame` and `hit_frame` will be present as well. It is recommended that you use these instead of 'frames' directly.

Bio.SearchIO.ExonerateIO.exonerate_text module

Bio.SearchIO parser for Exonerate plain text output format.

class Bio.SearchIO.ExonerateIO.exonerate_text.**ExonerateTextParser**(*handle*)

Bases: _BaseExonerateParser

Parser for Exonerate plain text output.

parse_alignment_block(*header*)

Parse alignment block, return query result, hits, hsps.

__abstractmethods__ = frozenset({})

__annotations__ = {'_ALN_MARK': 'Optional[str]'}

class Bio.SearchIO.ExonerateIO.exonerate_text.**ExonerateTextIndexer**(*filename*, ***kwargs*)

Bases: _BaseExonerateIndexer

Indexer class for Exonerate plain text.

get_qresult_id(*pos*)

Return the query ID from the nearest "Query:" line.

get_raw(*offset*)

Return the raw string of a QueryResult object from the given offset.

__abstractmethods__ = frozenset({})

__annotations__ = {'_parser': 'Optional[Type[_BaseExonerateParser]]',
'_query_mark': 'Optional[bytes]'}

Bio.SearchIO.ExonerateIO.exonerate_vulgar module

Bio.SearchIO parser for Exonerate vulgar output format.

class Bio.SearchIO.ExonerateIO.exonerate_vulgar.**ExonerateVulgarParser**(*handle*)

Bases: _BaseExonerateParser

Parser for Exonerate vulgar strings.

parse_alignment_block(*header*)

Parse alignment block for vulgar format, return query results, hits, hsps.

__abstractmethods__ = frozenset({})

__annotations__ = {'_ALN_MARK': 'Optional[str]'}

class Bio.SearchIO.ExonerateIO.exonerate_vulgar.**ExonerateVulgarIndexer**(*filename*, ***kwargs*)

Bases: _BaseExonerateIndexer

Indexer class for exonerate vulgar lines.

get_qresult_id(*pos*)

Return the query ID of the nearest vulgar line.

get_raw(*offset*)

Return the raw bytes string of a QueryResult object from the given offset.

```
__abstractmethods__ = frozenset({})

__annotations__ = {'_parser':
typing.Type[Bio.SearchIO.ExonerateIO._base._BaseExonerateParser], '_query_mark':
'Optional[bytes]'}
```

Module contents

Bio.SearchIO support for Exonerate output formats.

This module adds support for handling Exonerate outputs. Exonerate is a generic tool for pairwise sequence comparison that allows you to align sequences using several different models.

Bio.SearchIO.ExonerateIO was tested on the following Exonerate versions and models:

- version: 2.2
- models: - affine:local - cdna2genome - coding2coding - est2genome - genome2genome - ner - protein2dna - protein2genome - ungapped - ungapped:translated

Although model testing were not exhaustive, ExonerateIO should be able to cope with all Exonerate models. Please file a bug report if you stumble upon an unparsable file.

More information on Exonerate is available on its home page at www.ebi.ac.uk/~guy/exonerate/

Supported Formats

- Plain text alignment - 'exonerate-text' - parsing, indexing
- Vulgar line - 'exonerate-vulgar' - parsing, indexing
- Cigar line - 'exonerate-cigar' - parsing, indexing

On Exonerate, these output formats are not exclusive to one another. For example, you may have both plain text and vulgar output in the same file. ExonerateIO can only handle one of these at a time, however. If you have a file containing both plain text and vulgar lines, for example, you have to pick either 'exonerate-text' or 'exonerate-vulgar' to parse it.

Due to the cigar format specification, many features of the alignments such as introns or frameshifts may be collapsed into a single feature (in this case, they are labelled 'D' for 'deletion'). The parser does not attempt to guess whether the D label it encounters is a real deletion or a collapsed feature. As such, parsing or indexing using 'exonerate-cigar' may yield different results compared to 'exonerate-text' or 'exonerate-vulgar'.

exonerate-text

The plain text output / C4 alignment is the output triggered by the '–showalignemnt' flag. Compared to the two other output formats, this format contains the most information, having the complete query and hit sequences of the alignment.

Here are some examples of the C4 output alignment that ExonerateIO can handle (coordinates not written in scale):

```
1. simple ungapped alignments

       1 : ATGGGCAATATCCTTCGGAAAGGTCAGCAAAT :      56
           ||||||||||||||||||||||||||||||||
 1319275 : ATGGGCAATATCCTTCGGAAAGGTCAGCAAAT : 1319220

2. alignments with frameshifts:
```

(continues on next page)

(continued from previous page)

```
    129 : -TGCCGTTACCAT----GACGAAAGTATTAAT : 160
          -CysArgTyrHis----AspGluSerIleAsn
          #||||||||||||####|||||||||||||||
          #CysArgTyrHis####AspGluSerIleAsn
1234593 : GTGCCGTTACCATCGGTGACGAAAGTATTAAT : 1234630

3. alignments with introns and split codons:

   382 :    {A}                             {CC}AAA                 :    358
         AAA{T}  >>>> Target Intron 3 >>>>  {hr}LysATGAGCGATGAAAATA
         || { }++         55423 bp        ++{  } !  |||  ||||||||||
         AAC{L}gt.........................ag{eu}AspTTGAATGATGAAAATA
 42322 :    {C}                             {TG}GAT                 :  97769

4. alignments with NER blocks

   111 : CAGAAAA--<   31  >--CTGCCCAGAAT--<   10  >--AACGAGCGTTCCG- :    184
         | |||||--< NER 1 >--| ||||| | |--< NER 2 >--|||  | ||||||-
297911 : CTGAAAA--<   29  >--CCGCCCAAAGT--<   13  >--AACTGGAGTTCCG- : 297993
```

ExonerateIO utilizes the HSPFragment model quite extensively to deal with non- ungapped alignments. For any single HSPFragment, if ExonerateIO sees an intron, a NER block, or a frameshift, it will break the fragment into two HSPFragment objects and adjust each of their start and end coordinate appropriately.

You may notice that Exonerate always uses the three letter amino acid codes to display protein sequences. If the protein itself is part of the query sequence, such as in the protein2dna model, ExonerateIO will transform the protein sequence into using one letter codes. This is because the SeqRecord objects that store the sequences are designed for single-letter sequences only. If Exonerate also outputs the underlying nucleotide sequence, it will be saved into an `aln_annotation` entry as a list of triplets.

If the protein sequence is not part of the actual alignment, such as in the est2genome or genome2genome models, ExonerateIO will keep the three letter codes and store them as `aln_annotation` entries. In these cases, the hit and query sequences may be used directly as SeqRecord objects as they are one-letter nucleotide codes. The three-letter protein sequences are then stored as entries in the `aln_annotation` dictionary.

For 'exonerate-text', ExonerateIO provides the following object attributes:

Object	Attribute	Value
QueryResult	description	query sequence description
	id	query sequence ID
	model	alignment model
	program	'exonerate'
Hit	description	hit sequence description
	id	hit sequence ID
HSP	hit_split_codons	list of split codon coordinates in the hit sequence
	score	alignment score
	query_split_codons	list of split codon coordinates in the query sequence
HSPFragment	aln_annotation	alignment similarity string, hit sequence annotation, and/or query sequence annotation
	hit	hit sequence
	hit_end	hit sequence end coordinate
	hit_frame	hit sequence reading frame
	hit_start	hit sequence start coordinate
	hit_strand	hit sequence strand
	query	query sequence
	query_end	query sequence end coordinate
	query_frame	query sequence reading frame
	query_start	query sequence start coordinate
	query_strand	query sequence strand

Note that you can also use the default HSP or HSPFragment properties. For example, to check the intron coordinates of your result you can use the `query_inter_ranges` or `hit_inter_ranges` properties:

```
>>> from Bio import SearchIO
>>> fname = 'Exonerate/exn_22_m_genome2genome.exn'
>>> all_qresult = list(SearchIO.parse(fname, 'exonerate-text'))
>>> hsp = all_qresult[-1][-1][-1]   # last qresult, last hit, last hsp
>>> hsp
HSP(...)
>>> hsp.query_inter_ranges
[(388, 449), (284, 319), (198, 198), (114, 161)]
>>> hsp.hit_inter_ranges
[(487387, 641682), (386207, 487327), (208677, 386123), (71917, 208639)]
```

Here you can see that for both query and hit introns, the coordinates in each tuple is always (start, end) where start <= end. But when you compare each tuple to the next, the coordinates decrease. This is an indication that both the query and hit sequences lie on the minus strand. Exonerate outputs minus strand results in a decreasing manner; the start coordinate is always bigger than the end coordinate. ExonerateIO preserves the fragment ordering as a whole, but uses its own standard to store an individual fragment's start and end coordinates.

You may also notice that the third tuple in `query_inter_ranges` is (198, 198), two exact same numbers. This means that the query sequence does not have any gaps at that position. The gap is only present in the hit sequence, where we see that the third tuple contains (208677, 386123), a gap of about 177k bases.

Another example is to use the `hit_frame_all` and `query_frame_all` to see if there are any frameshifts in your alignment:

```
>>> from Bio import SearchIO
>>> fname = 'Exonerate/exn_22_m_coding2coding_fshifts.exn'
>>> qresult = next(SearchIO.parse(fname, 'exonerate-text'))
```

(continues on next page)

(continued from previous page)

```
>>> hsp = qresult[0][0]      # first hit, first hsp
>>> hsp
HSP(...)
>>> hsp.query_frame_all
[1, 2, 2, 2]
>>> hsp.hit_frame_all
[1, 1, 3, 1]
```

Here you can see that the alignment as a whole has three frameshifts. The first one occurs in the query sequence, after the first fragment (1 -> 2 shift), the second one occurs in the hit sequence, after the second fragment (1 -> 3 shift), and the last one also occurs in the hit sequence, before the last fragment (3 -> 1 shift).

There are other default HSP properties that you can use to ease your workflow. Please refer to the HSP object documentation for more details.

exonerate-vulgar

The vulgar format provides a compact way of representing alignments created by Exonerate. In general, it contains the same information as the plain text output except for the 'model' information and the actual sequences themselves. You can expect that the coordinates obtained from using 'exonerate-text' and 'exonerate-vulgar' to be the same. Both formats also creates HSPFragment using the same triggers: introns, NER blocks, and/or frameshifts.

exonerate-cigar

The cigar format provides an even more compact representation of Exonerate alignments. However, this comes with a cost of losing information. In the cigar format, for example, introns are treated as simple deletions. This makes it impossible for the parser to distinguish between simple deletions or intron regions. As such, 'exonerate-cigar' may produce different sets of coordinates and fragments compared to 'exonerate-vulgar' or 'exonerate-text'.

Bio.SearchIO.HHsuiteIO package

Submodules

Bio.SearchIO.HHsuiteIO.hhsuite2_text module

Bio.SearchIO parser for HHSUITE version 2 and 3 plain text output format.

class Bio.SearchIO.HHsuiteIO.hhsuite2_text.**Hhsuite2TextParser**(*handle*)

Bases: object

Parser for the HHSUITE version 2 and 3 text output.

__init__(*handle*)

Initialize the class.

__iter__()

Iterate over query results - there will only ever be one.

Module contents

Bio.SearchIO support for HHSUITE output formats.

This module adds support for parsing HHSUITE version 2 output.

More information about HHSUITE are available through these links: - Github repository: https://github.com/soedinglab/hh-suite - Wiki: https://github.com/soedinglab/hh-suite/wiki

Bio.SearchIO.HmmerIO package

Submodules

Bio.SearchIO.HmmerIO.hmmer2_text module

Bio.SearchIO parser for HMMER 2 text output.

class Bio.SearchIO.HmmerIO.hmmer2_text.**Hmmer2TextParser**(*handle*)

Bases: object

Iterator for the HMMER 2.0 text output.

__init__(*handle*)

Initialize the class.

__iter__()

Iterate over Hmmer2TextParser, yields query results.

read_next(*rstrip=True*)

Return the next non-empty line, trailing whitespace removed.

push_back(*line*)

Un-read a line that should not be parsed yet.

parse_key_value()

Parse key-value pair separated by colon.

parse_preamble()

Parse HMMER2 preamble.

parse_qresult()

Parse a HMMER2 query block.

parse_hits()

Parse a HMMER2 hit block, beginning with the hit table.

parse_hsps(*hit_placeholders*)

Parse a HMMER2 hsp block, beginning with the hsp table.

parse_hsp_alignments()

Parse a HMMER2 HSP alignment block.

class Bio.SearchIO.HmmerIO.hmmer2_text.**Hmmer2TextIndexer**(**args*, ***kwargs*)

Bases: _BaseHmmerTextIndexer

Indexer for hmmer2-text format.

qresult_start = b'Query'

qresult_end = b'//'

__iter__()

Iterate over Hmmer2TextIndexer; yields query results' key, offsets, 0.

__abstractmethods__ = frozenset({})

__annotations__ = {}

Bio.SearchIO.HmmerIO.hmmer3_domtab module

Bio.SearchIO parser for HMMER domain table output format.

class Bio.SearchIO.HmmerIO.hmmer3_domtab.**Hmmer3DomtabHmmhitParser**(*handle*)

Bases: Hmmer3DomtabParser

HMMER domain table parser using hit coordinates.

Parser for the HMMER domain table format that assumes HMM profile coordinates are hit coordinates.

hmm_as_hit = True

class Bio.SearchIO.HmmerIO.hmmer3_domtab.**Hmmer3DomtabHmmqueryParser**(*handle*)

Bases: Hmmer3DomtabParser

HMMER domain table parser using query coordinates.

Parser for the HMMER domain table format that assumes HMM profile coordinates are query coordinates.

hmm_as_hit = False

__annotations__ = {}

class Bio.SearchIO.HmmerIO.hmmer3_domtab.**Hmmer3DomtabHmmhitIndexer**(*filename*, ***kwargs*)

Bases: *Hmmer3TabIndexer*

HMMER domain table indexer using hit coordinates.

Indexer class for HMMER domain table output that assumes HMM profile coordinates are hit coordinates.

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.SearchIO.HmmerIO.hmmer3_domtab.**Hmmer3DomtabHmmqueryIndexer**(*filename*, ***kwargs*)

Bases: *Hmmer3TabIndexer*

HMMER domain table indexer using query coordinates.

Indexer class for HMMER domain table output that assumes HMM profile coordinates are query coordinates.

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.SearchIO.HmmerIO.hmmer3_domtab.**Hmmer3DomtabHmmhitWriter**(*handle*)

Bases: object

HMMER domain table writer using hit coordinates.

Writer for hmmer3-domtab output format which writes hit coordinates as HMM profile coordinates.

hmm_as_hit = True

__init__(*handle*)

Initialize the class.

write_file(*qresults*)

Write to the handle.

Returns a tuple of how many QueryResult, Hit, and HSP objects were written.

class Bio.SearchIO.HmmerIO.hmmer3_domtab.**Hmmer3DomtabHmmqueryWriter**(*handle*)

Bases: *Hmmer3DomtabHmmhitWriter*

HMMER domain table writer using query coordinates.

Writer for hmmer3-domtab output format which writes query coordinates as HMM profile coordinates.

hmm_as_hit = False

__annotations__ = {}

Bio.SearchIO.HmmerIO.hmmer3_tab module

Bio.SearchIO parser for HMMER table output format.

class Bio.SearchIO.HmmerIO.hmmer3_tab.**Hmmer3TabParser**(*handle*)

Bases: object

Parser for the HMMER table format.

__init__(*handle*)

Initialize the class.

__iter__()

Iterate over Hmmer3TabParser, yields query results.

__annotations__ = {}

class Bio.SearchIO.HmmerIO.hmmer3_tab.**Hmmer3TabIndexer**(*filename*, ***kwargs*)

Bases: SearchIndexer

Indexer class for HMMER table output.

__iter__()

Iterate over the file handle; yields key, start offset, and length.

get_raw(*offset*)

Return the raw bytes string of a QueryResult object from the given offset.

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.SearchIO.HmmerIO.hmmer3_tab.**Hmmer3TabWriter**(*handle*)

Bases: object

Writer for hmmer3-tab output format.

__init__(*handle*)

Initialize the class.

write_file(*qresults*)

Write to the handle.

Returns a tuple of how many QueryResult, Hit, and HSP objects were written.

Bio.SearchIO.HmmerIO.hmmer3_text module

Bio.SearchIO parser for HMMER plain text output format.

class Bio.SearchIO.HmmerIO.hmmer3_text.**Hmmer3TextParser**(*handle*)

Bases: object

Parser for the HMMER 3.0 text output.

__init__(*handle*)

Initialize the class.

__iter__()

Iterate over query results.

class Bio.SearchIO.HmmerIO.hmmer3_text.**Hmmer3TextIndexer**(**args*, ***kwargs*)

Bases: _BaseHmmerTextIndexer

Indexer class for HMMER plain text output.

qresult_start = b'Query: '

qresult_end = b'//'

__iter__()

Iterate over Hmmer3TextIndexer; yields query results' key, offsets, 0.

__abstractmethods__ = frozenset({})

__annotations__ = {}

Module contents

Bio.SearchIO support for HMMER output formats.

This module adds support for parsing HMMER outputs. HMMER is a suite of programs implementing the profile hidden Markov models to find similarity across protein sequences.

Bio.SearchIO.HmmerIO was tested on the following HMMER versions and flavors:

- HMMER3 flavors: hmmscan, hmmsearch, phmmer
- HMMER2 flavors: hmmpfam, hmmsearch

More information on HMMER are available through these links:

- Web page: http://hmmer.org

- User guide: http://eddylab.org/software/hmmer/Userguide.pdf

Supported formats

Bio.SearchIO.HmmerIO supports the following HMMER output formats:

- Plain text, v3.0 - 'hmmer3-text' - parsing, indexing
- Table, v3.0 - 'hmmer3-tab' - parsing, indexing, writing
- Domain table, v3.0 - 'hmmer3-domtab'* - parsing, indexing, writing
- Plain text, v2.x - 'hmmer2-text' - parsing, indexing
- For the domain table output, due to the way HMMER outputs the sequence coordinates, you have to specify what HMMER flavor produced the output as the file format. So instead of using 'hmmer3-domtab', you have to use either 'hmmscan3-domtab', 'hmmsearch3-domtab', or 'phmmer3-domtab' as the file format name.

Note that for all output formats, HMMER uses its own convention of input and output coordinates. It does not use the term 'hit' or 'query', instead it uses 'hmm' or 'ali'. For example, 'hmmfrom' is the start coordinate of the HMM sequence while 'alifrom' is the start coordinate of the protein sequence.

HmmerIO is aware of this different naming scheme and will adjust them accordingly to fit SearchIO's object model. If HmmerIO sees that the output file to parse was written by hmmsearch or phmmer, all 'hmm' coordinates will be the hit coordinates and 'ali' coordinates will be the query coordinates. Conversely, if the HMMER flavor is hmmscan, 'hmm' will be query and 'ali' will be hit.

This is why the 'hmmer3-domtab' format has to be specified with the source HMMER flavor. The parsers need to know which is the hit and which is the query. 'hmmer3-text' has its source program information present in the file, while 'hmmer3-tab' does not output any coordinates. That's why both of these formats do not need direct flavor specification like 'hmmer3-domtab'.

Also note that when using the domain table format writers, it will use HMMER's naming convention ('hmm' and 'ali') so the files you write will be similar to files written by a real HMMER program.

hmmer2-text and hmmer3-text

The parser for HMMER 3.0 plain text output can parse output files with alignment blocks (default) or without (with the '–noali' flag). If the alignment blocks are present, you can also parse files with variable alignment width (using the '–notextw' or '–textw' flag).

The following SearchIO objects attributes are provided. Rows marked with '*' denotes attributes not available in the hmmer2-text format:

Object	Attribute	Value
QueryResult	accession	accession (if present)
	description	query sequence description
	id	query sequence ID
	program	HMMER flavor
	seq_len*	full length of query sequence
	target	target search database
	version	BLAST version
Hit	bias*	hit-level bias
	bitscore	hit-level score
	description	hit sequence description

contin

Table 4 – continued from previous page

Object	Attribute	Value
HSP	domain_exp_num*	expected number of domains in the hit (exp column)
	domain_obs_num	observed number of domains in the hit (N column)
	evalue	hit-level e-value
	id	hit sequence ID
	is_included*	boolean, whether the hit is in the inclusion threshold or not
	acc_avg*	expected accuracy per alignment residue (acc column)
	bias*	hsp-level bias
	bitscore	hsp-level score
	domain_index	the domain index set by HMMER
	env_end*	end coordinate of the envelope
	env_endtype*	envelope end types (e.g. '[]', '..', '[.', etc.)
	env_start*	start coordinate of the envelope
	evalue	hsp-level independent e-value
	evalue_cond*	hsp-level conditional e-value
	hit_endtype	hit sequence end types
	is_included*	boolean, whether the hit of the hsp is in the inclusion threshold
	query_endtype	query sequence end types
HSPFragment (also via HSP)	aln_annotation	alignment similarity string and other annotations (e.g. PP, CS)
	aln_span	length of alignment fragment
	hit	hit sequence
	hit_end	hit sequence end coordinate, may be 'hmmto' or 'alito' depending on the HMM
	hit_start	hit sequence start coordinate, may be 'hmmfrom' or 'alifrom' depending on the
	hit_strand	hit sequence strand
	query	query sequence
	query_end	query sequence end coordinate, may be 'hmmto' or 'alito' depending on the HM
	query_start	query sequence start coordinate, may be 'hmmfrom' or 'alifrom' depending on
	query_strand	query sequence strand

hmmer3-tab

The following SearchIO objects attributes are provided:

Object	Attribute	Column / Value
QueryResult	accession	query accession (if present)
	description	query sequence description
	id	query name
Hit	accession	hit accession
	bias	hit-level bias
	bitscore	hit-level score
	description	hit sequence description
	cluster_num	clu column
	domain_exp_num	exp column
	domain_included_num	inc column
	domain_obs_num	dom column
	domain_reported_num	rep column
	env_num	env column
	evalue	hit-level evalue
	id	target name
	overlap_num	ov column
	region_num	reg column
HSP	bias	bias of the best domain
	bitscore	bitscore of the best domain
	evalue	evalue of the best domain

hmmer3-domtab

To parse domain table files, you must use the HMMER flavor that produced the file. So instead of using 'hmmer3-domtab', use either 'hmmsearch3-domtab', 'hmmscan3-domtab', or 'phmmer3-domtab'.

The following SearchIO objects attributes are provided:

Object	Attribute	Value
QueryResult	accession	accession
	description	query sequence description
	id	query sequence ID
	seq_len	full length of query sequence
Hit	accession	accession
	bias	hit-level bias
	bitscore	hit-level score
	description	hit sequence description
	evalue	hit-level e-value
	id	hit sequence ID
	seq_len	length of hit sequence or HMM
HSP	acc_avg	expected accuracy per alignment residue (acc column)
	bias	hsp-level bias
	bitscore	hsp-level score
	domain_index	the domain index set by HMMER
	env_end	end coordinate of the envelope
	env_start	start coordinate of the envelope
	evalue	hsp-level independent e-value
	evalue_cond	hsp-level conditional e-value
HSPFragment (also via HSP)	hit_end	hit sequence end coordinate, may be 'hmmto' or 'alito' depending on the HMMER flavor
	hit_start	hit sequence start coordinate, may be 'hmmfrom' or 'alifrom' depending on the HMMER flavor
	hit_strand	hit sequence strand
	query_end	query sequence end coordinate, may be 'hmmto' or 'alito' depending on the HMMER flavor
	query_start	query sequence start coordinate, may be 'hmmfrom' or 'alifrom' depending on the HMMER flavor
	query_strand	query sequence strand

Bio.SearchIO.InterproscanIO package

Submodules

Bio.SearchIO.InterproscanIO.interproscan_xml module

Bio.SearchIO parser for InterProScan XML output formats.

class Bio.SearchIO.InterproscanIO.interproscan_xml.**InterproscanXmlParser**(*handle*)

Bases: object

Parser for the InterProScan XML format.

__init__(*handle*)

Initialize the class.

__iter__()

Iterate qresults.

Module contents

Bio.SearchIO support for InterProScan output formats.

This module adds support for parsing InterProScan XML output. The InterProScan is available as a command line program or on EMBL-EBI's web page. Bio.SearchIO.InterproscanIO was tested on the following version:

- versions: 5.26-65.0 (interproscan-model-2.1.xsd)

More information about InterProScan are available through these links: - Publication: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3998142/ - Web interface: https://www.ebi.ac.uk/interpro/search/sequence-search - Documentation: https://github.com/ebi-pf-team/interproscan/wiki

Supported format

Bio.SearchIO.InterproscanIO supports the following format:

- XML - 'interproscan-xml' - parsing

interproscan-xml

The interproscan-xml parser follows the InterProScan XML described here: https://github.com/ebi-pf-team/interproscan/wiki/OutputFormats

Object	Attribute	XML Element
QueryResult	target	`InterPro`
	program	`InterProScan`
	version	`protein-matches.interproscan-version`
Hit	accession	`signature.name`
	id	`signature.ac`
	description	`signature.desc`
	dbxrefs	`IPR:entry.ac go-xref.id pathway-xref.db:pathway-xref.id`
	attributes ['Target'] ['Target version'] ['Hit type']	`*-match / *-location signature-library-release.library signature-library-release.version`
HSP	bitscore	`*-location.score`
	evalue	`*-location.evalue`
HSPFragment (also via HSP)	query_start	`*-location.start`
	query_end	`*-location.end`
	hit_start	`*-location.hmm-start`
	hit_end	`*-location.hmm-end`
	query	`sequence`

InterProScan XML files may contain a match with multiple locations or multiple matches to the same protein with a single location. In both cases, the match is uniquely stored as a HIT object and the locations as HSP objects.

`HSP.*start == *start - 1` (Since every start position is 0-based in Biopython)

`HSP.aln_span == query-end - query-start`

The types of matches or locations (eg. hmmer3-match, hmmer3-location, coils-match, panther-location) are stored in hit.attributes['Hit type']. For instance, for every 'phobious-match', there will be a 'phobious-location'. Therefore, Hit.type will store the string excluding '-match' or '-location' ('phobious', in this example).

Submodules

Bio.SearchIO.BlatIO module

Bio.SearchIO parser for BLAT output formats.

This module adds support for parsing BLAT outputs. BLAT (BLAST-Like Alignment Tool) is a sequence similarity search program initially built for annotating the human genome.

Bio.SearchIO.BlastIO was tested using standalone BLAT version 34, psLayout version 3. It should be able to parse psLayout version 4 without problems.

More information on BLAT is available from these sites:

- Publication: http://genome.cshlp.org/content/12/4/656
- User guide: http://genome.ucsc.edu/goldenPath/help/blatSpec.html
- Source download: http://www.soe.ucsc.edu/~kent/src
- Executable download: http://hgdownload.cse.ucsc.edu/admin/exe/
- Blat score calculation: http://genome.ucsc.edu/FAQ/FAQblat.html#blat4

Supported Formats

BlatIO supports parsing, indexing, and writing for both PSL and PSLX output formats, with or without header. To parse, index, or write PSLX files, use the 'pslx' keyword argument and set it to True.

```
>>> # blat-psl defaults to PSL files
>>> from Bio import SearchIO
>>> psl = 'Blat/psl_34_004.psl'
>>> qresult = SearchIO.read(psl, 'blat-psl')
>>> qresult
QueryResult(id='hg19_dna', 10 hits)
```

```
>>> # set the pslx flag to parse PSLX files
>>> pslx = 'Blat/pslx_34_004.pslx'
>>> qresult = SearchIO.read(pslx, 'blat-psl', pslx=True)
>>> qresult
QueryResult(id='hg19_dna', 10 hits)
```

For parsing and indexing, you do not need to specify whether the file has a header or not. For writing, if you want to write a header, you can set the 'header' keyword argument to True. This will write a 'psLayout version 3' header to your output file.

```
>>> from Bio import SearchIO
>>> qresult = SearchIO.read(psl, "blat-psl")
>>> SearchIO.write(qresult, "example.psl", "blat-psl", header=True)
(1, 10, 19, 23)
>>> import os
>>> os.remove("example.psl")
```

Note that the number of HSPFragments written may exceed the number of HSP objects. This is because in PSL files, it is possible to have single matches consisting of noncontiguous sequence fragments. This is where the HSPFragment object comes into play. These fragments are grouped into a single HSP because they share the same statistics (e.g.

match numbers, BLAT score, etc.). However, they do not share the same sequence attributes, such as the start and end coordinates, making them distinct objects.

In addition to parsing PSL(X) files, BlatIO also computes the percent identities and scores of your search results. This is done using the calculation formula posted here: http://genome.ucsc.edu/FAQ/FAQblat.html#blat4. It mimics the score and percent identity calculation done by UCSC's web BLAT service.

Since BlatIO parses the file in a single pass, it expects all results from the same query to be in consecutive rows. If the results from one query are spread in nonconsecutive rows, BlatIO will consider them to be separate QueryResult objects.

In most cases, the PSL(X) format uses the same coordinate system as Python (zero-based, half open). These coordinates are anchored on the plus strand. However, if the query aligns on the minus strand, BLAT will anchor the qStarts coordinates on the minus strand instead. BlatIO is aware of this, and will re-anchor the qStarts coordinates to the plus strand whenever it sees a minus strand query match. Conversely, when you write out to a PSL(X) file, BlatIO will reanchor qStarts to the minus strand again.

BlatIO provides the following attribute-column mapping:

Object	Attribute	Column Name, Value
QueryResult	id	Q name, query sequence ID
	seq_len	Q size, query sequence full length
Hit	id	T name, hit sequence ID
	seq_len	T size, hit sequence full length
HSP	hit_end	T end, end coordinate of the last hit fragment
	hit_gap_num	T gap bases, number of bases inserted in hit
	hit_gapopen_num	T gap count, number of hit gap inserts
	hit_span_all	blockSizes, sizes of each fragment
	hit_start	T start, start coordinate of the first hit fragment
	hit_start_all	tStarts, start coordinate of each hit fragment
	match_num	match, number of non-repeat matches
	mismatch_num	mismatch, number of mismatches
	match_rep_num	rep. match, number of matches that are part of repeats
	n_num	N's, number of N bases
	query_end	Q end, end coordinate of the last
	query_gap_num	query fragment Q gap bases, number of bases inserted in query
	query_gapopen_num	Q gap count, number of query gap inserts
	query_span_all	blockSizes, sizes of each fragment
	query_start	Q start, start coordinate of the first query block
	query_start_all	qStarts, start coordinate of each query fragment
	len[1]	block count, the number of blocks in the alignment
HSPFragment	hit	hit sequence, if present
	hit_strand	strand, hit sequence strand
	query	query sequence, if present
	query_strand	strand, query sequence strand

In addition to the column mappings above, BlatIO also provides the following object attributes:

[1] You can obtain the number of blocks / fragments in the HSP by invoking `len` on the HSP

Object	Attribute	Value
HSP	gapopen_num	**Q gap count + T gap count, total** number of gap openings
	ident_num	matches + repmatches, total number of identical residues
	ident_pct	percent identity, calculated using UCSC's formula
	query_is_protein	boolean, whether the query sequence is a protein
	score	HSP score, calculated using UCSC's formula

Finally, the default HSP and HSPFragment properties are also provided. See the HSP and HSPFragment documentation for more details on these properties.

class Bio.SearchIO.BlatIO.**BlatPslParser**(*handle*, *pslx=False*)

Bases: object

Parser for the BLAT PSL format.

__init__(*handle*, *pslx=False*)

Initialize the class.

__iter__()

Iterate over BlatPslParser, yields query results.

class Bio.SearchIO.BlatIO.**BlatPslIndexer**(*filename*, *pslx=False*)

Bases: SearchIndexer

Indexer class for BLAT PSL output.

__init__(*filename*, *pslx=False*)

Initialize the class.

__iter__()

Iterate over the file handle; yields key, start offset, and length.

get_raw(*offset*)

Return raw bytes string of a QueryResult object from the given offset.

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.SearchIO.BlatIO.**BlatPslWriter**(*handle*, *header=False*, *pslx=False*)

Bases: object

Writer for the blat-psl format.

__init__(*handle*, *header=False*, *pslx=False*)

Initialize the class.

write_file(*qresults*)

Write query results to file.

Bio.SearchIO.FastaIO module

Bio.SearchIO support for Bill Pearson's FASTA tools.

This module adds support for parsing FASTA outputs. FASTA is a suite of programs that finds regions of local or global similarity between protein or nucleotide sequences, either by searching databases or identifying local duplications.

Bio.SearchIO.FastaIO was tested on the following FASTA flavors and versions:

- flavors: fasta, ssearch, tfastx
- versions: 35, 36

Other flavors and/or versions may introduce some bugs. Please file a bug report if you see such problems to Biopython's bug tracker.

More information on FASTA are available through these links:

- Website: http://fasta.bioch.virginia.edu/fasta_www2/fasta_list2.shtml
- User guide: http://fasta.bioch.virginia.edu/fasta_www2/fasta_guide.pdf

Supported Formats

Bio.SearchIO.FastaIO supports parsing and indexing FASTA outputs triggered by the -m 10 flag. Other formats that mimic other programs (e.g. the BLAST tabular format using the -m 8 flag) may be parseable but using SearchIO's other parsers (in this case, using the 'blast-tab' parser).

fasta-m10

Note that in FASTA -m 10 outputs, HSPs from different strands are considered to be from different hits. They are listed as two separate entries in the hit table. FastaIO recognizes this and will group HSPs with the same hit ID into a single Hit object, regardless of strand.

FASTA also sometimes output extra sequences adjacent to the HSP match. These extra sequences are discarded by FastaIO. Only regions containing the actual sequence match are extracted.

The following object attributes are provided:

Object	Attribute	Value
QueryResult	description	query sequence description
	id	query sequence ID
	program	FASTA flavor
	seq_len	full length of query sequence
	target	target search database
	version	FASTA version
Hit	seq_len	full length of the hit sequence
HSP	bitscore	*_bits line
	evalue	*_expect line
	ident_pct	*_ident line
	init1_score	*_init1 line
	initn_score	*_initn line
	opt_score	*_opt line, *_s-w opt line
	pos_pct	*_sim line
	sw_score	*_score line
	z_score	*_z-score line
HSPFragment (also via HSP)	aln_annotation	al_cons block, if present
	hit	hit sequence
	hit_end	hit sequence end coordinate
	hit_start	hit sequence start coordinate
	hit_strand	hit sequence strand
	query	query sequence
	query_end	query sequence end coordinate
	query_start	query sequence start coordinate
	query_strand	query sequence strand

class Bio.SearchIO.FastaIO.**FastaM10Parser**(*handle*, *_FastaM10Parser__parse_hit_table=False*)

Bases: object

Parser for Bill Pearson's FASTA suite's -m 10 output.

__init__(*handle*, *_FastaM10Parser__parse_hit_table=False*)

Initialize the class.

__iter__()

Iterate over FastaM10Parser object yields query results.

class Bio.SearchIO.FastaIO.**FastaM10Indexer**(*filename*)

Bases: SearchIndexer

Indexer class for Bill Pearson's FASTA suite's -m 10 output.

__init__(*filename*)

Initialize the class.

__iter__()

Iterate over FastaM10Indexer; yields query results' keys, start offsets, offset lengths.

get_raw(*offset*)

Return the raw record from the file as a bytes string.

__abstractmethods__ = frozenset({})

__annotations__ = {}

Module contents

Biopython interface for sequence search program outputs.

The SearchIO submodule provides parsers, indexers, and writers for outputs from various sequence search programs. It provides an API similar to SeqIO and AlignIO, with the following main functions: `parse`, `read`, `to_dict`, `index`, `index_db`, `write`, and `convert`.

SearchIO parses a search output file's contents into a hierarchy of four nested objects: QueryResult, Hit, HSP, and HSPFragment. Each of them models a part of the search output file:

- QueryResult represents a search query. This is the main object returned by the input functions and it contains all other objects.
- Hit represents a database hit,
- HSP represents high-scoring alignment region(s) in the hit,
- HSPFragment represents a contiguous alignment within the HSP

In addition to the four objects above, SearchIO is also tightly integrated with the SeqRecord objects (see SeqIO) and MultipleSeqAlignment objects (see AlignIO). SeqRecord objects are used to store the actual matching hit and query sequences, while MultipleSeqAlignment objects store the alignment between them.

A detailed description of these objects' features and their example usages are available in their respective documentations.

Input

The main function for parsing search output files is Bio.SearchIO.parse(...). This function parses a given search output file and returns a generator object that yields one QueryResult object per iteration.

`parse` takes two arguments: 1) a file handle or a filename of the input file (the search output file) and 2) the format name.

```
>>> from Bio import SearchIO
>>> for qresult in SearchIO.parse('Blast/mirna.xml', 'blast-xml'):
...     print("%s %s" % (qresult.id, qresult.description))
...
33211 mir_1
33212 mir_2
33213 mir_3
```

SearchIO also provides the Bio.SearchIO.read(...) function, which is intended for use on search output files containing only one query. `read` returns one QueryResult object and will raise an exception if the source file contains more than one query:

```
>>> qresult = SearchIO.read('Blast/xml_2226_blastp_004.xml', 'blast-xml')
>>> print("%s %s" % (qresult.id, qresult.description))
...
gi|11464971:4-101 pleckstrin [Mus musculus]
```

```
>>> SearchIO.read('Blast/mirna.xml', 'blast-xml')
Traceback (most recent call last):
...
ValueError: ...
```

For accessing search results of large output files, you may use the indexing functions Bio.SearchIO.index(...) or Bio.SearchIO.index_db(...). They have a similar interface to their counterparts in SeqIO and AlignIO, with the addition of optional, format-specific keyword arguments.

Output

SearchIO has write support for several formats, accessible from the Bio.SearchIO.write(...) function. This function returns a tuple of four numbers: the number of QueryResult, Hit, HSP, and HSPFragment written:

```
qresults = SearchIO.parse('Blast/mirna.xml', 'blast-xml')
SearchIO.write(qresults, 'results.tab', 'blast-tab')
<stdout> (3, 239, 277, 277)
```

Note that different writers may require different attribute values of the SearchIO objects. This limits the scope of writable search results to search results possessing the required attributes.

For example, the writer for HMMER domain table output requires the conditional e-value attribute from each HSP object, among others. If you try to write to the HMMER domain table format and your HSPs do not have this attribute, an exception will be raised.

Conversion

SearchIO provides a shortcut function Bio.SearchIO.convert(...) to convert a given file into another format. Under the hood, `convert` simply parses a given output file and writes it to another using the `parse` and `write` functions.

Note that the same restrictions found in Bio.SearchIO.write(...) apply to the convert function as well.

Conventions

The main goal of creating SearchIO is to have a common, easy to use interface across different search output files. As such, we have also created some conventions / standards for SearchIO that extend beyond the common object model. These conventions apply to all files parsed by SearchIO, regardless of their individual formats.

Python-style sequence coordinates

When storing sequence coordinates (start and end values), SearchIO uses the Python-style slice convention: zero-based and half-open intervals. For example, if in a BLAST XML output file the start and end coordinates of an HSP are 10 and 28, they would become 9 and 28 in SearchIO. The start coordinate becomes 9 because Python indices start from zero, while the end coordinate remains 28 as Python slices omit the last item in an interval.

Beside giving you the benefits of standardization, this convention also makes the coordinates usable for slicing sequences. For example, given a full query sequence and the start and end coordinates of an HSP, one can use the coordinates to extract part of the query sequence that results in the database hit.

When these objects are written to an output file using SearchIO.write(...), the coordinate values are restored to their respective format's convention. Using the example above, if the HSP would be written to an XML file, the start and end coordinates would become 10 and 28 again.

Sequence coordinate order

Some search output formats reverse the start and end coordinate sequences according to the sequence's strand.

In SearchIO, start coordinates are always smaller than the end coordinates, regardless of their originating strand. This ensures consistency when using the coordinates to slice full sequences.

Note that this coordinate order convention is only enforced in the HSPFragment level. If an HSP object has several HSPFragment objects, each individual fragment will conform to this convention. But the order of the fragments within the HSP object follows what the search output file uses.

Similar to the coordinate style convention, the start and end coordinates' order are restored to their respective formats when the objects are written using Bio.SearchIO.write(...).

Frames and strand values

SearchIO only allows -1, 0, 1 and None as strand values. For frames, the only allowed values are integers from -3 to 3 (inclusive) and None. Both of these are standard Biopython conventions.

Supported Formats

Below is a list of search program output formats supported by SearchIO.

Support for parsing, indexing, and writing:

- **blast-tab - BLAST+ tabular output. Both variants without comments** (-m 6 flag) and with comments (-m 7 flag) are supported.
- blast-xml - BLAST+ XML output.
- **blat-psl - The default output of BLAT (PSL format). Variants with or** without header are both supported. PSLX (PSL + sequences) is also supported.
- hmmer3-tab - HMMER3 table output.
- **hmmer3-domtab - HMMER3 domain table output. When using this format, the** program name has to be specified. For example, for parsing hmmscan output, the name would be 'hmmscan-domtab'.

Support for parsing and indexing:

- exonerate-text - Exonerate plain text output.
- exonerate-vulgar - Exonerate vulgar line.
- exonerate-cigar - Exonerate cigar line.
- fasta-m10 - Bill Pearson's FASTA -m 10 output.
- **hmmer3-text - HMMER3 regular text output format. Supported HMMER3** subprograms are hmmscan, hmmsearch, and phmmer.
- **hmmer2-text - HMMER2 regular text output format. Supported HMMER2** subprograms are hmmpfam, hmmsearch.

Support for parsing:

- hhsuite2-text - HHSUITE plain text output.

Each of these formats have different keyword arguments available for use with the main SearchIO functions. More details and examples are available in each of the format's documentation.

Bio.SearchIO.**read**(*handle*, *format=None*, ***kwargs*)

Turn a search output file containing one query into a single QueryResult.

- handle - Handle to the file, or the filename as a string.
- format - Lower case string denoting one of the supported formats.
- kwargs - Format-specific keyword arguments.

`read` is used for parsing search output files containing exactly one query:

```
>>> from Bio import SearchIO
>>> qresult = SearchIO.read('Blast/xml_2226_blastp_004.xml', 'blast-xml')
>>> print("%s %s" % (qresult.id, qresult.description))
...
gi|11464971:4-101 pleckstrin [Mus musculus]
```

If the given handle has no results, an exception will be raised:

```
>>> from Bio import SearchIO
>>> qresult = SearchIO.read('Blast/tab_2226_tblastn_002.txt', 'blast-tab')
Traceback (most recent call last):
...
ValueError: No query results found in handle
```

Similarly, if the given handle has more than one result, an exception will be raised:

```
>>> from Bio import SearchIO
>>> qresult = SearchIO.read('Blast/tab_2226_tblastn_001.txt', 'blast-tab')
Traceback (most recent call last):
...
ValueError: More than one query result found in handle
```

Like `parse`, `read` may also accept keyword argument(s) depending on the search output file format.

Bio.SearchIO.**parse**(*handle*, *format=None*, ***kwargs*)

Iterate over search tool output file as QueryResult objects.

Arguments:

- handle - Handle to the file, or the filename as a string.
- format - Lower case string denoting one of the supported formats.
- kwargs - Format-specific keyword arguments.

This function is used to iterate over each query in a given search output file:

```
>>> from Bio import SearchIO
>>> qresults = SearchIO.parse('Blast/mirna.xml', 'blast-xml')
>>> qresults
<generator object ...>
>>> for qresult in qresults:
...     print("Search %s has %i hits" % (qresult.id, len(qresult)))
...
Search 33211 has 100 hits
Search 33212 has 44 hits
Search 33213 has 95 hits
```

Depending on the file format, `parse` may also accept additional keyword argument(s) that modifies the behavior of the format parser. Here is a simple example, where the keyword argument enables parsing of a commented BLAST tabular output file:

```
>>> from Bio import SearchIO
>>> for qresult in SearchIO.parse('Blast/mirna.tab', 'blast-tab', comments=True):
...     print("Search %s has %i hits" % (qresult.id, len(qresult)))
...
Search 33211 has 100 hits
Search 33212 has 44 hits
Search 33213 has 95 hits
```

Bio.SearchIO.**to_dict**(*qresults*, *key_function=None*)

Turn a QueryResult iterator or list into a dictionary.

- qresults - Iterable returning QueryResult objects.
- **key_function - Optional callback function which when given a**
  QueryResult object should return a unique key for the dictionary. Defaults to using .id of the result.

This function enables access of QueryResult objects from a single search output file using its identifier.

```
>>> from Bio import SearchIO
>>> qresults = SearchIO.parse('Blast/wnts.xml', 'blast-xml')
>>> search_dict = SearchIO.to_dict(qresults)
>>> list(search_dict)
['gi|195230749:301-1383', 'gi|325053704:108-1166', ..., 'gi|53729353:216-1313']
>>> search_dict['gi|156630997:105-1160']
QueryResult(id='gi|156630997:105-1160', 5 hits)
```

By default, the dictionary key is the QueryResult's string ID. This may be changed by supplying a callback function that returns the desired identifier. Here is an example using a function that removes the 'gi|' part in the beginning of the QueryResult ID.

```
>>> from Bio import SearchIO
>>> qresults = SearchIO.parse('Blast/wnts.xml', 'blast-xml')
>>> key_func = lambda qresult: qresult.id.split('|')[1]
>>> search_dict = SearchIO.to_dict(qresults, key_func)
>>> list(search_dict)
['195230749:301-1383', '325053704:108-1166', ..., '53729353:216-1313']
>>> search_dict['156630997:105-1160']
QueryResult(id='gi|156630997:105-1160', 5 hits)
```

Note that the callback function does not change the QueryResult's ID value. It only changes the key value used to retrieve the associated QueryResult.

As this function loads all QueryResult objects into memory, it may be unsuitable for dealing with files containing many queries. In that case, it is recommended that you use either `index` or `index_db`.

Since Python 3.7, the default dict class maintains key order, meaning this dictionary will reflect the order of records given to it. For CPython and PyPy, this was already implemented for Python 3.6, so effectively you can always assume the record order is preserved.

Bio.SearchIO.**index**(*filename*, *format=None*, *key_function=None*, ***kwargs*)

Indexes a search output file and returns a dictionary-like object.

- filename - string giving name of file to be indexed

- format - Lower case string denoting one of the supported formats.
- **key_function - Optional callback function which when given a** QueryResult should return a unique key for the dictionary.
- kwargs - Format-specific keyword arguments.

Index returns a pseudo-dictionary object with QueryResult objects as its values and a string identifier as its keys. The function is mainly useful for dealing with large search output files, as it enables access to any given QueryResult object much faster than using parse or read.

Index works by storing in-memory the start locations of all queries in a file. When a user requests access to the query, this function will jump to its start position, parse the whole query, and return it as a QueryResult object:

```
>>> from Bio import SearchIO
>>> search_idx = SearchIO.index('Blast/wnts.xml', 'blast-xml')
>>> search_idx
SearchIO.index('Blast/wnts.xml', 'blast-xml', key_function=None)
>>> sorted(search_idx)
['gi|156630997:105-1160', 'gi|195230749:301-1383', ..., 'gi|53729353:216-1313']
>>> search_idx['gi|195230749:301-1383']
QueryResult(id='gi|195230749:301-1383', 5 hits)
>>> search_idx.close()
```

If the file is BGZF compressed, this is detected automatically. Ordinary GZIP files are not supported:

```
>>> from Bio import SearchIO
>>> search_idx = SearchIO.index('Blast/wnts.xml.bgz', 'blast-xml')
>>> search_idx
SearchIO.index('Blast/wnts.xml.bgz', 'blast-xml', key_function=None)
>>> search_idx['gi|195230749:301-1383']
QueryResult(id='gi|195230749:301-1383', 5 hits)
>>> search_idx.close()
```

You can supply a custom callback function to alter the default identifier string. This function should accept as its input the QueryResult ID string and return a modified version of it.

```
>>> from Bio import SearchIO
>>> key_func = lambda id: id.split('|')[1]
>>> search_idx = SearchIO.index('Blast/wnts.xml', 'blast-xml', key_func)
>>> search_idx
SearchIO.index('Blast/wnts.xml', 'blast-xml', key_function=<function <lambda> at ...
↪>)
>>> sorted(search_idx)
['156630997:105-1160', ..., '371502086:108-1205', '53729353:216-1313']
>>> search_idx['156630997:105-1160']
QueryResult(id='gi|156630997:105-1160', 5 hits)
>>> search_idx.close()
```

Note that the callback function does not change the QueryResult's ID value. It only changes the key value used to retrieve the associated QueryResult.

Bio.SearchIO.**index_db**(*index_filename*, *filenames=None*, *format=None*, *key_function=None*, ***kwargs*)

Indexes several search output files into an SQLite database.

- index_filename - The SQLite filename.

- **filenames - List of strings specifying file(s) to be indexed, or when**
  indexing a single file this can be given as a string. (optional if reloading an existing index, but must match)
- **format - Lower case string denoting one of the supported formats.**
  (optional if reloading an existing index, but must match)
- **key_function - Optional callback function which when given a**
  QueryResult identifier string should return a unique key for the dictionary.
- kwargs - Format-specific keyword arguments.

The `index_db` function is similar to `index` in that it indexes the start position of all queries from search output files. The main difference is instead of storing these indices in-memory, they are written to disk as an SQLite database file. This allows the indices to persist between Python sessions. This enables access to any queries in the file without any indexing overhead, provided it has been indexed at least once.

```
>>> from Bio import SearchIO
>>> idx_filename = ":memory:" # Use a real filename, this is in RAM only!
>>> db_idx = SearchIO.index_db(idx_filename, 'Blast/mirna.xml', 'blast-xml')
>>> sorted(db_idx)
['33211', '33212', '33213']
>>> db_idx['33212']
QueryResult(id='33212', 44 hits)
>>> db_idx.close()
```

`index_db` can also index multiple files and store them in the same database, making it easier to group multiple search files and access them from a single interface.

```
>>> from Bio import SearchIO
>>> idx_filename = ":memory:" # Use a real filename, this is in RAM only!
>>> files = ['Blast/mirna.xml', 'Blast/wnts.xml']
>>> db_idx = SearchIO.index_db(idx_filename, files, 'blast-xml')
>>> sorted(db_idx)
['33211', '33212', '33213', 'gi|156630997:105-1160', ..., 'gi|53729353:216-1313']
>>> db_idx['33212']
QueryResult(id='33212', 44 hits)
>>> db_idx.close()
```

One common example where this is helpful is if you had a large set of query sequences (say ten thousand) which you split into ten query files of one thousand sequences each in order to run as ten separate BLAST jobs on a cluster. You could use `index_db` to index the ten BLAST output files together for seamless access to all the results as one dictionary.

Note that ':memory:' rather than an index filename tells SQLite to hold the index database in memory. This is useful for quick tests, but using the Bio.SearchIO.index(...) function instead would use less memory.

BGZF compressed files are supported, and detected automatically. Ordinary GZIP compressed files are not supported.

See also Bio.SearchIO.index(), Bio.SearchIO.to_dict(), and the Python module glob which is useful for building lists of files.

`Bio.SearchIO.`**`write`**(*qresults*, *handle*, *format=None*, ***kwargs*)

Write QueryResult objects to a file in the given format.

- **qresults - An iterator returning QueryResult objects or a single**
  QueryResult object.

- handle - Handle to the file, or the filename as a string.
- format - Lower case string denoting one of the supported formats.
- kwargs - Format-specific keyword arguments.

The `write` function writes QueryResult object(s) into the given output handle / filename. You can supply it with a single QueryResult object or an iterable returning one or more QueryResult objects. In both cases, the function will return a tuple of four values: the number of QueryResult, Hit, HSP, and HSPFragment objects it writes to the output file:

```
from Bio import SearchIO
qresults = SearchIO.parse('Blast/mirna.xml', 'blast-xml')
SearchIO.write(qresults, 'results.tab', 'blast-tab')
<stdout> (3, 239, 277, 277)
```

The output of different formats may be adjusted using the format-specific keyword arguments. Here is an example that writes BLAT PSL output file with a header:

```
from Bio import SearchIO
qresults = SearchIO.parse('Blat/psl_34_001.psl', 'blat-psl')
SearchIO.write(qresults, 'results.tab', 'blat-psl', header=True)
<stdout> (2, 13, 22, 26)
```

Bio.SearchIO.**convert**(*in_file*, *in_format*, *out_file*, *out_format*, *in_kwargs=None*, *out_kwargs=None*)

Convert between two search output formats, return number of records.

- in_file - Handle to the input file, or the filename as string.
- in_format - Lower case string denoting the format of the input file.
- out_file - Handle to the output file, or the filename as string.
- out_format - Lower case string denoting the format of the output file.
- in_kwargs - Dictionary of keyword arguments for the input function.
- out_kwargs - Dictionary of keyword arguments for the output function.

The convert function is a shortcut function for `parse` and `write`. It has the same return type as `write`. Format-specific arguments may be passed to the convert function, but only as dictionaries.

Here is an example of using `convert` to convert from a BLAST+ XML file into a tabular file with comments:

```
from Bio import SearchIO
in_file = 'Blast/mirna.xml'
in_fmt = 'blast-xml'
out_file = 'results.tab'
out_fmt = 'blast-tab'
out_kwarg = {'comments': True}
SearchIO.convert(in_file, in_fmt, out_file, out_fmt, out_kwargs=out_kwarg)
<stdout> (3, 239, 277, 277)
```

Given that different search output file provide different statistics and different level of details, the convert function is limited only to converting formats that have the same statistics and for conversion to formats with the same level of detail, or less.

For example, converting from a BLAST+ XML output to a HMMER table file is not possible, as these are two search programs with different kinds of statistics. In theory, you may provide the necessary values required by

the HMMER table file (e.g. conditional e-values, envelope coordinates, etc). However, these values are likely to hold little meaning as they are not true HMMER-computed values.

Another example is converting from BLAST+ XML to BLAST+ tabular file. This is possible, as BLAST+ XML provide all the values necessary to create a BLAST+ tabular file. However, the reverse conversion may not be possible. There are more details covered in the XML file that are not found in a tabular file (e.g. the lambda and kappa values)

28.1.29 Bio.SeqIO package

Submodules

Bio.SeqIO.AbiIO module

Bio.SeqIO parser for the ABI format.

ABI is the format used by Applied Biosystem's sequencing machines to store sequencing results.

For more details on the format specification, visit: http://www6.appliedbiosystems.com/support/software_community/ABIF_File_Format.pdf

class Bio.SeqIO.AbiIO.**AbiIterator**(*source*, *trim=False*)

Bases: *SequenceIterator*

Parser for Abi files.

__init__(*source*, *trim=False*)

Return an iterator for the Abi file format.

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

iterate(*handle*)

Parse the file and generate SeqRecord objects.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

Bio.SeqIO.AceIO module

Bio.SeqIO support for the "ace" file format.

You are expected to use this module via the Bio.SeqIO functions. See also the Bio.Sequencing.Ace module which offers more than just accessing the contig consensus sequences in an ACE file as SeqRecord objects.

Bio.SeqIO.AceIO.**AceIterator**(*source*)

Return SeqRecord objects from an ACE file.

This uses the Bio.Sequencing.Ace module to do the hard work. Note that by iterating over the file in a single pass, we are forced to ignore any WA, CT, RT or WR footer tags.

Ace files include the base quality for each position, which are taken to be PHRED style scores. Just as if you had read in a FASTQ or QUAL file using PHRED scores using Bio.SeqIO, these are stored in the SeqRecord's letter_annotations dictionary under the "phred_quality" key.

```
>>> from Bio import SeqIO
>>> with open("Ace/consed_sample.ace") as handle:
...     for record in SeqIO.parse(handle, "ace"):
...         print("%s %s... %i" % (record.id, record.seq[:10], len(record)))
...         print(max(record.letter_annotations["phred_quality"]))
Contig1 agccccgggc... 1475
90
```

However, ACE files do not include a base quality for any gaps in the consensus sequence, and these are represented in Biopython with a quality of zero. Using zero is perhaps misleading as there may be very strong evidence to support the gap in the consensus. Previous versions of Biopython therefore used None instead, but this complicated usage, and prevented output of the gapped sequence as FASTQ format.

```
>>> from Bio import SeqIO
>>> with open("Ace/contig1.ace") as handle:
...     for record in SeqIO.parse(handle, "ace"):
...         print("%s ...%s..." % (record.id, record.seq[85:95]))
...         print(record.letter_annotations["phred_quality"][85:95])
...         print(max(record.letter_annotations["phred_quality"]))
Contig1 ...AGAGG-ATGC...
[57, 57, 54, 57, 57, 0, 57, 72, 72, 72]
90
Contig2 ...GAATTACTAT...
[68, 68, 68, 68, 68, 68, 68, 68, 68, 68]
90
```

Bio.SeqIO.FastaIO module

Bio.SeqIO support for the “fasta” (aka FastA or Pearson) file format.

You are expected to use this module via the Bio.SeqIO functions.

`Bio.SeqIO.FastaIO.`**`SimpleFastaParser`**(*handle*)

Iterate over Fasta records as string tuples.

Arguments:

- handle - input stream opened in text mode

For each record a tuple of two strings is returned, the FASTA title line (without the leading ‘>’ character), and the sequence (with any whitespace removed). The title line is not divided up into an identifier (the first word) and comment or description.

```
>>> with open("Fasta/dups.fasta") as handle:
...     for values in SimpleFastaParser(handle):
...         print(values)
...
('alpha', 'ACGTA')
('beta', 'CGTC')
('gamma', 'CCGCC')
('alpha (again - this is a duplicate entry to test the indexing code)', 'ACGTA')
('delta', 'CGCGC')
```

Bio.SeqIO.FastaIO.**FastaTwoLineParser**(*handle*)

Iterate over no-wrapping Fasta records as string tuples.

Arguments:

- handle - input stream opened in text mode

Functionally the same as SimpleFastaParser but with a strict interpretation of the FASTA format as exactly two lines per record, the greater-than-sign identifier with description, and the sequence with no line wrapping.

Any line wrapping will raise an exception, as will excess blank lines (other than the special case of a zero-length sequence as the second line of a record).

Examples

This file uses two lines per FASTA record:

```
>>> with open("Fasta/aster_no_wrap.pro") as handle:
...     for title, seq in FastaTwoLineParser(handle):
...         print("%s = %s..." % (title, seq[:3]))
...
gi|3298468|dbj|BAA31520.1| SAMIPF = GGH...
```

This equivalent file uses line wrapping:

```
>>> with open("Fasta/aster.pro") as handle:
...     for title, seq in FastaTwoLineParser(handle):
...         print("%s = %s..." % (title, seq[:3]))
...
Traceback (most recent call last):
  ...
ValueError: Expected FASTA record starting with '>' character. Perhaps this file is␣
↪using FASTA line wrapping? Got: 'MTFGLVYTVYATAIDPKKGSLGTIAPIAIGFIVGANI'
```

class Bio.SeqIO.FastaIO.**FastaIterator**(*source: IO[str] | PathLike | str | bytes*, *alphabet: None = None*)

Bases: *SequenceIterator*

Parser for Fasta files.

__init__(*source: IO[str] | PathLike | str | bytes*, *alphabet: None = None*) → None

Iterate over Fasta records as SeqRecord objects.

Arguments:

- source - input stream opened in text mode, or a path to a file
- alphabet - optional alphabet, not used. Leave as None.

By default this will act like calling Bio.SeqIO.parse(handle, "fasta") with no custom handling of the title lines:

```
>>> with open("Fasta/dups.fasta") as handle:
...     for record in FastaIterator(handle):
...         print(record.id)
...
alpha
beta
```

(continues on next page)

(continued from previous page)

```
gamma
alpha
delta
```

If you want to modify the records before writing, for example to change the ID of each record, you can use a generator function as follows:

```
>>> def modify_records(records):
...     for record in records:
...         record.id = record.id.upper()
...         yield record
...
>>> with open('Fasta/dups.fasta') as handle:
...     for record in modify_records(FastaIterator(handle)):
...         print(record.id)
...
ALPHA
BETA
GAMMA
ALPHA
DELTA
```

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

iterate(*handle*)

Parse the file and generate SeqRecord objects.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.FastaIO.**FastaTwoLineIterator**(*source*)

Bases: *SequenceIterator*

Parser for Fasta files with exactly two lines per record.

__init__(*source*)

Iterate over two-line Fasta records (as SeqRecord objects).

Arguments:

- source - input stream opened in text mode, or a path to a file

This uses a strict interpretation of the FASTA as requiring exactly two lines per record (no line wrapping).

Only the default title to ID/name/description parsing offered by the relaxed FASTA parser is offered.

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

iterate(*handle*)

Parse the file and generate SeqRecord objects.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.FastaIO.**FastaWriter**(*target*, *wrap=60*, *record2title=None*)

Bases: *SequenceWriter*

Class to write Fasta format files (OBSOLETE).

Please use the `as_fasta` function instead, or the top level `Bio.SeqIO.write()` function instead using `format="fasta"`.

__init__(*target*, *wrap=60*, *record2title=None*)

Create a Fasta writer (OBSOLETE).

Arguments:

- target - Output stream opened in text mode, or a path to a file.
- wrap - Optional line length used to wrap sequence lines. Defaults to wrapping the sequence at 60 characters Use zero (or None) for no wrapping, giving a single long line for the sequence.
- record2title - Optional function to return the text to be used for the title line of each record. By default a combination of the record.id and record.description is used. If the record.description starts with the record.id, then just the record.description is used.

You can either use:

```
handle = open(filename, "w")
writer = FastaWriter(handle)
writer.write_file(myRecords)
handle.close()
```

Or, follow the sequential file writer system, for example:

```
handle = open(filename, "w")
writer = FastaWriter(handle)
writer.write_header() # does nothing for Fasta files
...
Multiple writer.write_record() and/or writer.write_records() calls
...
writer.write_footer() # does nothing for Fasta files
handle.close()
```

write_record(*record*)

Write a single Fasta record to the file.

class Bio.SeqIO.FastaIO.**FastaTwoLineWriter**(*handle*, *record2title=None*)

Bases: *FastaWriter*

Class to write 2-line per record Fasta format files (OBSOLETE).

This means we write the sequence information without line wrapping, and will always write a blank line for an empty sequence.

Please use the `as_fasta_2line` function instead, or the top level `Bio.SeqIO.write()` function instead using `format="fasta"`.

__init__(*handle*, *record2title=None*)

Create a 2-line per record Fasta writer (OBSOLETE).

Arguments:

- handle - Handle to an output file, e.g. as returned by open(filename, "w")
- record2title - Optional function to return the text to be used for the title line of each record. By default a combination of the record.id and record.description is used. If the record.description starts with the record.id, then just the record.description is used.

You can either use:

```
handle = open(filename, "w")
writer = FastaWriter(handle)
writer.write_file(myRecords)
handle.close()
```

Or, follow the sequential file writer system, for example:

```
handle = open(filename, "w")
writer = FastaWriter(handle)
writer.write_header() # does nothing for Fasta files
...
Multiple writer.write_record() and/or writer.write_records() calls
...
writer.write_footer() # does nothing for Fasta files
handle.close()
```

__annotations__ = {}

Bio.SeqIO.FastaIO.**as_fasta**(*record*)

Turn a SeqRecord into a FASTA formatted string.

This is used internally by the SeqRecord's .format("fasta") method and by the SeqIO.write(..., ..., "fasta") function.

Bio.SeqIO.FastaIO.**as_fasta_2line**(*record*)

Turn a SeqRecord into a two-line FASTA formatted string.

This is used internally by the SeqRecord's .format("fasta-2line") method and by the SeqIO.write(..., ..., "fasta-2line") function.

Bio.SeqIO.GckIO module

Bio.SeqIO support for the "gck" file format.

The GCK binary format is generated by the Gene Construction Kit software from Textco BioSoftware, Inc.

class Bio.SeqIO.GckIO.**GckIterator**(*source*)

Bases: *SequenceIterator*

Parser for GCK files.

__init__(*source*)

Break up a GCK file into SeqRecord objects.

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

Note that a GCK file can only contain one sequence, so this iterator will always return a single record.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

Bio.SeqIO.GfaIO module

Bio.SeqIO support for the Graphical Fragment Assembly format.

This format is output by many assemblers and includes linkage information for how the different sequences fit together, however, we just care about the segment (sequence) information.

Documentation: - Version 1.x: https://gfa-spec.github.io/GFA-spec/GFA1.html - Version 2.0: https://gfa-spec.github.io/GFA-spec/GFA2.html

Bio.SeqIO.GfaIO.**Gfa1Iterator**(*source*)

Parser for GFA 1.x files.

Documentation: https://gfa-spec.github.io/GFA-spec/GFA1.html

Bio.SeqIO.GfaIO.**Gfa2Iterator**(*source*)

Parser for GFA 2.0 files.

Documentation for version 2: https://gfa-spec.github.io/GFA-spec/GFA2.html

Bio.SeqIO.IgIO module

Bio.SeqIO support for the "ig" (IntelliGenetics or MASE) file format.

This module is for reading and writing IntelliGenetics format files as SeqRecord objects. This file format appears to be the same as the MASE multiple sequence alignment format.

You are expected to use this module via the Bio.SeqIO functions.

class Bio.SeqIO.IgIO.**IgIterator**(*source*)

Bases: *SequenceIterator*

Parser for IntelliGenetics files.

__init__(*source*)

Iterate over IntelliGenetics records (as SeqRecord objects).

source - file-like object opened in text mode, or a path to a file

The optional free format file header lines (which start with two semi-colons) are ignored.

The free format commentary lines at the start of each record (which start with a semi-colon) are recorded as a single string with embedded new line characters in the SeqRecord's annotations dictionary under the key 'comment'.

Examples

```
>>> with open("IntelliGenetics/TAT_mase_nuc.txt") as handle:
...     for record in IgIterator(handle):
...         print("%s length %i" % (record.id, len(record)))
...
A_U455 length 303
B_HXB2R length 306
C_UG268A length 267
D_ELI length 309
F_BZ163A length 309
O_ANT70 length 342
O_MVP5180 length 348
CPZGAB length 309
CPZANT length 309
A_ROD length 390
B_EHOA length 420
D_MM251 length 390
STM_STM length 387
VER_AGM3 length 354
GRI_AGM677 length 264
SAB_SAB1C length 219
SYK_SYK length 330
```

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

iterate(*handle*)

Iterate over the records in the IntelliGenetics file.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

Bio.SeqIO.InsdcIO module

Bio.SeqIO support for the "genbank" and "embl" file formats.

You are expected to use this module via the Bio.SeqIO functions. Note that internally this module calls Bio.GenBank to do the actual parsing of GenBank, EMBL and IMGT files.

See Also: International Nucleotide Sequence Database Collaboration http://www.insdc.org/

GenBank http://www.ncbi.nlm.nih.gov/Genbank/

EMBL Nucleotide Sequence Database http://www.ebi.ac.uk/embl/

DDBJ (DNA Data Bank of Japan) http://www.ddbj.nig.ac.jp/

IMGT (use a variant of EMBL format with longer feature indents) http://imgt.cines.fr/download/LIGM-DB/userman_doc.html http://imgt.cines.fr/download/LIGM-DB/ftable_doc.html http://www.ebi.ac.uk/imgt/hla/docs/manual.html

class Bio.SeqIO.InsdcIO.**GenBankIterator**(*source*)

Bases: *SequenceIterator*

Parser for GenBank files.

__init__(*source*)

Break up a Genbank file into SeqRecord objects.

Argument source is a file-like object opened in text mode or a path to a file. Every section from the LOCUS line to the terminating // becomes a single SeqRecord with associated annotation and features.

Note that for genomes or chromosomes, there is typically only one record.

This gets called internally by Bio.SeqIO for the GenBank file format:

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("GenBank/cor6_6.gb", "gb"):
...     print(record.id)
...
X55053.1
X62281.1
M81224.1
AJ237582.1
L31939.1
AF297471.1
```

Equivalently,

```
>>> with open("GenBank/cor6_6.gb") as handle:
...     for record in GenBankIterator(handle):
...         print(record.id)
...
X55053.1
X62281.1
M81224.1
AJ237582.1
L31939.1
AF297471.1
```

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.InsdcIO.**EmblIterator**(*source*)

Bases: *SequenceIterator*

Parser for EMBL files.

__init__(*source*)

Break up an EMBL file into SeqRecord objects.

Argument source is a file-like object opened in text mode or a path to a file. Every section from the LOCUS line to the terminating // becomes a single SeqRecord with associated annotation and features.

Note that for genomes or chromosomes, there is typically only one record.

This gets called internally by Bio.SeqIO for the EMBL file format:

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("EMBL/epo_prt_selection.embl", "embl"):
...     print(record.id)
...
A00022.1
A00028.1
A00031.1
A00034.1
A00060.1
A00071.1
A00072.1
A00078.1
CQ797900.1
```

Equivalently,

```
>>> with open("EMBL/epo_prt_selection.embl") as handle:
...     for record in EmblIterator(handle):
...         print(record.id)
...
A00022.1
A00028.1
A00031.1
A00034.1
A00060.1
A00071.1
A00072.1
A00078.1
CQ797900.1
```

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.InsdcIO.**ImgtIterator**(*source*)

Bases: *SequenceIterator*

Parser for IMGT files.

__init__(*source*)

Break up an IMGT file into SeqRecord objects.

Argument source is a file-like object opened in text mode or a path to a file. Every section from the LOCUS line to the terminating // becomes a single SeqRecord with associated annotation and features.

Note that for genomes or chromosomes, there is typically only one record.

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.InsdcIO.**GenBankCdsFeatureIterator**(*source*)

Bases: *SequenceIterator*

Parser for GenBank files, creating a SeqRecord for each CDS feature.

__init__(*source*)

Break up a Genbank file into SeqRecord objects for each CDS feature.

Argument source is a file-like object opened in text mode or a path to a file.

Every section from the LOCUS line to the terminating // can contain many CDS features. These are returned as with the stated amino acid translation sequence (if given).

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.InsdcIO.**EmblCdsFeatureIterator**(*source*)

Bases: *SequenceIterator*

Parser for EMBL files, creating a SeqRecord for each CDS feature.

__init__(*source*)

Break up a EMBL file into SeqRecord objects for each CDS feature.

Argument source is a file-like object opened in text mode or a path to a file.

Every section from the LOCUS line to the terminating // can contain many CDS features. These are returned as with the stated amino acid translation sequence (if given).

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.InsdcIO.**GenBankWriter**(*target: IO | PathLike | str | bytes*, *mode: str = 'w'*)

Bases: _InsdcWriter

GenBank writer.

HEADER_WIDTH = 12

QUALIFIER_INDENT = 21

STRUCTURED_COMMENT_START = '-START##'

STRUCTURED_COMMENT_END = '-END##'

STRUCTURED_COMMENT_DELIM = ' :: '

LETTERS_PER_LINE = 60

SEQUENCE_INDENT = 9

write_record(*record*)

Write a single record to the output file.

__annotations__ = {}

class Bio.SeqIO.InsdcIO.EmblWriter(*target: IO | PathLike | str | bytes*, *mode: str = 'w'*)

Bases: _InsdcWriter

EMBL writer.

HEADER_WIDTH = 5

QUALIFIER_INDENT = 21

QUALIFIER_INDENT_STR = 'FT '

QUALIFIER_INDENT_TMP = 'FT %s '

FEATURE_HEADER = 'FH Key Location/Qualifiers\nFH\n'

LETTERS_PER_BLOCK = 10

BLOCKS_PER_LINE = 6

LETTERS_PER_LINE = 60

POSITION_PADDING = 10

write_record(*record*)

Write a single record to the output file.

__annotations__ = {}

class Bio.SeqIO.InsdcIO.ImgtWriter(*target: IO | PathLike | str | bytes*, *mode: str = 'w'*)

Bases: *EmblWriter*

IMGT writer (EMBL format variant).

HEADER_WIDTH = 5

QUALIFIER_INDENT = 25

QUALIFIER_INDENT_STR = 'FT '

QUALIFIER_INDENT_TMP = 'FT %s '

FEATURE_HEADER = 'FH Key Location/Qualifiers\nFH\n'

__annotations__ = {}

Bio.SeqIO.Interfaces module

Bio.SeqIO support module (not for general use).

Unless you are writing a new parser or writer for Bio.SeqIO, you should not use this module. It provides base classes to try and simplify things.

class Bio.SeqIO.Interfaces.**SequenceIterator**(*source: IO | PathLike | str | bytes*, *alphabet: None = None*, *mode: str = 't'*, *fmt: str | None = None*)

Bases: ABC, Generic

Base class for building SeqRecord iterators.

You should write a parse method that returns a SeqRecord generator. You may wish to redefine the __init__ method as well.

__init__(*source: IO | PathLike | str | bytes*, *alphabet: None = None*, *mode: str = 't'*, *fmt: str | None = None*) → None

Create a SequenceIterator object.

Arguments: - source - input file stream, or path to input file - alphabet - no longer used, should be None

This method MAY be overridden by any subclass.

Note when subclassing: - there should be a single non-optional argument, the source. - you do not have to require an alphabet. - you can add additional optional arguments.

__next__()

Return the next entry.

__iter__()

Iterate over the entries as a SeqRecord objects.

Example usage for Fasta files:

```
with open("example.fasta","r") as myFile:
    myFastaReader = FastaIterator(myFile)
    for record in myFastaReader:
        print(record.id)
        print(record.seq)
```

This method SHOULD NOT be overridden by any subclass. It should be left as is, which will call the subclass implementation of __next__ to actually parse the file.

abstract parse(*handle: IO*) → Iterator[*SeqRecord*]

Start parsing the file, and return a SeqRecord iterator.

__abstractmethods__ = frozenset({'parse'})

__annotations__ = {}

__orig_bases__ = (<class 'abc.ABC'>, typing.Generic[~AnyStr])

__parameters__ = (~AnyStr,)

class Bio.SeqIO.Interfaces.**SequenceWriter**(*target: IO | PathLike | str | bytes*, *mode: str = 'w'*)

Bases: object

Base class for sequence writers. This class should be subclassed.

It is intended for sequential file formats with an (optional) header, repeated records, and an (optional) footer, as well as for interlaced file formats such as Clustal.

The user may call the write_file() method to write a complete file containing the sequences.

Alternatively, users may call the write_header(), followed by multiple calls to write_record() and/or write_records(), followed finally by write_footer().

Note that write_header() cannot require any assumptions about the number of records.

__init__(*target: IO | PathLike | str | bytes*, *mode: str = 'w'*) → None

Create the writer object.

clean(*text: str*) → str

Use this to avoid getting newlines in the output.

write_header()

Write the file header to the output file.

write_footer()

Write the file footer to the output file.

write_record(*record*)

Write a single record to the output file.

record - a SeqRecord object

__annotations__ = {}

write_records(*records*, *maxcount=None*)

Write records to the output file, and return the number of records.

records - A list or iterator returning SeqRecord objects maxcount - The maximum number of records allowed by the file format, or None if there is no maximum.

write_file(*records*, *mincount=0*, *maxcount=None*)

Write a complete file with the records, and return the number of records.

records - A list or iterator returning SeqRecord objects

Bio.SeqIO.NibIO module

Bio.SeqIO support for the UCSC nib file format.

Nib stands for nibble (4 bit) representation of nucleotide sequences. The two nibbles in a byte each store one nucleotide, represented numerically as follows:

- 0 - T
- 1 - C
- 2 - A
- 3 - G
- 4 - N (unknown)

As the first bit in a nibble is set if the nucleotide is soft-masked, we additionally have:

- 8 - t
- 9 - c

- a - a
- b - g
- c - n (unknown)

A nib file contains only one sequence record. You are expected to use this module via the Bio.SeqIO functions under the format name "nib":

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("Nib/test_even_bigendian.nib", "nib")
>>> print("%i %s..." % (len(record), record.seq[:20]))
50 nAGAAGagccgcNGgCActt...
```

For detailed information on the file format, please see the UCSC description at https://genome.ucsc.edu/FAQ/FAQformat.html.

class Bio.SeqIO.NibIO.**NibIterator**(*source*)

Bases: *SequenceIterator*

Parser for nib files.

__init__(*source*)

Iterate over a nib file and yield a SeqRecord.

- source - a file-like object or a path to a file in the nib file format as defined by UCSC; the file must be opened in binary mode.

Note that a nib file always contains only one sequence record. The sequence of the resulting SeqRecord object should match the sequence generated by Jim Kent's nibFrag utility run with the -masked option.

This function is used internally via the Bio.SeqIO functions:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("Nib/test_even_bigendian.nib", "nib")
>>> print("%s %i" % (record.seq, len(record)))
nAGAAGagccgcNGgCActtGAnTAtCGTCgcCacCaGncGncTtGNtGG 50
```

You can also call it directly:

```
>>> with open("Nib/test_even_bigendian.nib", "rb") as handle:
...     for record in NibIterator(handle):
...         print("%s %i" % (record.seq, len(record)))
...
nAGAAGagccgcNGgCActtGAnTAtCGTCgcCacCaGncGncTtGNtGG 50
```

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

iterate(*handle*, *byteorder*)

Iterate over the records in the nib file.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.NibIO.**NibWriter**(*target*)

Bases: *SequenceWriter*

Nib file writer.

__init__(*target*)

Initialize a Nib writer object.

Arguments:

- target - output stream opened in binary mode, or a path to a file

write_header()

Write the file header.

write_record(*record*)

Write a single record to the output file.

write_file(*records*)

Write the complete file with the records, and return the number of records.

__annotations__ = {}

Bio.SeqIO.PdbIO module

Bio.SeqIO support for accessing sequences in PDB and mmCIF files.

Bio.SeqIO.PdbIO.**AtomIterator**(*pdb_id*, *structure*)

Return SeqRecords from Structure objects.

Base function for sequence parsers that read structures Bio.PDB parsers.

Once a parser from Bio.PDB has been used to load a structure into a Bio.PDB.Structure.Structure object, there is no difference in how the sequence parser interprets the residue sequence. The functions in this module may be used by SeqIO modules wishing to parse sequences from lists of residues.

Calling functions must pass a Bio.PDB.Structure.Structure object.

See Bio.SeqIO.PdbIO.PdbAtomIterator and Bio.SeqIO.PdbIO.CifAtomIterator for details.

class Bio.SeqIO.PdbIO.**PdbSeqresIterator**(*source*)

Bases: *SequenceIterator*

Parser for PDB files.

__init__(*source*)

Return SeqRecord objects for each chain in a PDB file.

Arguments:

- source - input stream opened in text mode, or a path to a file

The sequences are derived from the SEQRES lines in the PDB file header, not the atoms of the 3D structure.

Specifically, these PDB records are handled: DBREF, DBREF1, DBREF2, SEQADV, SEQRES, MODRES

See: http://www.wwpdb.org/documentation/format23/sect3.html

This gets called internally via Bio.SeqIO for the SEQRES based interpretation of the PDB file format:

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("PDB/1A8O.pdb", "pdb-seqres"):
...     print("Record id %s, chain %s" % (record.id, record.annotations["chain
↪"]))
...     print(record.dbxrefs)
...
Record id 1A8O:A, chain A
['UNP:P12497', 'UNP:POL_HV1N5']
```

Equivalently,

```
>>> with open("PDB/1A8O.pdb") as handle:
...     for record in PdbSeqresIterator(handle):
...         print("Record id %s, chain %s" % (record.id, record.annotations[
↪"chain"]))
...         print(record.dbxrefs)
...
Record id 1A8O:A, chain A
['UNP:P12497', 'UNP:POL_HV1N5']
```

Note the chain is recorded in the annotations dictionary, and any PDB DBREF lines are recorded in the database cross-references list.

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

iterate(*handle*)

Iterate over the records in the PDB file.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

Bio.SeqIO.PdbIO.**PdbAtomIterator**(*source*)

Return SeqRecord objects for each chain in a PDB file.

Argument source is a file-like object or a path to a file.

The sequences are derived from the 3D structure (ATOM records), not the SEQRES lines in the PDB file header.

Unrecognised three letter amino acid codes (e.g. "CSD") from HETATM entries are converted to "X" in the sequence.

In addition to information from the PDB header (which is the same for all records), the following chain specific information is placed in the annotation:

record.annotations["residues"] = List of residue ID strings record.annotations["chain"] = Chain ID (typically A, B ,...) record.annotations["model"] = Model ID (typically zero)

Where amino acids are missing from the structure, as indicated by residue numbering, the sequence is filled in with 'X' characters to match the size of the missing region, and None is included as the corresponding entry in the list record.annotations["residues"].

This function uses the Bio.PDB module to do most of the hard work. The annotation information could be improved but this extra parsing should be done in parse_pdb_header, not this module.

This gets called internally via Bio.SeqIO for the atom based interpretation of the PDB file format:

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("PDB/1A8O.pdb", "pdb-atom"):
...     print("Record id %s, chain %s" % (record.id, record.annotations["chain"]))
...
Record id 1A8O:A, chain A
```

Equivalently,

```
>>> with open("PDB/1A8O.pdb") as handle:
...     for record in PdbAtomIterator(handle):
...         print("Record id %s, chain %s" % (record.id, record.annotations["chain
↪"]))
...
Record id 1A8O:A, chain A
```

Bio.SeqIO.PdbIO.**CifSeqresIterator**(*source*)

Return SeqRecord objects for each chain in an mmCIF file.

Argument source is a file-like object or a path to a file.

The sequences are derived from the _entity_poly_seq entries in the mmCIF file, not the atoms of the 3D structure.

Specifically, these mmCIF records are handled: _pdbx_poly_seq_scheme and _struct_ref_seq. The _pdbx_poly_seq records contain sequence information, and the _struct_ref_seq records contain database cross-references.

See: http://mmcif.wwpdb.org/dictionaries/mmcif_pdbx_v40.dic/Categories/pdbx_poly_seq_scheme.html and http://mmcif.wwpdb.org/dictionaries/mmcif_pdbx_v50.dic/Categories/struct_ref_seq.html

This gets called internally via Bio.SeqIO for the sequence-based interpretation of the mmCIF file format:

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("PDB/1A8O.cif", "cif-seqres"):
...     print("Record id %s, chain %s" % (record.id, record.annotations["chain"]))
...     print(record.dbxrefs)
...
Record id 1A8O:A, chain A
['UNP:P12497', 'UNP:POL_HV1N5']
```

Equivalently,

```
>>> with open("PDB/1A8O.cif") as handle:
...     for record in CifSeqresIterator(handle):
...         print("Record id %s, chain %s" % (record.id, record.annotations["chain
↪"]))
...         print(record.dbxrefs)
...
Record id 1A8O:A, chain A
['UNP:P12497', 'UNP:POL_HV1N5']
```

Note the chain is recorded in the annotations dictionary, and any mmCIF _struct_ref_seq entries are recorded in the database cross-references list.

Bio.SeqIO.PdbIO.**CifAtomIterator**(*source*)

Return SeqRecord objects for each chain in an mmCIF file.

Argument source is a file-like object or a path to a file.

The sequences are derived from the 3D structure (_atom_site.* fields) in the mmCIF file.

Unrecognised three letter amino acid codes (e.g. "CSD") from HETATM entries are converted to "X" in the sequence.

In addition to information from the PDB header (which is the same for all records), the following chain specific information is placed in the annotation:

record.annotations["residues"] = List of residue ID strings record.annotations["chain"] = Chain ID (typically A, B ,...) record.annotations["model"] = Model ID (typically zero)

Where amino acids are missing from the structure, as indicated by residue numbering, the sequence is filled in with 'X' characters to match the size of the missing region, and None is included as the corresponding entry in the list record.annotations["residues"].

This function uses the Bio.PDB module to do most of the hard work. The annotation information could be improved but this extra parsing should be done in parse_pdb_header, not this module.

This gets called internally via Bio.SeqIO for the atom based interpretation of the PDB file format:

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("PDB/1A8O.cif", "cif-atom"):
...     print("Record id %s, chain %s" % (record.id, record.annotations["chain"]))
...
Record id 1A8O:A, chain A
```

Equivalently,

```
>>> with open("PDB/1A8O.cif") as handle:
...     for record in CifAtomIterator(handle):
...         print("Record id %s, chain %s" % (record.id, record.annotations["chain
↪"]))
...
Record id 1A8O:A, chain A
```

Bio.SeqIO.PhdIO module

Bio.SeqIO support for the "phd" file format.

PHD files are output by PHRED and used by PHRAP and CONSED.

You are expected to use this module via the Bio.SeqIO functions, under the format name "phd". See also the underlying Bio.Sequencing.Phd module.

For example, using Bio.SeqIO we can read in one of the example PHRED files from the Biopython unit tests:

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("Phd/phd1", "phd"):
...     print(record.id)
...     print("%s..." % record.seq[:10])
...     print("%s..." % record.letter_annotations["phred_quality"][:10])
34_222_(80-A03-19).b.ab1
ctccgtcgga...
[9, 9, 10, 19, 22, 37, 28, 28, 24, 22]...
425_103_(81-A03-19).g.ab1
cgggatccca...
[14, 17, 22, 10, 10, 10, 15, 8, 8, 9]...
```

(continues on next page)

(continued from previous page)

```
425_7_(71-A03-19).b.ab1
acataaatca...
[10, 10, 10, 10, 8, 8, 6, 6, 6, 6]...
```

Since PHRED files contain quality scores, you can save them as FASTQ or as QUAL files, for example using Bio.SeqIO.write(...), or simply with the format method of the SeqRecord object:

```
>>> print(record[:50].format("fastq"))
@425_7_(71-A03-19).b.ab1
acataaatcaaattactnaccaacacacaaaccngtctcgcgtagtggag
+
++++))''''')(''')$!$''')''''(+.''$!$))))+)))'''''''
```

Or,

```
>>> print(record[:50].format("qual"))
>425_7_(71-A03-19).b.ab1
10 10 10 10 8 8 6 6 6 6 8 7 6 6 6 8 3 0 3 6 6 6 8 6 6 6 6 7
10 13 6 6 3 0 3 8 8 8 8 10 8 8 8 6 6 6 6 6 6 6
```

Note these examples only show the first 50 bases to keep the output short.

`Bio.SeqIO.PhdIO.`**`PhdIterator`**(*source: IO[str] | PathLike | str | bytes*) → Iterator[*SeqRecord*]

Return SeqRecord objects from a PHD file.

Arguments:

- source - input stream opened in text mode, or a path to a file

This uses the Bio.Sequencing.Phd module to do the hard work.

class `Bio.SeqIO.PhdIO.`**`PhdWriter`**(*handle: IO | PathLike | str | bytes*)

Bases: *SequenceWriter*

Class to write Phd format files.

`__init__`(*handle: IO | PathLike | str | bytes*) → None

Initialize the class.

`write_record`(*record*)

Write a single Phd record to the file.

`__annotations__ = {}`

Bio.SeqIO.PirIO module

Bio.SeqIO support for the "pir" (aka PIR or NBRF) file format.

This module is for reading and writing PIR or NBRF format files as SeqRecord objects.

You are expected to use this module via the Bio.SeqIO functions, or if the file contains a sequence alignment, optionally via Bio.AlignIO instead.

This format was introduced for the Protein Information Resource (PIR), a project of the National Biomedical Research Foundation (NBRF). The PIR database itself is now part of UniProt.

The file format is described online at: http://www.ebi.ac.uk/help/pir_frame.html http://www.cmbi.kun.nl/bioinf/tools/crab_pir.html (currently down)

An example file in this format would be:

```
>P1;CRAB_ANAPL
ALPHA CRYSTALLIN B CHAIN (ALPHA(B)-CRYSTALLIN).
  MDITIHNPLI RRPLFSWLAP SRIFDQIFGE HLQESELLPA SPSLSPFLMR
  SPIFRMPSWL ETGLSEMRLE KDKFSVNLDV KHFSPEELKV KVLGDMVEIH
  GKHEERQDEH GFIAREFNRK YRIPADVDPL TITSSLSLDG VLTVSAPRKQ
  SDVPERSIPI TREEKPAIAG AQRK*

>P1;CRAB_BOVIN
ALPHA CRYSTALLIN B CHAIN (ALPHA(B)-CRYSTALLIN).
  MDIAIHHPWI RRPFFPFHSP SRLFDQFFGE HLLESDLFPA STSLSPFYLR
  PPSFLRAPSW IDTGLSEMRL EKDRFSVNLD VKHFSPEELK VKVLGDVIEV
  HGKHEERQDE HGFISREFHR KYRIPADVDP LAITSSLSSD GVLTVNGPRK
  QASGPERTIP ITREEKPAVT AAPKK*
```

Or, an example of a multiple sequence alignment:

```
>P1;S27231
rhodopsin - northern leopard frog
MNGTEGPNFY IPMSNKTGVV RSPFDYPQYY LAEPWKYSVL AAYMFLLILL GLPINFMTLY
VTIQHKKLRT PLNYILLNLG VCNHFMVLCG FTITMYTSLH GYFVFGQTGC YFEGFFATLG
GEIALWSLVV LAIERYIVVC KPMSNFRFGE NHAMMGVAFT WIMALACAVP PLFGWSRYIP
EGMQCSCGVD YYTLKPEVNN ESFVIYMFVV HFLIPLIIIS FCYGRLVCTV KEAAAQQQES
ATTQKAEKEV TRMVIIMVIF FLICWVPYAY VAFYIFTHQG SEFGPIFMTV PAFFAKSSAI
YNPVIYIMLN KQFRNCMITT LCCGKNPFGD DDASSAATSK TEATSVSTSQ VSPA*

>P1;I51200
rhodopsin - African clawed frog
MNGTEGPNFY VPMSNKTGVV RSPFDYPQYY LAEPWQYSAL AAYMFLLILL GLPINFMTLF
VTIQHKKLRT PLNYILLNLV FANHFMVLCG FTVTMYTSMH GYFIFGPTGC YIEGFFATLG
GEVALWSLVV LAVERYIVVC KPMANFRFGE NHAIMGVAFT WIMALSCAAP PLFGWSRYIP
EGMQCSCGVD YYTLKPEVNN ESFVIYMFIV HFTIPLIVIF FCYGRLLCTV KEAAAQQQES
LTTQKAEKEV TRMVVIMVVF FLICWVPYAY VAFYIFTHQG SNFGPVFMTV PAFFAKSSAI
YNPVIYIVLN KQFRNCLITT LCCGKNPFGD EDGSSAATSK TEASSVSSSQ VSPA*

>P1;JN0120
rhodopsin - Japanese lamprey
MNGTEGDNFY VPFSNKTGLA RSPYEYPQYY LAEPWKYSAL AAYMFFLILV GFPVNFLTLF
VTVQHKKLRT PLNYILLNLA MANLFMVLFG FTVTMYTSMN GYFVFGPTMC SIEGFFATLG
GEVALWSLVV LAIERYIVIC KPMGNFRFGN THAIMGVAFT WIMALACAAP PLVGWSRYIP
EGMQCSCGPD YYTLNPNFNN ESYVVYMFVV HFLVPFVIIF FCYGRLLCTV KEAAAAQQES
ASTQKAEKEV TRMVVLMVIG FLVCWVPYAS VAFYIFTHQG SDFGATFMTL PAFFAKSSAL
YNPVIYILMN KQFRNCMITT LCCGKNPLGD DE-SGASTSKT EVSSVSTSPV SPA*
```

As with the FASTA format, each record starts with a line beginning with ">" character. There is then a two letter sequence type (P1, F1, DL, DC, RL, RC, or XX), a semi colon, and the identification code. The second like is free text description. The remaining lines contain the sequence itself, terminating in an asterisk. Space separated blocks of ten letters as shown above are typical.

Sequence codes and their meanings:

- P1 - Protein (complete)
- F1 - Protein (fragment)

- D1 - DNA (e.g. EMBOSS seqret output)
- DL - DNA (linear)
- DC - DNA (circular)
- RL - RNA (linear)
- RC - RNA (circular)
- N3 - tRNA
- N1 - Other functional RNA
- XX - Unknown

class Bio.SeqIO.PirIO.**PirIterator**(*source*)

Bases: *SequenceIterator*

Parser for PIR files.

__init__(*source*)

Iterate over a PIR file and yield SeqRecord objects.

source - file-like object or a path to a file.

Examples

```
>>> with open("NBRF/DMB_prot.pir") as handle:
...     for record in PirIterator(handle):
...         print("%s length %i" % (record.id, len(record)))
HLA:HLA00489 length 263
HLA:HLA00490 length 94
HLA:HLA00491 length 94
HLA:HLA00492 length 80
HLA:HLA00493 length 175
HLA:HLA01083 length 188
```

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

iterate(*handle*)

Iterate over the records in the PIR file.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.PirIO.**PirWriter**(*handle*, *wrap=60*, *record2title=None*, *code=None*)

Bases: *SequenceWriter*

Class to write PIR format files.

__init__(*handle*, *wrap=60*, *record2title=None*, *code=None*)

Create a PIR writer.

Arguments:

- handle - Handle to an output file, e.g. as returned by open(filename, "w")

- wrap - Optional line length used to wrap sequence lines. Defaults to wrapping the sequence at 60 characters Use zero (or None) for no wrapping, giving a single long line for the sequence.
- record2title - Optional function to return the text to be used for the title line of each record. By default a combination of the record.id, record.name and record.description is used.
- code - Optional sequence code must be one of P1, F1, D1, DL, DC, RL, RC, N3 and XX. By default None is used, which means auto detection based on the molecule type in the record annotation.

You can either use:

```
handle = open(filename, "w")
writer = PirWriter(handle)
writer.write_file(myRecords)
handle.close()
```

Or, follow the sequential file writer system, for example:

```
handle = open(filename, "w")
writer = PirWriter(handle)
writer.write_header() # does nothing for PIR files
...
Multiple writer.write_record() and/or writer.write_records() calls
...
writer.write_footer() # does nothing for PIR files
handle.close()
```

write_record(*record*)

Write a single PIR record to the file.

__annotations__ = {}

Bio.SeqIO.QualityIO module

Bio.SeqIO support for the FASTQ and QUAL file formats.

Note that you are expected to use this code via the Bio.SeqIO interface, as shown below.

The FASTQ file format is used frequently at the Wellcome Trust Sanger Institute to bundle a FASTA sequence and its PHRED quality data (integers between 0 and 90). Rather than using a single FASTQ file, often paired FASTA and QUAL files are used containing the sequence and the quality information separately.

The PHRED software reads DNA sequencing trace files, calls bases, and assigns a non-negative quality value to each called base using a logged transformation of the error probability, Q = -10 log10(Pe), for example:

```
Pe = 1.0,         Q =  0
Pe = 0.1,         Q = 10
Pe = 0.01,        Q = 20
...
Pe = 0.00000001,  Q = 80
Pe = 0.000000001, Q = 90
```

In typical raw sequence reads, the PHRED quality valuea will be from 0 to 40. In the QUAL format these quality values are held as space separated text in a FASTA like file format. In the FASTQ format, each quality values is encoded with a single ASCI character using chr(Q+33), meaning zero maps to the character "!" and for example 80 maps to "q". For the Sanger FASTQ standard the allowed range of PHRED scores is 0 to 93 inclusive. The sequences and quality are then stored in pairs in a FASTA like format.

Unfortunately there is no official document describing the FASTQ file format, and worse, several related but different variants exist. For more details, please read this open access publication:

```
The Sanger FASTQ file format for sequences with quality scores, and the
Solexa/Illumina FASTQ variants.
P.J.A.Cock (Biopython), C.J.Fields (BioPerl), N.Goto (BioRuby),
M.L.Heuer (BioJava) and P.M. Rice (EMBOSS).
Nucleic Acids Research 2010 38(6):1767-1771
https://doi.org/10.1093/nar/gkp1137
```

The good news is that Roche 454 sequencers can output files in the QUAL format, and sensibly they use PHREP style scores like Sanger. Converting a pair of FASTA and QUAL files into a Sanger style FASTQ file is easy. To extract QUAL files from a Roche 454 SFF binary file, use the Roche off instrument command line tool "sffinfo" with the -q or -qual argument. You can extract a matching FASTA file using the -s or -seq argument instead.

The bad news is that Solexa/Illumina did things differently - they have their own scoring system AND their own incompatible versions of the FASTQ format. Solexa/Illumina quality scores use Q = - 10 log10 (Pe / (1-Pe)), which can be negative. PHRED scores and Solexa scores are NOT interchangeable (but a reasonable mapping can be achieved between them, and they are approximately equal for higher quality reads).

Confusingly early Solexa pipelines produced a FASTQ like file but using their own score mapping and an ASCII offset of 64. To make things worse, for the Solexa/Illumina pipeline 1.3 onwards, they introduced a third variant of the FASTQ file format, this time using PHRED scores (which is more consistent) but with an ASCII offset of 64.

i.e. There are at least THREE different and INCOMPATIBLE variants of the FASTQ file format: The original Sanger PHRED standard, and two from Solexa/Illumina.

The good news is that as of CASAVA version 1.8, Illumina sequencers will produce FASTQ files using the standard Sanger encoding.

You are expected to use this module via the Bio.SeqIO functions, with the following format names:

- "qual" means simple quality files using PHRED scores (e.g. from Roche 454)
- "fastq" means Sanger style FASTQ files using PHRED scores and an ASCII offset of 33 (e.g. from the NCBI Short Read Archive and Illumina 1.8+). These can potentially hold PHRED scores from 0 to 93.
- "fastq-sanger" is an alias for "fastq".
- "fastq-solexa" means old Solexa (and also very early Illumina) style FASTQ files, using Solexa scores with an ASCII offset 64. These can hold Solexa scores from -5 to 62.
- "fastq-illumina" means newer Illumina 1.3 to 1.7 style FASTQ files, using PHRED scores but with an ASCII offset 64, allowing PHRED scores from 0 to 62.

We could potentially add support for "qual-solexa" meaning QUAL files which contain Solexa scores, but thus far there isn't any reason to use such files.

For example, consider the following short FASTQ file:

```
@EAS54_6_R1_2_1_413_324
CCCTTCTTGTCTTCAGCGTTTCTCC
+
;;3;;;;;;;;;;;;7;;;;;;;88
@EAS54_6_R1_2_1_540_792
TTGGCAGGCCAAGGCCGATGGATCA
+
;;;;;;;;;;;7;;;;;-;;;3;83
@EAS54_6_R1_2_1_443_348
GTTGCTTCTGGCGTGGGTGGGGGGG
```

(continues on next page)

(continued from previous page)

```
+
;;;;;;;;;;;9;7;;.7;393333
```

This contains three reads of length 25. From the read length these were probably originally from an early Solexa/Illumina sequencer but this file follows the Sanger FASTQ convention (PHRED style qualities with an ASCII offset of 33). This means we can parse this file using Bio.SeqIO using "fastq" as the format name:

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("Quality/example.fastq", "fastq"):
...     print("%s %s" % (record.id, record.seq))
EAS54_6_R1_2_1_413_324 CCCTTCTTGTCTTCAGCGTTTCTCC
EAS54_6_R1_2_1_540_792 TTGGCAGGCCAAGGCCGATGGATCA
EAS54_6_R1_2_1_443_348 GTTGCTTCTGGCGTGGGTGGGGGGG
```

The qualities are held as a list of integers in each record's annotation:

```
>>> print(record)
ID: EAS54_6_R1_2_1_443_348
Name: EAS54_6_R1_2_1_443_348
Description: EAS54_6_R1_2_1_443_348
Number of features: 0
Per letter annotation for: phred_quality
Seq('GTTGCTTCTGGCGTGGGTGGGGGGG')
>>> print(record.letter_annotations["phred_quality"])
[26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 24, 26, 22, 26, 26, 13, 22, 26, 18, 24, 18,␣
↪18, 18, 18]
```

You can use the SeqRecord format method to show this in the QUAL format:

```
>>> print(record.format("qual"))
>EAS54_6_R1_2_1_443_348
26 26 26 26 26 26 26 26 26 26 26 24 26 22 26 26 13 22 26 18
24 18 18 18 18
```

Or go back to the FASTQ format, use "fastq" (or "fastq-sanger"):

```
>>> print(record.format("fastq"))
@EAS54_6_R1_2_1_443_348
GTTGCTTCTGGCGTGGGTGGGGGGG
+
;;;;;;;;;;;9;7;;.7;393333
```

Or, using the Illumina 1.3+ FASTQ encoding (PHRED values with an ASCII offset of 64):

```
>>> print(record.format("fastq-illumina"))
@EAS54_6_R1_2_1_443_348
GTTGCTTCTGGCGTGGGTGGGGGGG
+
ZZZZZZZZZZZZXZVZZMVZRXRRRR
```

You can also get Biopython to convert the scores and show a Solexa style FASTQ file:

```
>>> print(record.format("fastq-solexa"))
@EAS54_6_R1_2_1_443_348
GTTGCTTCTGGCGTGGGTGGGGGGG
+
ZZZZZZZZZZZXZVZZMVZRXRRRR
```

Notice that this is actually the same output as above using "fastq-illumina" as the format! The reason for this is all these scores are high enough that the PHRED and Solexa scores are almost equal. The differences become apparent for poor quality reads. See the functions solexa_quality_from_phred and phred_quality_from_solexa for more details.

If you wanted to trim your sequences (perhaps to remove low quality regions, or to remove a primer sequence), try slicing the SeqRecord objects. e.g.

```
>>> sub_rec = record[5:15]
>>> print(sub_rec)
ID: EAS54_6_R1_2_1_443_348
Name: EAS54_6_R1_2_1_443_348
Description: EAS54_6_R1_2_1_443_348
Number of features: 0
Per letter annotation for: phred_quality
Seq('TTCTGGCGTG')
>>> print(sub_rec.letter_annotations["phred_quality"])
[26, 26, 26, 26, 26, 26, 24, 26, 22, 26]
>>> print(sub_rec.format("fastq"))
@EAS54_6_R1_2_1_443_348
TTCTGGCGTG
+
;;;;;;9;7;
```

If you wanted to, you could read in this FASTQ file, and save it as a QUAL file:

```
>>> from Bio import SeqIO
>>> record_iterator = SeqIO.parse("Quality/example.fastq", "fastq")
>>> with open("Quality/temp.qual", "w") as out_handle:
...     SeqIO.write(record_iterator, out_handle, "qual")
3
```

You can of course read in a QUAL file, such as the one we just created:

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("Quality/temp.qual", "qual"):
...     print("%s read of length %d" % (record.id, len(record.seq)))
EAS54_6_R1_2_1_413_324 read of length 25
EAS54_6_R1_2_1_540_792 read of length 25
EAS54_6_R1_2_1_443_348 read of length 25
```

Notice that QUAL files don't have a proper sequence present! But the quality information is there:

```
>>> print(record)
ID: EAS54_6_R1_2_1_443_348
Name: EAS54_6_R1_2_1_443_348
Description: EAS54_6_R1_2_1_443_348
Number of features: 0
Per letter annotation for: phred_quality
```

(continues on next page)

(continued from previous page)

```
Undefined sequence of length 25
>>> print(record.letter_annotations["phred_quality"])
[26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 24, 26, 22, 26, 26, 13, 22, 26, 18, 24, 18,␣
↪18, 18, 18]
```

Just to keep things tidy, if you are following this example yourself, you can delete this temporary file now:

```
>>> import os
>>> os.remove("Quality/temp.qual")
```

Sometimes you won't have a FASTQ file, but rather just a pair of FASTA and QUAL files. Because the Bio.SeqIO system is designed for reading single files, you would have to read the two in separately and then combine the data. However, since this is such a common thing to want to do, there is a helper iterator defined in this module that does this for you - PairedFastaQualIterator.

Alternatively, if you have enough RAM to hold all the records in memory at once, then a simple dictionary approach would work:

```
>>> from Bio import SeqIO
>>> reads = SeqIO.to_dict(SeqIO.parse("Quality/example.fasta", "fasta"))
>>> for rec in SeqIO.parse("Quality/example.qual", "qual"):
...     reads[rec.id].letter_annotations["phred_quality"]=rec.letter_annotations["phred_
↪quality"]
```

You can then access any record by its key, and get both the sequence and the quality scores.

```
>>> print(reads["EAS54_6_R1_2_1_540_792"].format("fastq"))
@EAS54_6_R1_2_1_540_792
TTGGCAGGCCAAGGCCGATGGATCA
+
;;;;;;;;;;;7;;;;;-;;;3;83
```

It is important that you explicitly tell Bio.SeqIO which FASTQ variant you are using ("fastq" or "fastq-sanger" for the Sanger standard using PHRED values, "fastq-solexa" for the original Solexa/Illumina variant, or "fastq-illumina" for the more recent variant), as this cannot be detected reliably automatically.

To illustrate this problem, let's consider an artificial example:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> test = SeqRecord(Seq("NACGTACGTA"), id="Test", description="Made up!")
>>> print(test.format("fasta"))
>Test Made up!
NACGTACGTA

>>> print(test.format("fastq"))
Traceback (most recent call last):
 ...
ValueError: No suitable quality scores found in letter_annotations of SeqRecord␣
↪(id=Test).
```

We created a sample SeqRecord, and can show it in FASTA format - but for QUAL or FASTQ format we need to provide some quality scores. These are held as a list of integers (one for each base) in the letter_annotations dictionary:

```
>>> test.letter_annotations["phred_quality"] = [0, 1, 2, 3, 4, 5, 10, 20, 30, 40]
>>> print(test.format("qual"))
>Test Made up!
0 1 2 3 4 5 10 20 30 40

>>> print(test.format("fastq"))
@Test Made up!
NACGTACGTA
+
!"#$%&+5?I
```

We can check this FASTQ encoding - the first PHRED quality was zero, and this mapped to a exclamation mark, while the final score was 40 and this mapped to the letter "I":

```
>>> ord('!') - 33
0
>>> ord('I') - 33
40
>>> [ord(letter)-33 for letter in '!"#$%&+5?I']
[0, 1, 2, 3, 4, 5, 10, 20, 30, 40]
```

Similarly, we could produce an Illumina 1.3 to 1.7 style FASTQ file using PHRED scores with an offset of 64:

```
>>> print(test.format("fastq-illumina"))
@Test Made up!
NACGTACGTA
+
@ABCDEJT^h
```

And we can check this too - the first PHRED score was zero, and this mapped to "@", while the final score was 40 and this mapped to "h":

```
>>> ord("@") - 64
0
>>> ord("h") - 64
40
>>> [ord(letter)-64 for letter in "@ABCDEJT^h"]
[0, 1, 2, 3, 4, 5, 10, 20, 30, 40]
```

Notice how different the standard Sanger FASTQ and the Illumina 1.3 to 1.7 style FASTQ files look for the same data! Then we have the older Solexa/Illumina format to consider which encodes Solexa scores instead of PHRED scores.

First let's see what Biopython says if we convert the PHRED scores into Solexa scores (rounding to one decimal place):

```
>>> for q in [0, 1, 2, 3, 4, 5, 10, 20, 30, 40]:
...     print("PHRED %i maps to Solexa %0.1f" % (q, solexa_quality_from_phred(q)))
PHRED 0 maps to Solexa -5.0
PHRED 1 maps to Solexa -5.0
PHRED 2 maps to Solexa -2.3
PHRED 3 maps to Solexa -0.0
PHRED 4 maps to Solexa 1.8
PHRED 5 maps to Solexa 3.3
PHRED 10 maps to Solexa 9.5
PHRED 20 maps to Solexa 20.0
```

(continues on next page)

(continued from previous page)

```
PHRED 30 maps to Solexa 30.0
PHRED 40 maps to Solexa 40.0
```

Now here is the record using the old Solexa style FASTQ file:

```
>>> print(test.format("fastq-solexa"))
@Test Made up!
NACGTACGTA
+
;;>@BCJT^h
```

Again, this is using an ASCII offset of 64, so we can check the Solexa scores:

```
>>> [ord(letter)-64 for letter in ";;>@BCJT^h"]
[-5, -5, -2, 0, 2, 3, 10, 20, 30, 40]
```

This explains why the last few letters of this FASTQ output matched that using the Illumina 1.3 to 1.7 format - high quality PHRED scores and Solexa scores are approximately equal.

Bio.SeqIO.QualityIO.**solexa_quality_from_phred**(*phred_quality: float*) → float

Convert a PHRED quality (range 0 to about 90) to a Solexa quality.

PHRED and Solexa quality scores are both log transformations of a probality of error (high score = low probability of error). This function takes a PHRED score, transforms it back to a probability of error, and then re-expresses it as a Solexa score. This assumes the error estimates are equivalent.

How does this work exactly? Well the PHRED quality is minus ten times the base ten logarithm of the probability of error:

```
phred_quality = -10*log(error,10)
```

Therefore, turning this round:

```
error = 10 ** (- phred_quality / 10)
```

Now, Solexa qualities use a different log transformation:

```
solexa_quality = -10*log(error/(1-error),10)
```

After substitution and a little manipulation we get:

```
solexa_quality = 10*log(10**(phred_quality/10.0) - 1, 10)
```

However, real Solexa files use a minimum quality of -5. This does have a good reason - a random base call would be correct 25% of the time, and thus have a probability of error of 0.75, which gives 1.25 as the PHRED quality, or -4.77 as the Solexa quality. Thus (after rounding), a random nucleotide read would have a PHRED quality of 1, or a Solexa quality of -5.

Taken literally, this logarithic formula would map a PHRED quality of zero to a Solexa quality of minus infinity. Of course, taken literally, a PHRED score of zero means a probability of error of one (i.e. the base call is definitely wrong), which is worse than random! In practice, a PHRED quality of zero usually means a default value, or perhaps random - and therefore mapping it to the minimum Solexa score of -5 is reasonable.

In conclusion, we follow EMBOSS, and take this logarithmic formula but also apply a minimum value of -5.0 for the Solexa quality, and also map a PHRED quality of zero to -5.0 as well.

Note this function will return a floating point number, it is up to you to round this to the nearest integer if appropriate. e.g.

```
>>> print("%0.2f" % round(solexa_quality_from_phred(80), 2))
80.00
>>> print("%0.2f" % round(solexa_quality_from_phred(50), 2))
50.00
>>> print("%0.2f" % round(solexa_quality_from_phred(20), 2))
19.96
>>> print("%0.2f" % round(solexa_quality_from_phred(10), 2))
9.54
>>> print("%0.2f" % round(solexa_quality_from_phred(5), 2))
3.35
>>> print("%0.2f" % round(solexa_quality_from_phred(4), 2))
1.80
>>> print("%0.2f" % round(solexa_quality_from_phred(3), 2))
-0.02
>>> print("%0.2f" % round(solexa_quality_from_phred(2), 2))
-2.33
>>> print("%0.2f" % round(solexa_quality_from_phred(1), 2))
-5.00
>>> print("%0.2f" % round(solexa_quality_from_phred(0), 2))
-5.00
```

Notice that for high quality reads PHRED and Solexa scores are numerically equal. The differences are important for poor quality reads, where PHRED has a minimum of zero but Solexa scores can be negative.

Finally, as a special case where None is used for a "missing value", None is returned:

```
>>> print(solexa_quality_from_phred(None))
None
```

Bio.SeqIO.QualityIO.**phred_quality_from_solexa**(*solexa_quality: float*) → float

Convert a Solexa quality (which can be negative) to a PHRED quality.

PHRED and Solexa quality scores are both log transformations of a probality of error (high score = low probability of error). This function takes a Solexa score, transforms it back to a probability of error, and then re-expresses it as a PHRED score. This assumes the error estimates are equivalent.

The underlying formulas are given in the documentation for the sister function solexa_quality_from_phred, in this case the operation is:

```
phred_quality = 10*log(10**(solexa_quality/10.0) + 1, 10)
```

This will return a floating point number, it is up to you to round this to the nearest integer if appropriate. e.g.

```
>>> print("%0.2f" % round(phred_quality_from_solexa(80), 2))
80.00
>>> print("%0.2f" % round(phred_quality_from_solexa(20), 2))
20.04
>>> print("%0.2f" % round(phred_quality_from_solexa(10), 2))
10.41
>>> print("%0.2f" % round(phred_quality_from_solexa(0), 2))
3.01
>>> print("%0.2f" % round(phred_quality_from_solexa(-5), 2))
1.19
```

Note that a solexa_quality less then -5 is not expected, will trigger a warning, but will still be converted as per the logarithmic mapping (giving a number between 0 and 1.19 back).

As a special case where None is used for a "missing value", None is returned:

```
>>> print(phred_quality_from_solexa(None))
None
```

Bio.SeqIO.QualityIO.**FastqGeneralIterator**(*source: IO[str] | PathLike | str | bytes*) → Iterator[tuple[str, str, str]]

Iterate over Fastq records as string tuples (not as SeqRecord objects).

Arguments:

- source - input stream opened in text mode, or a path to a file

This code does not try to interpret the quality string numerically. It just returns tuples of the title, sequence and quality as strings. For the sequence and quality, any whitespace (such as new lines) is removed.

Our SeqRecord based FASTQ iterators call this function internally, and then turn the strings into a SeqRecord objects, mapping the quality string into a list of numerical scores. If you want to do a custom quality mapping, then you might consider calling this function directly.

For parsing FASTQ files, the title string from the "@" line at the start of each record can optionally be omitted on the "+" lines. If it is repeated, it must be identical.

The sequence string and the quality string can optionally be split over multiple lines, although several sources discourage this. In comparison, for the FASTA file format line breaks between 60 and 80 characters are the norm.

WARNING - Because the "@" character can appear in the quality string, this can cause problems as this is also the marker for the start of a new sequence. In fact, the "+" sign can also appear as well. Some sources recommended having no line breaks in the quality to avoid this, but even that is not enough, consider this example:

```
@071113_EAS56_0053:1:1:998:236
TTTCTTGCCCCCATAGACTGAGACCTTCCCTAAATA
+071113_EAS56_0053:1:1:998:236
IIIIIIIIIIIIIIIIIIIIIIIIIIIIICII+III
@071113_EAS56_0053:1:1:182:712
ACCCAGCTAATTTTTGTATTTTTGTTAGAGACAGTG
+
@IIIIIIIIIIIIIIICDIIIII<%<6&-*).(*%+
@071113_EAS56_0053:1:1:153:10
TGTTCTGAAGGAAGGTGTGCGTGCGTGTGTGTGTGT
+
IIIIIIIIIIIICIIGIIIII>IAIIIE65I=II:6
@071113_EAS56_0053:1:3:990:501
TGGGAGGTTTTATGTGGA
AAGCAGCAATGTACAAGA
+
IIIIIII.IIIIII1@44
@-7.%<&+/$/%4(++(%
```

This is four PHRED encoded FASTQ entries originally from an NCBI source (given the read length of 36, these are probably Solexa Illumina reads where the quality has been mapped onto the PHRED values).

This example has been edited to illustrate some of the nasty things allowed in the FASTQ format. Firstly, on the "+" lines most but not all of the (redundant) identifiers are omitted. In real files it is likely that all or none of these extra identifiers will be present.

Secondly, while the first three sequences have been shown without line breaks, the last has been split over multiple lines. In real files any line breaks are likely to be consistent.

Thirdly, some of the quality string lines start with an "@" character. For the second record this is unavoidable. However for the fourth sequence this only happens because its quality string is split over two lines. A naive parser could wrongly treat any line starting with an "@" as the beginning of a new sequence! This code copes with this possible ambiguity by keeping track of the length of the sequence which gives the expected length of the quality string.

Using this tricky example file as input, this short bit of code demonstrates what this parsing function would return:

```
>>> with open("Quality/tricky.fastq") as handle:
...     for (title, sequence, quality) in FastqGeneralIterator(handle):
...         print(title)
...         print("%s %s" % (sequence, quality))
...
071113_EAS56_0053:1:1:998:236
TTTCTTGCCCCCATAGACTGAGACCTTCCCTAAATA IIIIIIIIIIIIIIIIIIIIIIIIIIIIICII+III
071113_EAS56_0053:1:1:182:712
ACCCAGCTAATTTTTGTATTTTTGTTAGAGACAGTG @IIIIIIIIIIIIIIICDIIIII<%<6&-*).(*%+
071113_EAS56_0053:1:1:153:10
TGTTCTGAAGGAAGGTGTGCGTGCGTGTGTGTGTGT IIIIIIIIIIIICIIGIIIII>IAIIIE65I=II:6
071113_EAS56_0053:1:3:990:501
TGGGAGGTTTTATGTGGAAAGCAGCAATGTACAAGA IIIIIII.IIIIII1@44@-7.%<&+/$/%4(++(%
```

Finally we note that some sources state that the quality string should start with "!" (which using the PHRED mapping means the first letter always has a quality score of zero). This rather restrictive rule is not widely observed, so is therefore ignored here. One plus point about this "!" rule is that (provided there are no line breaks in the quality sequence) it would prevent the above problem with the "@" character.

class Bio.SeqIO.QualityIO.**FastqPhredIterator**(*source: IO[str] | PathLike | str | bytes*, *alphabet: None = None*)

Bases: *SequenceIterator*[str]

Parser for FASTQ files.

__init__(*source: IO[str] | PathLike | str | bytes*, *alphabet: None = None*)

Iterate over FASTQ records as SeqRecord objects.

Arguments:

- source - input stream opened in text mode, or a path to a file
- alphabet - optional alphabet, no longer used. Leave as None.

For each sequence in a (Sanger style) FASTQ file there is a matching string encoding the PHRED qualities (integers between 0 and about 90) using ASCII values with an offset of 33.

For example, consider a file containing three short reads:

```
@EAS54_6_R1_2_1_413_324
CCCTTCTTGTCTTCAGCGTTTCTCC
+
;;3;;;;;;;;;;;;7;;;;;;;88
@EAS54_6_R1_2_1_540_792
TTGGCAGGCCAAGGCCGATGGATCA
+
```

(continues on next page)

(continued from previous page)

```
;;;;;;;;;;;7;;;;;-;;;3;83
@EAS54_6_R1_2_1_443_348
GTTGCTTCTGGCGTGGGTGGGGGGG
+
;;;;;;;;;;;9;7;;.7;393333
```

For each sequence (e.g. "CCCTTCTTGTCTTCAGCGTTTCTCC") there is a matching string encoding the PHRED qualities using a ASCII values with an offset of 33 (e.g. ";;3;;;;;;;;;;;;7;;;;;;;88").

Using this module directly you might run:

```
>>> with open("Quality/example.fastq") as handle:
...     for record in FastqPhredIterator(handle):
...         print("%s %s" % (record.id, record.seq))
EAS54_6_R1_2_1_413_324 CCCTTCTTGTCTTCAGCGTTTCTCC
EAS54_6_R1_2_1_540_792 TTGGCAGGCCAAGGCCGATGGATCA
EAS54_6_R1_2_1_443_348 GTTGCTTCTGGCGTGGGTGGGGGGG
```

Typically however, you would call this via Bio.SeqIO instead with "fastq" (or "fastq-sanger") as the format:

```
>>> from Bio import SeqIO
>>> with open("Quality/example.fastq") as handle:
...     for record in SeqIO.parse(handle, "fastq"):
...         print("%s %s" % (record.id, record.seq))
EAS54_6_R1_2_1_413_324 CCCTTCTTGTCTTCAGCGTTTCTCC
EAS54_6_R1_2_1_540_792 TTGGCAGGCCAAGGCCGATGGATCA
EAS54_6_R1_2_1_443_348 GTTGCTTCTGGCGTGGGTGGGGGGG
```

If you want to look at the qualities, they are record in each record's per-letter-annotation dictionary as a simple list of integers:

```
>>> print(record.letter_annotations["phred_quality"])
[26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 24, 26, 22, 26, 26, 13, 22, 26, 18,
↪ 24, 18, 18, 18, 18]
```

To modify the records returned by the parser, you can use a generator function. For example, to store the mean PHRED quality in the record description, use

```
>>> from statistics import mean
>>> def modify_records(records):
...     for record in records:
...         record.description = mean(record.letter_annotations['phred_quality
↪'])
...         yield record
...
>>> with open('Quality/example.fastq') as handle:
...     for record in modify_records(FastqPhredIterator(handle)):
...         print(record.id, record.description)
...
EAS54_6_R1_2_1_413_324 25.28
EAS54_6_R1_2_1_540_792 24.52
EAS54_6_R1_2_1_443_348 23.4
```

parse(*handle: IO[str]*) → Iterator[*SeqRecord*]

Start parsing the file, and return a SeqRecord iterator.

iterate(*handle: IO[str]*) → Iterator[*SeqRecord*]

Parse the file and generate SeqRecord objects.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__orig_bases__ = (Bio.SeqIO.Interfaces.SequenceIterator[str],)

__parameters__ = ()

Bio.SeqIO.QualityIO.**FastqSolexaIterator**(*source: IO[str] | PathLike | str | bytes*, *alphabet: None = None*) → Iterator[*SeqRecord*]

Parse old Solexa/Illumina FASTQ like files (which differ in the quality mapping).

The optional arguments are the same as those for the FastqPhredIterator.

For each sequence in Solexa/Illumina FASTQ files there is a matching string encoding the Solexa integer qualities using ASCII values with an offset of 64. Solexa scores are scaled differently to PHRED scores, and Biopython will NOT perform any automatic conversion when loading.

NOTE - This file format is used by the OLD versions of the Solexa/Illumina pipeline. See also the FastqIlluminaIterator function for the NEW version.

For example, consider a file containing these five records:

```
@SLXA-B3_649_FC8437_R1_1_1_610_79
GATGTGCAATACCTTTGTAGAGGAA
+SLXA-B3_649_FC8437_R1_1_1_610_79
YYYYYYYYYYYYYYYYYYWYWYYSU
@SLXA-B3_649_FC8437_R1_1_1_397_389
GGTTTGAGAAAGAGAAATGAGATAA
+SLXA-B3_649_FC8437_R1_1_1_397_389
YYYYYYYYYWYYYYWWYYYWYWYWW
@SLXA-B3_649_FC8437_R1_1_1_850_123
GAGGGTGTTGATCATGATGATGGCG
+SLXA-B3_649_FC8437_R1_1_1_850_123
YYYYYYYYYYYYYWYYWYYSYYYSY
@SLXA-B3_649_FC8437_R1_1_1_362_549
GGAAACAAAGTTTTTCTCAACATAG
+SLXA-B3_649_FC8437_R1_1_1_362_549
YYYYYYYYYYYYYYYYYYWWWWYWY
@SLXA-B3_649_FC8437_R1_1_1_183_714
GTATTATTTAATGGCATACACTCAA
+SLXA-B3_649_FC8437_R1_1_1_183_714
YYYYYYYYYYWYYYYWYWWUWWWQQ
```

Using this module directly you might run:

```
>>> with open("Quality/solexa_example.fastq") as handle:
...     for record in FastqSolexaIterator(handle):
...         print("%s %s" % (record.id, record.seq))
SLXA-B3_649_FC8437_R1_1_1_610_79 GATGTGCAATACCTTTGTAGAGGAA
SLXA-B3_649_FC8437_R1_1_1_397_389 GGTTTGAGAAAGAGAAATGAGATAA
```

(continues on next page)

(continued from previous page)

```
SLXA-B3_649_FC8437_R1_1_1_850_123 GAGGGTGTTGATCATGATGATGGCG
SLXA-B3_649_FC8437_R1_1_1_362_549 GGAAACAAAGTTTTTCTCAACATAG
SLXA-B3_649_FC8437_R1_1_1_183_714 GTATTATTTAATGGCATACACTCAA
```

Typically however, you would call this via Bio.SeqIO instead with "fastq-solexa" as the format:

```
>>> from Bio import SeqIO
>>> with open("Quality/solexa_example.fastq") as handle:
...     for record in SeqIO.parse(handle, "fastq-solexa"):
...         print("%s %s" % (record.id, record.seq))
SLXA-B3_649_FC8437_R1_1_1_610_79 GATGTGCAATACCTTTGTAGAGGAA
SLXA-B3_649_FC8437_R1_1_1_397_389 GGTTTGAGAAAGAGAAATGAGATAA
SLXA-B3_649_FC8437_R1_1_1_850_123 GAGGGTGTTGATCATGATGATGGCG
SLXA-B3_649_FC8437_R1_1_1_362_549 GGAAACAAAGTTTTTCTCAACATAG
SLXA-B3_649_FC8437_R1_1_1_183_714 GTATTATTTAATGGCATACACTCAA
```

If you want to look at the qualities, they are recorded in each record's per-letter-annotation dictionary as a simple list of integers:

```
>>> print(record.letter_annotations["solexa_quality"])
[25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 23, 25, 25, 25, 25, 23, 25, 23, 23, 21, 23,
↪ 23, 23, 17, 17]
```

These scores aren't very good, but they are high enough that they map almost exactly onto PHRED scores:

```
>>> print("%0.2f" % phred_quality_from_solexa(25))
25.01
```

Let's look at faked example read which is even worse, where there are more noticeable differences between the Solexa and PHRED scores:

```
@slxa_0001_1_0001_01
ACGTACGTACGTACGTACGTACGTACGTACGTACGTNNNNNN
+slxa_0001_1_0001_01
hgfedcba`_^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA@?>=<;
```

Again, you would typically use Bio.SeqIO to read this file in (rather than calling the Bio.SeqIO.QualtityIO module directly). Most FASTQ files will contain thousands of reads, so you would normally use Bio.SeqIO.parse() as shown above. This example has only as one entry, so instead we can use the Bio.SeqIO.read() function:

```
>>> from Bio import SeqIO
>>> with open("Quality/solexa_faked.fastq") as handle:
...     record = SeqIO.read(handle, "fastq-solexa")
>>> print("%s %s" % (record.id, record.seq))
slxa_0001_1_0001_01 ACGTACGTACGTACGTACGTACGTACGTACGTACGTNNNNNN
>>> print(record.letter_annotations["solexa_quality"])
[40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20,
↪ 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -3,
↪ -4, -5]
```

These quality scores are so low that when converted from the Solexa scheme into PHRED scores they look quite different:

```
>>> print("%0.2f" % phred_quality_from_solexa(-1))
2.54
>>> print("%0.2f" % phred_quality_from_solexa(-5))
1.19
```

Note you can use the Bio.SeqIO.write() function or the SeqRecord's format method to output the record(s):

```
>>> print(record.format("fastq-solexa"))
@slxa_0001_1_0001_01
ACGTACGTACGTACGTACGTACGTACGTACGTACGTNNNNNN
+
hgfedcba`_^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA@?>=<;
```

Note this output is slightly different from the input file as Biopython has left out the optional repetition of the sequence identifier on the "+" line. If you want the to use PHRED scores, use "fastq" or "qual" as the output format instead, and Biopython will do the conversion for you:

```
>>> print(record.format("fastq"))
@slxa_0001_1_0001_01
ACGTACGTACGTACGTACGTACGTACGTACGTACGTNNNNNN
+
IHGFEDCBA@?>=<;:9876543210/.-,++*)('&&%%$$##""
```

```
>>> print(record.format("qual"))
>slxa_0001_1_0001_01
40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21
20 19 18 17 16 15 14 13 12 11 10 10 9 8 7 6 5 5 4 4 3 3 2 2
1 1
```

As shown above, the poor quality Solexa reads have been mapped to the equivalent PHRED score (e.g. -5 to 1 as shown earlier).

Bio.SeqIO.QualityIO.**FastqIlluminaIterator**(*source: IO[str] | PathLike | str | bytes*, *alphabet: None = None*) → Iterator[*SeqRecord*]

Parse Illumina 1.3 to 1.7 FASTQ like files (which differ in the quality mapping).

The optional arguments are the same as those for the FastqPhredIterator.

For each sequence in Illumina 1.3+ FASTQ files there is a matching string encoding PHRED integer qualities using ASCII values with an offset of 64.

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("Quality/illumina_faked.fastq", "fastq-illumina")
>>> print("%s %s" % (record.id, record.seq))
Test ACGTACGTACGTACGTACGTACGTACGTACGTACGTN
>>> max(record.letter_annotations["phred_quality"])
40
>>> min(record.letter_annotations["phred_quality"])
0
```

NOTE - Older versions of the Solexa/Illumina pipeline encoded Solexa scores with an ASCII offset of 64. They are approximately equal but only for high quality reads. If you have an old Solexa/Illumina file with negative Solexa scores, and try and read this as an Illumina 1.3+ file it will fail:

```
>>> record2 = SeqIO.read("Quality/solexa_faked.fastq", "fastq-illumina")
Traceback (most recent call last):
   ...
ValueError: Invalid character in quality string
```

NOTE - True Sanger style FASTQ files use PHRED scores with an offset of 33.

class Bio.SeqIO.QualityIO.**QualPhredIterator**(*source: IO[str] | PathLike | str | bytes*, *alphabet: None = None*)

Bases: *SequenceIterator*

Parser for QUAL files with PHRED quality scores but no sequence.

__init__(*source: IO[str] | PathLike | str | bytes*, *alphabet: None = None*) → None

For QUAL files which include PHRED quality scores, but no sequence.

For example, consider this short QUAL file:

```
>EAS54_6_R1_2_1_413_324
26 26 18 26 26 26 26 26 26 26 26 26 26 26 26 22 26 26 26 26
26 26 26 23 23
>EAS54_6_R1_2_1_540_792
26 26 26 26 26 26 26 26 26 26 26 22 26 26 26 26 26 12 26 26
26 18 26 23 18
>EAS54_6_R1_2_1_443_348
26 26 26 26 26 26 26 26 26 26 26 24 26 22 26 26 13 22 26 18
24 18 18 18 18
```

Using this module directly you might run:

```
>>> with open("Quality/example.qual") as handle:
...     for record in QualPhredIterator(handle):
...         print("%s read of length %d" % (record.id, len(record.seq)))
EAS54_6_R1_2_1_413_324 read of length 25
EAS54_6_R1_2_1_540_792 read of length 25
EAS54_6_R1_2_1_443_348 read of length 25
```

Typically however, you would call this via Bio.SeqIO instead with "qual" as the format:

```
>>> from Bio import SeqIO
>>> with open("Quality/example.qual") as handle:
...     for record in SeqIO.parse(handle, "qual"):
...         print("%s read of length %d" % (record.id, len(record.seq)))
EAS54_6_R1_2_1_413_324 read of length 25
EAS54_6_R1_2_1_540_792 read of length 25
EAS54_6_R1_2_1_443_348 read of length 25
```

Only the sequence length is known, as the QUAL file does not contain the sequence string itself.

The quality scores themselves are available as a list of integers in each record's per-letter-annotation:

```
>>> print(record.letter_annotations["phred_quality"])
[26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 24, 26, 22, 26, 26, 13, 22, 26, 18,
↪ 24, 18, 18, 18, 18]
```

You can still slice one of these SeqRecord objects:

```
>>> sub_record = record[5:10]
>>> print("%s %s" % (sub_record.id, sub_record.letter_annotations["phred_quality
↪"]))
EAS54_6_R1_2_1_443_348 [26, 26, 26, 26, 26]
```

As of Biopython 1.59, this parser will accept files with negatives quality scores but will replace them with the lowest possible PHRED score of zero. This will trigger a warning, previously it raised a ValueError exception.

parse(*handle: IO*) → Iterator[*SeqRecord*]

Start parsing the file, and return a SeqRecord iterator.

iterate(*handle: IO*) → Iterator[*SeqRecord*]

Parse the file and generate SeqRecord objects.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.QualityIO.**FastqPhredWriter**(*target: IO | PathLike | str | bytes*, *mode: str = 'w'*)

Bases: *SequenceWriter*

Class to write standard FASTQ format files (using PHRED quality scores) (OBSOLETE).

Although you can use this class directly, you are strongly encouraged to use the `as_fastq` function, or top level `Bio.SeqIO.write()` function instead via the format name "fastq" or the alias "fastq-sanger".

For example, this code reads in a standard Sanger style FASTQ file (using PHRED scores) and re-saves it as another Sanger style FASTQ file:

```
>>> from Bio import SeqIO
>>> record_iterator = SeqIO.parse("Quality/example.fastq", "fastq")
>>> with open("Quality/temp.fastq", "w") as out_handle:
...     SeqIO.write(record_iterator, out_handle, "fastq")
3
```

You might want to do this if the original file included extra line breaks, which while valid may not be supported by all tools. The output file from Biopython will have each sequence on a single line, and each quality string on a single line (which is considered desirable for maximum compatibility).

In this next example, an old style Solexa/Illumina FASTQ file (using Solexa quality scores) is converted into a standard Sanger style FASTQ file using PHRED qualities:

```
>>> from Bio import SeqIO
>>> record_iterator = SeqIO.parse("Quality/solexa_example.fastq", "fastq-solexa")
>>> with open("Quality/temp.fastq", "w") as out_handle:
...     SeqIO.write(record_iterator, out_handle, "fastq")
5
```

This code is also called if you use the .format("fastq") method of a SeqRecord, or .format("fastq-sanger") if you prefer that alias.

Note that Sanger FASTQ files have an upper limit of PHRED quality 93, which is encoded as ASCII 126, the tilde. If your quality scores are truncated to fit, a warning is issued.

P.S. To avoid cluttering up your working directory, you can delete this temporary file now:

```
>>> import os
>>> os.remove("Quality/temp.fastq")
```

write_record(*record:* SeqRecord) → None

Write a single FASTQ record to the file.

__annotations__ = {}

Bio.SeqIO.QualityIO.**as_fastq**(*record:* SeqRecord) → str

Turn a SeqRecord into a Sanger FASTQ formatted string.

This is used internally by the SeqRecord's .format("fastq") method and by the SeqIO.write(..., ..., "fastq") function, and under the format alias "fastq-sanger" as well.

class Bio.SeqIO.QualityIO.**QualPhredWriter**(*handle: IO[str] | PathLike | str | bytes, wrap: int = 60, record2title: Callable[[SeqRecord], str] | None = None*)

Bases: *SequenceWriter*

Class to write QUAL format files (using PHRED quality scores) (OBSOLETE).

Although you can use this class directly, you are strongly encouraged to use the `as_qual` function, or top level `Bio.SeqIO.write()` function instead.

For example, this code reads in a FASTQ file and saves the quality scores into a QUAL file:

```
>>> from Bio import SeqIO
>>> record_iterator = SeqIO.parse("Quality/example.fastq", "fastq")
>>> with open("Quality/temp.qual", "w") as out_handle:
...     SeqIO.write(record_iterator, out_handle, "qual")
3
```

This code is also called if you use the .format("qual") method of a SeqRecord.

P.S. Don't forget to clean up the temp file if you don't need it anymore:

```
>>> import os
>>> os.remove("Quality/temp.qual")
```

__init__(*handle: IO[str] | PathLike | str | bytes, wrap: int = 60, record2title: Callable[[SeqRecord], str] | None = None*) → None

Create a QUAL writer.

Arguments:

- handle - Handle to an output file, e.g. as returned by open(filename, "w")
- wrap - Optional line length used to wrap sequence lines. Defaults to wrapping the sequence at 60 characters. Use zero (or None) for no wrapping, giving a single long line for the sequence.
- record2title - Optional function to return the text to be used for the title line of each record. By default a combination of the record.id and record.description is used. If the record.description starts with the record.id, then just the record.description is used.

The record2title argument is present for consistency with the Bio.SeqIO.FastaIO writer class.

write_record(*record:* SeqRecord) → None

Write a single QUAL record to the file.

__annotations__ = {}

Bio.SeqIO.QualityIO.**as_qual**(*record:* SeqRecord) → str

Turn a SeqRecord into a QUAL formatted string.

This is used internally by the SeqRecord's .format("qual") method and by the SeqIO.write(..., ..., "qual") function.

class Bio.SeqIO.QualityIO.**FastqSolexaWriter**(*target: IO | PathLike | str | bytes*, *mode: str = 'w'*)

Bases: *SequenceWriter*

Write old style Solexa/Illumina FASTQ format files (with Solexa qualities) (OBSOLETE).

This outputs FASTQ files like those from the early Solexa/Illumina pipeline, using Solexa scores and an ASCII offset of 64. These are NOT compatible with the standard Sanger style PHRED FASTQ files.

If your records contain a "solexa_quality" entry under letter_annotations, this is used, otherwise any "phred_quality" entry will be used after conversion using the solexa_quality_from_phred function. If neither style of quality scores are present, an exception is raised.

Although you can use this class directly, you are strongly encouraged to use the `as_fastq_solexa` function, or top-level `Bio.SeqIO.write()` function instead. For example, this code reads in a FASTQ file and re-saves it as another FASTQ file:

```
>>> from Bio import SeqIO
>>> record_iterator = SeqIO.parse("Quality/solexa_example.fastq", "fastq-solexa")
>>> with open("Quality/temp.fastq", "w") as out_handle:
...     SeqIO.write(record_iterator, out_handle, "fastq-solexa")
5
```

You might want to do this if the original file included extra line breaks, which (while valid) may not be supported by all tools. The output file from Biopython will have each sequence on a single line, and each quality string on a single line (which is considered desirable for maximum compatibility).

This code is also called if you use the .format("fastq-solexa") method of a SeqRecord. For example,

```
>>> record = SeqIO.read("Quality/sanger_faked.fastq", "fastq-sanger")
>>> print(record.format("fastq-solexa"))
@Test PHRED qualities from 40 to 0 inclusive
ACGTACGTACGTACGTACGTACGTACGTACGTACGTN
+
hgfedcba`_^]\[ZYXWVUTSRQPONMLKJHGFECB@>;;
```

Note that Solexa FASTQ files have an upper limit of Solexa quality 62, which is encoded as ASCII 126, the tilde. If your quality scores must be truncated to fit, a warning is issued.

P.S. Don't forget to delete the temp file if you don't need it anymore:

```
>>> import os
>>> os.remove("Quality/temp.fastq")
```

write_record(*record:* SeqRecord) → None

Write a single FASTQ record to the file.

__annotations__ = {}

Bio.SeqIO.QualityIO.**as_fastq_solexa**(*record:* SeqRecord) → str

Turn a SeqRecord into a Solexa FASTQ formatted string.

This is used internally by the SeqRecord's .format("fastq-solexa") method and by the SeqIO.write(..., ..., "fastq-solexa") function.

class Bio.SeqIO.QualityIO.**FastqIlluminaWriter**(*target: IO | PathLike | str | bytes*, *mode: str = 'w'*)

Bases: *SequenceWriter*

Write Illumina 1.3+ FASTQ format files (with PHRED quality scores) (OBSOLETE).

This outputs FASTQ files like those from the Solexa/Illumina 1.3+ pipeline, using PHRED scores and an ASCII offset of 64. Note these files are NOT compatible with the standard Sanger style PHRED FASTQ files which use an ASCII offset of 32.

Although you can use this class directly, you are strongly encouraged to use the `as_fastq_illumina` or top-level `Bio.SeqIO.write()` function with format name "fastq-illumina" instead. This code is also called if you use the .format("fastq-illumina") method of a SeqRecord. For example,

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("Quality/sanger_faked.fastq", "fastq-sanger")
>>> print(record.format("fastq-illumina"))
@Test PHRED qualities from 40 to 0 inclusive
ACGTACGTACGTACGTACGTACGTACGTACGTACGTN
+
hgfedcba`_^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA@
```

Note that Illumina FASTQ files have an upper limit of PHRED quality 62, which is encoded as ASCII 126, the tilde. If your quality scores are truncated to fit, a warning is issued.

write_record(*record:* SeqRecord) → None

Write a single FASTQ record to the file.

__annotations__ = {}

Bio.SeqIO.QualityIO.**as_fastq_illumina**(*record:* SeqRecord) → str

Turn a SeqRecord into an Illumina FASTQ formatted string.

This is used internally by the SeqRecord's .format("fastq-illumina") method and by the SeqIO.write(..., ..., "fastq-illumina") function.

Bio.SeqIO.QualityIO.**PairedFastaQualIterator**(*fasta_source: IO[str] | PathLike | str | bytes*, *qual_source: IO[str] | PathLike | str | bytes*, *alphabet: None = None*) → Iterator[*SeqRecord*]

Iterate over matched FASTA and QUAL files as SeqRecord objects.

For example, consider this short QUAL file with PHRED quality scores:

```
>EAS54_6_R1_2_1_413_324
26 26 18 26 26 26 26 26 26 26 26 26 26 26 22 26 26 26 26
26 26 26 23 23
>EAS54_6_R1_2_1_540_792
26 26 26 26 26 26 26 26 26 26 26 22 26 26 26 26 26 12 26 26
26 18 26 23 18
>EAS54_6_R1_2_1_443_348
26 26 26 26 26 26 26 26 26 26 26 24 26 22 26 26 13 22 26 18
24 18 18 18 18
```

And a matching FASTA file:

```
>EAS54_6_R1_2_1_413_324
CCCTTCTTGTCTTCAGCGTTTCTCC
>EAS54_6_R1_2_1_540_792
```

(continues on next page)

(continued from previous page)

```
TTGGCAGGCCAAGGCCGATGGATCA
>EAS54_6_R1_2_1_443_348
GTTGCTTCTGGCGTGGGTGGGGGGG
```

You can parse these separately using Bio.SeqIO with the "qual" and "fasta" formats, but then you'll get a group of SeqRecord objects with no sequence, and a matching group with the sequence but not the qualities. Because it only deals with one input file handle, Bio.SeqIO can't be used to read the two files together - but this function can! For example,

```
>>> with open("Quality/example.fasta") as f:
...     with open("Quality/example.qual") as q:
...         for record in PairedFastaQualIterator(f, q):
...             print("%s %s" % (record.id, record.seq))
...
EAS54_6_R1_2_1_413_324 CCCTTCTTGTCTTCAGCGTTTCTCC
EAS54_6_R1_2_1_540_792 TTGGCAGGCCAAGGCCGATGGATCA
EAS54_6_R1_2_1_443_348 GTTGCTTCTGGCGTGGGTGGGGGGG
```

As with the FASTQ or QUAL parsers, if you want to look at the qualities, they are in each record's per-letter-annotation dictionary as a simple list of integers:

```
>>> print(record.letter_annotations["phred_quality"])
[26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 24, 26, 22, 26, 26, 13, 22, 26, 18, 24,
↪ 18, 18, 18, 18]
```

If you have access to data as a FASTQ format file, using that directly would be simpler and more straight forward. Note that you can easily use this function to convert paired FASTA and QUAL files into FASTQ files:

```
>>> from Bio import SeqIO
>>> with open("Quality/example.fasta") as f:
...     with open("Quality/example.qual") as q:
...         SeqIO.write(PairedFastaQualIterator(f, q), "Quality/temp.fastq", "fastq
↪")
...
3
```

And don't forget to clean up the temp file if you don't need it anymore:

```
>>> import os
>>> os.remove("Quality/temp.fastq")
```

Bio.SeqIO.SeqXmlIO module

Bio.SeqIO support for the "seqxml" file format, SeqXML.

This module is for reading and writing SeqXML format files as SeqRecord objects, and is expected to be used via the Bio.SeqIO API.

SeqXML is a lightweight XML format which is supposed be an alternative for FASTA files. For more Information see http://www.seqXML.org and Schmitt et al (2011), https://doi.org/10.1093/bib/bbr025

class Bio.SeqIO.SeqXmlIO.**ContentHandler**

Bases: ContentHandler

Handles XML events generated by the parser (PRIVATE).

__init__()

Create a handler to handle XML events.

startDocument()

Set XML handlers when an XML declaration is found.

startSeqXMLElement(*name*, *qname*, *attrs*)

Handle start of a seqXML element.

endSeqXMLElement(*name*, *qname*)

Handle end of the seqXML element.

startEntryElement(*name*, *qname*, *attrs*)

Set new entry with id and the optional entry source (PRIVATE).

endEntryElement(*name*, *qname*)

Handle end of an entry element.

startEntryFieldElementVersion01(*name*, *qname*, *attrs*)

Receive a field of an entry element and forward it for version 0.1.

startEntryFieldElement(*name*, *qname*, *attrs*)

Receive a field of an entry element and forward it for versions >=0.2.

startSpeciesElement(*attrs*)

Parse the species information.

endSpeciesElement(*name*, *qname*)

Handle end of a species element.

startDescriptionElement(*attrs*)

Parse the description.

endDescriptionElement(*name*, *qname*)

Handle the end of a description element.

startSequenceElement(*attrs*)

Parse DNA, RNA, or protein sequence.

endSequenceElement(*name*, *qname*)

Handle the end of a sequence element.

startDBRefElement(*attrs*)

Parse a database cross reference.

endDBRefElement(*name*, *qname*)

Handle the end of a DBRef element.

startPropertyElement(*attrs*)

Handle the start of a property element.

endPropertyElement(*name*, *qname*)

Handle the end of a property element.

characters(*data*)

Handle character data.

__annotations__ = {}

class Bio.SeqIO.SeqXmlIO.SeqXmlIterator(*stream_or_path*, *namespace=None*)

Bases: SequenceIterator

Parser for seqXML files.

Parses seqXML files and creates SeqRecords. Assumes valid seqXML please validate beforehand. It is assumed that all information for one record can be found within a record element or above. Two types of methods are called when the start tag of an element is reached. To receive only the attributes of an element before its end tag is reached implement _attr_TAGNAME. To get an element and its children as a DOM tree implement _elem_TAGNAME. Everything that is part of the DOM tree will not trigger any further method calls.

BLOCK = 1024

__init__(*stream_or_path*, *namespace=None*)

Create the object and initialize the XML parser.

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

iterate(*handle*)

Iterate over the records in the XML file.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.SeqXmlIO.SeqXmlWriter(*target*, *source=None*, *source_version=None*, *species=None*, *ncbiTaxId=None*)

Bases: SequenceWriter

Writes SeqRecords into seqXML file.

SeqXML requires the SeqRecord annotations to specify the molecule_type; the molecule type is required to contain the term "DNA", "RNA", or "protein".

__init__(*target*, *source=None*, *source_version=None*, *species=None*, *ncbiTaxId=None*)

Create Object and start the xml generator.

Arguments:

- target - Output stream opened in binary mode, or a path to a file.
- source - The source program/database of the file, for example UniProt.
- source_version - The version or release number of the source program or database from which the data originated.
- species - The scientific name of the species of origin of all entries in the file.
- ncbiTaxId - The NCBI taxonomy identifier of the species of origin.

write_header()

Write root node with document metadata.

write_record(*record*)

Write one record.

write_footer()

Close the root node and finish the XML document.

__annotations__ = {}

Bio.SeqIO.SffIO module

Bio.SeqIO support for the binary Standard Flowgram Format (SFF) file format.

SFF was designed by 454 Life Sciences (Roche), the Whitehead Institute for Biomedical Research and the Wellcome Trust Sanger Institute. SFF was also used as the native output format from early versions of Ion Torrent's PGM platform as well. You are expected to use this module via the Bio.SeqIO functions under the format name "sff" (or "sff-trim" as described below).

For example, to iterate over the records in an SFF file,

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("Roche/E3MFGYR02_random_10_reads.sff", "sff"):
...     print("%s %i %s..." % (record.id, len(record), record.seq[:20]))
...
E3MFGYR02JWQ7T 265 tcagGGTCTACATGTTGGTT...
E3MFGYR02JA6IL 271 tcagTTTTTTTTGGAAAGGA...
E3MFGYR02JHD4H 310 tcagAAAGACAAGTGGTATC...
E3MFGYR02GFKUC 299 tcagCGGCCGGGCCTCTCAT...
E3MFGYR02FTGED 281 tcagTGGTAATGGGGGGAAA...
E3MFGYR02FR9G7 261 tcagCTCCGTAAGAAGGTGC...
E3MFGYR02GAZMS 278 tcagAAAGAAGTAAGGTAAA...
E3MFGYR02HHZ8O 221 tcagACTTTCTTCTTTACCG...
E3MFGYR02GPGB1 269 tcagAAGCAGTGGTATCAAC...
E3MFGYR02F7Z7G 219 tcagAATCATCCACTTTTTA...
```

Each SeqRecord object will contain all the annotation from the SFF file, including the PHRED quality scores.

```
>>> print("%s %i" % (record.id, len(record)))
E3MFGYR02F7Z7G 219
>>> print("%s..." % record.seq[:10])
tcagAATCAT...
>>> print("%r..." % (record.letter_annotations["phred_quality"][:10]))
[22, 21, 23, 28, 26, 15, 12, 21, 28, 21]...
```

Notice that the sequence is given in mixed case, the central upper case region corresponds to the trimmed sequence. This matches the output of the Roche tools (and the 3rd party tool sff_extract) for SFF to FASTA.

```
>>> print(record.annotations["clip_qual_left"])
4
>>> print(record.annotations["clip_qual_right"])
134
>>> print(record.seq[:4])
tcag
>>> print("%s...%s" % (record.seq[4:20], record.seq[120:134]))
AATCATCCACTTTTTA...CAAAACACAAACAG
>>> print(record.seq[134:])
atcttatcaacaaaactcaaagttcctaactgagacacgcaacaggggataagacaaggcacacaggggataggnnnnnnnnnnn
```

The annotations dictionary also contains any adapter clip positions (usually zero), and information about the flows. e.g.

```
>>> len(record.annotations)
12
>>> print(record.annotations["flow_key"])
TCAG
>>> print(record.annotations["flow_values"][:10])
(83, 1, 128, 7, 4, 84, 6, 106, 3, 172)
>>> print(len(record.annotations["flow_values"]))
400
>>> print(record.annotations["flow_index"][:10])
(1, 2, 3, 2, 2, 0, 3, 2, 3, 3)
>>> print(len(record.annotations["flow_index"]))
219
```

Note that to convert from a raw reading in flow_values to the corresponding homopolymer stretch estimate, the value should be rounded to the nearest 100:

```
>>> print("%r..." % [int(round(value, -2)) // 100
...                  for value in record.annotations["flow_values"][:10]])
...
[1, 0, 1, 0, 0, 1, 0, 1, 0, 2]...
```

If a read name is exactly 14 alphanumeric characters, the annotations dictionary will also contain meta-data about the read extracted by interpreting the name as a 454 Sequencing System "Universal" Accession Number. Note that if a read name happens to be exactly 14 alphanumeric characters but was not generated automatically, these annotation records will contain nonsense information.

```
>>> print(record.annotations["region"])
2
>>> print(record.annotations["time"])
[2008, 1, 9, 16, 16, 0]
>>> print(record.annotations["coords"])
(2434, 1658)
```

As a convenience method, you can read the file with SeqIO format name "sff-trim" instead of "sff" to get just the trimmed sequences (without any annotation except for the PHRED quality scores and anything encoded in the read names):

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("Roche/E3MFGYR02_random_10_reads.sff", "sff-trim"):
...     print("%s %i %s..." % (record.id, len(record), record.seq[:20]))
...
E3MFGYR02JWQ7T 260 GGTCTACATGTTGGTTAACC...
E3MFGYR02JA6IL 265 TTTTTTTTGGAAAGGAAAAC...
E3MFGYR02JHD4H 292 AAAGACAAGTGGTATCAACG...
E3MFGYR02GFKUC 295 CGGCCGGGCCTCTCATCGGT...
E3MFGYR02FTGED 277 TGGTAATGGGGGGAAATTTA...
E3MFGYR02FR9G7 256 CTCCGTAAGAAGGTGCTGCC...
E3MFGYR02GAZMS 271 AAAGAAGTAAGGTAAATAAC...
E3MFGYR02HHZ8O 150 ACTTTCTTCTTTACCGTAAC...
E3MFGYR02GPGB1 221 AAGCAGTGGTATCAACGCAG...
E3MFGYR02F7Z7G 130 AATCATCCACTTTTTAACGT...
```

Looking at the final record in more detail, note how this differs to the example above:

```
>>> print("%s %i" % (record.id, len(record)))
E3MFGYR02F7Z7G 130
>>> print("%s..." % record.seq[:10])
AATCATCCAC...
>>> print("%r..." % record.letter_annotations["phred_quality"][:10])
[26, 15, 12, 21, 28, 21, 36, 28, 27, 27]...
>>> len(record.annotations)
4
>>> print(record.annotations["region"])
2
>>> print(record.annotations["coords"])
(2434, 1658)
>>> print(record.annotations["time"])
[2008, 1, 9, 16, 16, 0]
>>> print(record.annotations["molecule_type"])
DNA
```

You might use the Bio.SeqIO.convert() function to convert the (trimmed) SFF reads into a FASTQ file (or a FASTA file and a QUAL file), e.g.

```
>>> from Bio import SeqIO
>>> from io import StringIO
>>> out_handle = StringIO()
>>> count = SeqIO.convert("Roche/E3MFGYR02_random_10_reads.sff", "sff",
...                       out_handle, "fastq")
...
>>> print("Converted %i records" % count)
Converted 10 records
```

The output FASTQ file would start like this:

```
>>> print("%s..." % out_handle.getvalue()[:50])
@E3MFGYR02JWQ7T
tcagGGTCTACATGTTGGTTAACCCGTACTGATT...
```

Bio.SeqIO.index() provides memory efficient random access to the reads in an SFF file by name. SFF files can include an index within the file, which can be read in making this very fast. If the index is missing (or in a format not yet supported in Biopython) the file is indexed by scanning all the reads - which is a little slower. For example,

```
>>> from Bio import SeqIO
>>> reads = SeqIO.index("Roche/E3MFGYR02_random_10_reads.sff", "sff")
>>> record = reads["E3MFGYR02JHD4H"]
>>> print("%s %i %s..." % (record.id, len(record), record.seq[:20]))
E3MFGYR02JHD4H 310 tcagAAAGACAAGTGGTATC...
>>> reads.close()
```

Or, using the trimmed reads:

```
>>> from Bio import SeqIO
>>> reads = SeqIO.index("Roche/E3MFGYR02_random_10_reads.sff", "sff-trim")
>>> record = reads["E3MFGYR02JHD4H"]
>>> print("%s %i %s..." % (record.id, len(record), record.seq[:20]))
E3MFGYR02JHD4H 292 AAAGACAAGTGGTATCAACG...
>>> reads.close()
```

You can also use the Bio.SeqIO.write() function with the "sff" format. Note that this requires all the flow information etc, and thus is probably only useful for SeqRecord objects originally from reading another SFF file (and not the trimmed SeqRecord objects from parsing an SFF file as "sff-trim").

As an example, let's pretend this example SFF file represents some DNA which was pre-amplified with a PCR primers AAAGANNNNN. The following script would produce a sub-file containing all those reads whose post-quality clipping region (i.e. the sequence after trimming) starts with AAAGA exactly (the non- degenerate bit of this pretend primer):

```
>>> from Bio import SeqIO
>>> records = (record for record in
...            SeqIO.parse("Roche/E3MFGYR02_random_10_reads.sff", "sff")
...            if record.seq[record.annotations["clip_qual_left"]:].startswith("AAAGA"))
...
>>> count = SeqIO.write(records, "temp_filtered.sff", "sff")
>>> print("Selected %i records" % count)
Selected 2 records
```

Of course, for an assembly you would probably want to remove these primers. If you want FASTA or FASTQ output, you could just slice the SeqRecord. However, if you want SFF output we have to preserve all the flow information - the trick is just to adjust the left clip position!

```
>>> from Bio import SeqIO
>>> def filter_and_trim(records, primer):
...     for record in records:
...         if record.seq[record.annotations["clip_qual_left"]:].startswith(primer):
...             record.annotations["clip_qual_left"] += len(primer)
...             yield record
...
>>> records = SeqIO.parse("Roche/E3MFGYR02_random_10_reads.sff", "sff")
>>> count = SeqIO.write(filter_and_trim(records, "AAAGA"),
...                     "temp_filtered.sff", "sff")
...
>>> print("Selected %i records" % count)
Selected 2 records
```

We can check the results, note the lower case clipped region now includes the "AAAGA" sequence:

```
>>> for record in SeqIO.parse("temp_filtered.sff", "sff"):
...     print("%s %i %s..." % (record.id, len(record), record.seq[:20]))
...
E3MFGYR02JHD4H 310 tcagaaagaCAAGTGGTATC...
E3MFGYR02GAZMS 278 tcagaaagaAGTAAGGTAAA...
>>> for record in SeqIO.parse("temp_filtered.sff", "sff-trim"):
...     print("%s %i %s..." % (record.id, len(record), record.seq[:20]))
...
E3MFGYR02JHD4H 287 CAAGTGGTATCAACGCAGAG...
E3MFGYR02GAZMS 266 AGTAAGGTAAATAACAAACG...
>>> import os
>>> os.remove("temp_filtered.sff")
```

For a description of the file format, please see the Roche manuals and: http://www.ncbi.nlm.nih.gov/Traces/trace.cgi?cmd=show&f=formats&m=doc&s=formats

Bio.SeqIO.SffIO.**ReadRocheXmlManifest**(*handle*)

Read any Roche style XML manifest data in the SFF "index".

The SFF file format allows for multiple different index blocks, and Roche took advantage of this to define their own index block which also embeds an XML manifest string. This is not a publicly documented extension to the SFF file format, this was reverse engineered.

The handle should be to an SFF file opened in binary mode. This function will use the handle seek/tell functions and leave the handle in an arbitrary location.

Any XML manifest found is returned as a Python string, which you can then parse as appropriate, or reuse when writing out SFF files with the SffWriter class.

Returns a string, or raises a ValueError if an Roche manifest could not be found.

class Bio.SeqIO.SffIO.**SffIterator**(*source*, *alphabet=None*, *trim=False*)

Bases: *SequenceIterator*

Parser for Standard Flowgram Format (SFF) files.

__init__(*source*, *alphabet=None*, *trim=False*)

Iterate over Standard Flowgram Format (SFF) reads (as SeqRecord objects).

- source - path to an SFF file, e.g. from Roche 454 sequencing, or a file-like object opened in binary mode.
- alphabet - optional alphabet, unused. Leave as None.
- trim - should the sequences be trimmed?

The resulting SeqRecord objects should match those from a paired FASTA and QUAL file converted from the SFF file using the Roche 454 tool ssfinfo. i.e. The sequence will be mixed case, with the trim regions shown in lower case.

This function is used internally via the Bio.SeqIO functions:

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("Roche/E3MFGYR02_random_10_reads.sff", "sff"):
...     print("%s %i" % (record.id, len(record)))
...
E3MFGYR02JWQ7T 265
E3MFGYR02JA6IL 271
E3MFGYR02JHD4H 310
E3MFGYR02GFKUC 299
E3MFGYR02FTGED 281
E3MFGYR02FR9G7 261
E3MFGYR02GAZMS 278
E3MFGYR02HHZ8O 221
E3MFGYR02GPGB1 269
E3MFGYR02F7Z7G 219
```

You can also call it directly:

```
>>> with open("Roche/E3MFGYR02_random_10_reads.sff", "rb") as handle:
...     for record in SffIterator(handle):
...         print("%s %i" % (record.id, len(record)))
...
E3MFGYR02JWQ7T 265
E3MFGYR02JA6IL 271
E3MFGYR02JHD4H 310
E3MFGYR02GFKUC 299
E3MFGYR02FTGED 281
```

(continues on next page)

(continued from previous page)

```
E3MFGYR02FR9G7 261
E3MFGYR02GAZMS 278
E3MFGYR02HHZ8O 221
E3MFGYR02GPGB1 269
E3MFGYR02F7Z7G 219
```

Or, with the trim option:

```
>>> with open("Roche/E3MFGYR02_random_10_reads.sff", "rb") as handle:
...     for record in SffIterator(handle, trim=True):
...         print("%s %i" % (record.id, len(record)))
...
E3MFGYR02JWQ7T 260
E3MFGYR02JA6IL 265
E3MFGYR02JHD4H 292
E3MFGYR02GFKUC 295
E3MFGYR02FTGED 277
E3MFGYR02FR9G7 256
E3MFGYR02GAZMS 271
E3MFGYR02HHZ8O 150
E3MFGYR02GPGB1 221
E3MFGYR02F7Z7G 130
```

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

iterate(*handle*)

Parse the file and generate SeqRecord objects.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.SffIO.**SffWriter**(*target*, *index=True*, *xml=None*)

Bases: *SequenceWriter*

SFF file writer.

__init__(*target*, *index=True*, *xml=None*)

Initialize an SFF writer object.

Arguments:

- target - Output stream opened in binary mode, or a path to a file.
- index - Boolean argument, should we try and write an index?
- xml - Optional string argument, xml manifest to be recorded in the index block (see function ReadRocheXmlManifest for reading this data).

write_file(*records*)

Use this to write an entire file containing the given records.

write_header()

Write the SFF file header.

write_record(*record*)

Write a single additional record to the output file.

This assumes the header has been done.

__annotations__ = {}

Bio.SeqIO.SnapGeneIO module

Bio.SeqIO support for the SnapGene file format.

The SnapGene binary format is the native format used by the SnapGene program from GSL Biotech LLC.

class Bio.SeqIO.SnapGeneIO.**SnapGeneIterator**(*source*)

Bases: *SequenceIterator*

Parser for SnapGene files.

__init__(*source*)

Parse a SnapGene file and return a SeqRecord object.

Argument source is a file-like object or a path to a file.

Note that a SnapGene file can only contain one sequence, so this iterator will always return a single record.

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

iterate(*handle*)

Iterate over the records in the SnapGene file.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

Bio.SeqIO.SwissIO module

Bio.SeqIO support for the "swiss" (aka SwissProt/UniProt) file format.

You are expected to use this module via the Bio.SeqIO functions. See also the Bio.SwissProt module which offers more than just accessing the sequences as SeqRecord objects.

See also Bio.SeqIO.UniprotIO.py which supports the "uniprot-xml" format.

Bio.SeqIO.SwissIO.**SwissIterator**(*source*)

Break up a Swiss-Prot/UniProt file into SeqRecord objects.

Argument source is a file-like object or a path to a file.

Every section from the ID line to the terminating // becomes a single SeqRecord with associated annotation and features.

This parser is for the flat file "swiss" format as used by:

- Swiss-Prot aka SwissProt
- TrEMBL
- UniProtKB aka UniProt Knowledgebase

For consistency with BioPerl and EMBOSS we call this the "swiss" format. See also the SeqIO support for "uniprot-xml" format.

Rather than calling it directly, you are expected to use this parser via Bio.SeqIO.parse(..., format="swiss") instead.

Bio.SeqIO.TabIO module

Bio.SeqIO support for the "tab" (simple tab separated) file format.

You are expected to use this module via the Bio.SeqIO functions.

The "tab" format is an ad-hoc plain text file format where each sequence is on one (long) line. Each line contains the identifier/description, followed by a tab, followed by the sequence. For example, consider the following short FASTA format file:

```
>ID123456 possible binding site?
CATCNAGATGACACTACGACTACGACTCAGACTAC
>ID123457 random sequence
ACACTACGACTACGACTCAGACTACAAN
```

Apart from the descriptions, this can be represented in the simple two column tab separated format as follows:

```
ID123456(tab)CATCNAGATGACACTACGACTACGACTCAGACTAC
ID123457(tab)ACACTACGACTACGACTCAGACTACAAN
```

When reading this file, "ID123456" or "ID123457" will be taken as the record's .id and .name property. There is no other information to record.

Similarly, when writing to this format, Biopython will ONLY record the record's .id and .seq (and not the description or any other information) as in the example above.

class Bio.SeqIO.TabIO.**TabIterator**(*source*)

Bases: *SequenceIterator*

Parser for tab-delimited files.

__init__(*source*)

Iterate over tab separated lines as SeqRecord objects.

Each line of the file should contain one tab only, dividing the line into an identifier and the full sequence.

Arguments:

- source - file-like object opened in text mode, or a path to a file

The first field is taken as the record's .id and .name (regardless of any spaces within the text) and the second field is the sequence.

Any blank lines are ignored.

Examples

```
>>> with open("GenBank/NC_005816.tsv") as handle:
...     for record in TabIterator(handle):
...         print("%s length %i" % (record.id, len(record)))
gi|45478712|ref|NP_995567.1| length 340
gi|45478713|ref|NP_995568.1| length 260
gi|45478714|ref|NP_995569.1| length 64
gi|45478715|ref|NP_995570.1| length 123
gi|45478716|ref|NP_995571.1| length 145
gi|45478717|ref|NP_995572.1| length 357
gi|45478718|ref|NP_995573.1| length 138
gi|45478719|ref|NP_995574.1| length 312
gi|45478720|ref|NP_995575.1| length 99
gi|45478721|ref|NP_995576.1| length 90
```

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

iterate(*handle*)

Parse the file and generate SeqRecord objects.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.TabIO.**TabWriter**(*target: IO | PathLike | str | bytes*, *mode: str = 'w'*)

Bases: *SequenceWriter*

Class to write simple tab separated format files.

Each line consists of "id(tab)sequence" only.

Any description, name or other annotation is not recorded.

This class is not intended to be used directly. Instead, please use the function `as_tab`, or the top level `Bio.SeqIO.write()` function with `format="tab"`.

write_record(*record*)

Write a single tab line to the file.

__annotations__ = {}

Bio.SeqIO.TabIO.**as_tab**(*record*)

Return record as tab separated (id(tab)seq) string.

Bio.SeqIO.TwoBitIO module

Bio.SeqIO support for UCSC's "twoBit" (.2bit) file format.

This parser reads the index stored in the twoBit file, as well as the masked regions and the N's for each sequence. It also creates sequence data objects (_TwoBitSequenceData objects), which support only two methods: __len__ and __getitem__. The former will return the length of the sequence, while the latter returns the sequence (as a bytes object) for the requested region.

Using the information in the index, the __getitem__ method calculates the file position at which the requested region starts, and only reads the requested sequence region. Note that the full sequence of a record is loaded only if specifically requested, making the parser memory-efficient.

The TwoBitIterator object implements the __getitem__, keys, and __len__ methods that allow it to be used as a dictionary.

class Bio.SeqIO.TwoBitIO.**TwoBitIterator**(*source*)

Bases: *SequenceIterator*

Parser for UCSC twoBit (.2bit) files.

__init__(*source*)

Read the file index.

parse(*stream*)

Iterate over the sequences in the file.

__getitem__(*name*)

Return sequence associated with given name as a SeqRecord object.

keys()

Return a list with the names of the sequences in the file.

__len__()

Return number of sequences.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

Bio.SeqIO.UniprotIO module

Bio.SeqIO support for the "uniprot-xml" file format.

See Also: http://www.uniprot.org

The UniProt XML format essentially replaces the old plain text file format originally introduced by SwissProt ("swiss" format in Bio.SeqIO).

Bio.SeqIO.UniprotIO.**UniprotIterator**(*source*, *alphabet=None*, *return_raw_comments=False*)

Iterate over UniProt XML as SeqRecord objects.

parses an XML entry at a time from any UniProt XML file returns a SeqRecord for each iteration

This generator can be used in Bio.SeqIO

Argument source is a file-like object or a path to a file.

Optional argument alphabet should not be used anymore.

return_raw_comments = True –> comment fields are returned as complete XML to allow further processing skip_parsing_errors = True –> if parsing errors are found, skip to next entry

class Bio.SeqIO.UniprotIO.**Parser**(*elem*, *alphabet=None*, *return_raw_comments=False*)

Bases: object

Parse a UniProt XML entry to a SeqRecord.

Optional argument alphabet is no longer used.

return_raw_comments=True to get back the complete comment field in XML format

__init__(*elem*, *alphabet=None*, *return_raw_comments=False*)

Initialize the class.

parse()

Parse the input.

Bio.SeqIO.XdnaIO module

Bio.SeqIO support for the "xdna" file format.

The Xdna binary format is generated by Christian Marck's DNA Strider program and also used by Serial Cloner.

class Bio.SeqIO.XdnaIO.**XdnaIterator**(*source*)

Bases: *SequenceIterator*

Parser for Xdna files.

__init__(*source*)

Parse a Xdna file and return a SeqRecord object.

Argument source is a file-like object in binary mode or a path to a file.

Note that this is an "iterator" in name only since an Xdna file always contain a single sequence.

parse(*handle*)

Start parsing the file, and return a SeqRecord generator.

iterate(*handle*, *header*)

Parse the file and generate SeqRecord objects.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__parameters__ = ()

class Bio.SeqIO.XdnaIO.**XdnaWriter**(*target*)

Bases: *SequenceWriter*

Write files in the Xdna format.

__init__(*target*)

Initialize an Xdna writer object.

Arguments:

- target - Output stream opened in binary mode, or a path to a file.

write_file(*records*)

Write the specified record to a Xdna file.

Note that the function expects a list (or iterable) of records as per the SequenceWriter interface, but the list should contain only one record as the Xdna format is a mono-record format.

__annotations__ = {}

Module contents

Sequence input/output as SeqRecord objects.

Bio.SeqIO is also documented at SeqIO and by a whole chapter in our tutorial:

- HTML Tutorial
- PDF Tutorial

Input

The main function is Bio.SeqIO.parse(...) which takes an input file handle (or in recent versions of Biopython alternatively a filename as a string), and format string. This returns an iterator giving SeqRecord objects:

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("Fasta/f002", "fasta"):
...     print("%s %i" % (record.id, len(record)))
gi|1348912|gb|G26680|G26680 633
gi|1348917|gb|G26685|G26685 413
gi|1592936|gb|G29385|G29385 471
```

Note that the parse() function will invoke the relevant parser for the format with its default settings. You may want more control, in which case you need to create a format specific sequence iterator directly.

Some of these parsers are wrappers around low-level parsers which build up SeqRecord objects for the consistent SeqIO interface. In cases where the run-time is critical, such as large FASTA or FASTQ files, calling these underlying parsers will be much faster - in this case these generator functions which return tuples of strings:

```
>>> from Bio.SeqIO.FastaIO import SimpleFastaParser
>>> from Bio.SeqIO.QualityIO import FastqGeneralIterator
```

Input - Single Records

If you expect your file to contain one-and-only-one record, then we provide the following 'helper' function which will return a single SeqRecord, or raise an exception if there are no records or more than one record:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("Fasta/f001", "fasta")
>>> print("%s %i" % (record.id, len(record)))
gi|3318709|pdb|1A91| 79
```

This style is useful when you expect a single record only (and would consider multiple records an error). For example, when dealing with GenBank files for bacterial genomes or chromosomes, there is normally only a single record. Alternatively, use this with a handle when downloading a single record from the internet.

However, if you just want the first record from a file containing multiple record, use the next() function on the iterator:

```
>>> from Bio import SeqIO
>>> record = next(SeqIO.parse("Fasta/f002", "fasta"))
>>> print("%s %i" % (record.id, len(record)))
gi|1348912|gb|G26680|G26680 633
```

The above code will work as long as the file contains at least one record. Note that if there is more than one record, the remaining records will be silently ignored.

Input - Multiple Records

For non-interlaced files (e.g. Fasta, GenBank, EMBL) with multiple records using a sequence iterator can save you a lot of memory (RAM). There is less benefit for interlaced file formats (e.g. most multiple alignment file formats). However, an iterator only lets you access the records one by one.

If you want random access to the records by number, turn this into a list:

```
>>> from Bio import SeqIO
>>> records = list(SeqIO.parse("Fasta/f002", "fasta"))
>>> len(records)
3
>>> print(records[1].id)
gi|1348917|gb|G26685|G26685
```

If you want random access to the records by a key such as the record id, turn the iterator into a dictionary:

```
>>> from Bio import SeqIO
>>> record_dict = SeqIO.to_dict(SeqIO.parse("Fasta/f002", "fasta"))
>>> len(record_dict)
3
>>> print(len(record_dict["gi|1348917|gb|G26685|G26685"]))
413
```

However, using list() or the to_dict() function will load all the records into memory at once, and therefore is not possible on very large files. Instead, for *some* file formats Bio.SeqIO provides an indexing approach providing dictionary like access to any record. For example,

```
>>> from Bio import SeqIO
>>> record_dict = SeqIO.index("Fasta/f002", "fasta")
>>> len(record_dict)
3
>>> print(len(record_dict["gi|1348917|gb|G26685|G26685"]))
413
>>> record_dict.close()
```

Many but not all of the supported input file formats can be indexed like this. For example “fasta”, “fastq”, “qual” and even the binary format “sff” work, but alignment formats like “phylip”, “clustalw” and “nexus” will not.

In most cases you can also use SeqIO.index to get the record from the file as a raw string (not a SeqRecord). This can be useful for example to extract a sub-set of records from a file where SeqIO cannot output the file format (e.g. the plain text SwissProt format, “swiss”) or where it is important to keep the output 100% identical to the input). For example,

```
>>> from Bio import SeqIO
>>> record_dict = SeqIO.index("Fasta/f002", "fasta")
```

(continues on next page)

(continued from previous page)

```
>>> len(record_dict)
3
>>> print(record_dict.get_raw("gi|1348917|gb|G26685|G26685").decode())
>gi|1348917|gb|G26685|G26685 human STS STS_D11734.
CGGAGCCAGCGAGCATATGCTGCATGAGGACCTTTCTATCTTACATTATGGCTGGGAATCTTACTCTTTC
ATCTGATACCTTGTTCAGATTTCAAAATAGTTGTAGCCTTATCCTGGTTTTACAGATGTGAAACTTTCAA
GAGATTTACTGACTTTCCTAGAATAGTTTCTCTACTGGAAACCTGATGCTTTTATAAGCCATTGTGATTA
GGATGACTGTTACAGGCTTAGCTTTGTGTGAAANCCAGTCACCTTTCTCCTAGGTAATGAGTAGTGCTGT
TCATATTACTNTAAGTTCTATAGCATACTTGCNATCCTTTANCCATGCTTATCATANGTACCATTTGAGG
AATTGNTTTGCCCTTTTGGGTTTNTTNTTGGTAAANNNTTCCCGGGTGGGGGNGGTNNNGAAA

>>> print(record_dict["gi|1348917|gb|G26685|G26685"].format("fasta"))
>gi|1348917|gb|G26685|G26685 human STS STS_D11734.
CGGAGCCAGCGAGCATATGCTGCATGAGGACCTTTCTATCTTACATTATGGCTGGGAATC
TTACTCTTTCATCTGATACCTTGTTCAGATTTCAAAATAGTTGTAGCCTTATCCTGGTTT
TACAGATGTGAAACTTTCAAGAGATTTACTGACTTTCCTAGAATAGTTTCTCTACTGGAA
ACCTGATGCTTTTATAAGCCATTGTGATTAGGATGACTGTTACAGGCTTAGCTTTGTGTG
AAANCCAGTCACCTTTCTCCTAGGTAATGAGTAGTGCTGTTCATATTACTNTAAGTTCTA
TAGCATACTTGCNATCCTTTANCCATGCTTATCATANGTACCATTTGAGGAATTGNTTTG
CCCTTTTGGGTTTNTTNTTGGTAAANNNTTCCCGGGTGGGGGNGGTNNNGAAA

>>> record_dict.close()
```

Here the original file and what Biopython would output differ in the line wrapping. Also note that the get_raw method will return a bytes object, hence the use of decode to turn it into a string.

Also note that the get_raw method will preserve the newline endings. This example FASTQ file uses Unix style endings (b"n" only),

```
>>> from Bio import SeqIO
>>> fastq_dict = SeqIO.index("Quality/example.fastq", "fastq")
>>> len(fastq_dict)
3
>>> raw = fastq_dict.get_raw("EAS54_6_R1_2_1_540_792")
>>> raw.count(b"\n")
4
>>> raw.count(b"\r\n")
0
>>> b"\r" in raw
False
>>> len(raw)
78
>>> fastq_dict.close()
```

Here is the same file but using DOS/Windows new lines (b"rn" instead),

```
>>> from Bio import SeqIO
>>> fastq_dict = SeqIO.index("Quality/example_dos.fastq", "fastq")
>>> len(fastq_dict)
3
>>> raw = fastq_dict.get_raw("EAS54_6_R1_2_1_540_792")
>>> raw.count(b"\n")
4
```

(continues on next page)

(continued from previous page)

```
>>> raw.count(b"\r\n")
4
>>> b"\r\n" in raw
True
>>> len(raw)
82
>>> fastq_dict.close()
```

Because this uses two bytes for each new line, the file is longer than the Unix equivalent with only one byte.

Input - Alignments

You can read in alignment files as alignment objects using Bio.AlignIO. Alternatively, reading in an alignment file format via Bio.SeqIO will give you a SeqRecord for each row of each alignment:

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("Clustalw/hedgehog.aln", "clustal"):
...     print("%s %i" % (record.id, len(record)))
gi|167877390|gb|EDS40773.1| 447
gi|167234445|ref|NP_001107837. 447
gi|74100009|gb|AAZ99217.1| 447
gi|13990994|dbj|BAA33523.2| 447
gi|56122354|gb|AAV74328.1| 447
```

Output

Use the function Bio.SeqIO.write(...), which takes a complete set of SeqRecord objects (either as a list, or an iterator), an output file handle (or in recent versions of Biopython an output filename as a string) and of course the file format:

```
from Bio import SeqIO
records = ...
SeqIO.write(records, "example.faa", "fasta")
```

Or, using a handle:

```
from Bio import SeqIO
records = ...
with open("example.faa", "w") as handle:
  SeqIO.write(records, handle, "fasta")
```

You are expected to call this function once (with all your records) and if using a handle, make sure you close it to flush the data to the hard disk.

Output - Advanced

The effect of calling write() multiple times on a single file will vary depending on the file format, and is best avoided unless you have a strong reason to do so.

If you give a filename, then each time you call write() the existing file will be overwritten. For sequential files formats (e.g. fasta, genbank) each "record block" holds a single sequence. For these files it would probably be safe to call write() multiple times by re-using the same handle.

However, trying this for certain alignment formats (e.g. phylip, clustal, stockholm) would have the effect of concatenating several multiple sequence alignments together. Such files are created by the PHYLIP suite of programs for bootstrap analysis, but it is clearer to do this via Bio.AlignIO instead.

Worse, many fileformats have an explicit header and/or footer structure (e.g. any XMl format, and most binary file formats like SFF). Here making multiple calls to write() will result in an invalid file.

Conversion

The Bio.SeqIO.convert(...) function allows an easy interface for simple file format conversions. Additionally, it may use file format specific optimisations so this should be the fastest way too.

In general however, you can combine the Bio.SeqIO.parse(...) function with the Bio.SeqIO.write(...) function for sequence file conversion. Using generator expressions or generator functions provides a memory efficient way to perform filtering or other extra operations as part of the process.

File Formats

When specifying the file format, use lowercase strings. The same format names are also used in Bio.AlignIO and include the following:

- abi - Applied Biosystem's sequencing trace format
- abi-trim - Same as "abi" but with quality trimming with Mott's algorithm
- ace - Reads the contig sequences from an ACE assembly file.
- cif-atom - Uses Bio.PDB.MMCIFParser to determine the (partial) protein sequence as it appears in the structure based on the atomic coordinates.
- cif-seqres - Reads a macromolecular Crystallographic Information File (mmCIF) file to determine the complete protein sequence as defined by the _pdbx_poly_seq_scheme records.
- embl - The EMBL flat file format. Uses Bio.GenBank internally.
- fasta - The generic sequence file format where each record starts with an identifier line starting with a ">" character, followed by lines of sequence.
- fasta-2line - Stricter interpretation of the FASTA format using exactly two lines per record (no line wrapping).
- fastq - A "FASTA like" format used by Sanger which also stores PHRED sequence quality values (with an ASCII offset of 33).
- fastq-sanger - An alias for "fastq" for consistency with BioPerl and EMBOSS
- fastq-solexa - Original Solexa/Illumnia variant of the FASTQ format which encodes Solexa quality scores (not PHRED quality scores) with an ASCII offset of 64.
- fastq-illumina - Solexa/Illumina 1.3 to 1.7 variant of the FASTQ format which encodes PHRED quality scores with an ASCII offset of 64 (not 33). Note as of version 1.8 of the CASAVA pipeline Illumina will produce FASTQ files using the standard Sanger encoding.

- gck - Gene Construction Kit's format.
- genbank - The GenBank or GenPept flat file format.
- gb - An alias for "genbank", for consistency with NCBI Entrez Utilities
- gfa1 - Graphical Fragment Assemblyv versions 1.x. Only segment lines are parsed and all linkage information is ignored.
- gfa2 - Graphical Fragment Assembly version 2.0. Only segment lines are parsed and all linkage information is ignored.
- ig - The IntelliGenetics file format, apparently the same as the MASE alignment format.
- imgt - An EMBL like format from IMGT where the feature tables are more indented to allow for longer feature types.
- nib - UCSC's nib file format for nucleotide sequences, which uses one nibble (4 bits) to represent each nucleotide, and stores two nucleotides in one byte.
- pdb-seqres - Reads a Protein Data Bank (PDB) file to determine the complete protein sequence as it appears in the header (no dependencies).
- pdb-atom - Uses Bio.PDB to determine the (partial) protein sequence as it appears in the structure based on the atom coordinate section of the file (requires NumPy for Bio.PDB).
- phd - Output from PHRED, used by PHRAP and CONSED for input.
- pir - A "FASTA like" format introduced by the National Biomedical Research Foundation (NBRF) for the Protein Information Resource (PIR) database, now part of UniProt.
- seqxml - SeqXML, simple XML format described in Schmitt et al (2011).
- sff - Standard Flowgram Format (SFF), typical output from Roche 454.
- sff-trim - Standard Flowgram Format (SFF) with given trimming applied.
- snapgene - SnapGene's native format.
- swiss - Plain text Swiss-Prot aka UniProt format.
- tab - Simple two column tab separated sequence files, where each line holds a record's identifier and sequence. For example, this is used as by Aligent's eArray software when saving microarray probes in a minimal tab delimited text file.
- qual - A "FASTA like" format holding PHRED quality values from sequencing DNA, but no actual sequences (usually provided in separate FASTA files).
- uniprot-xml - The UniProt XML format (replacement for the SwissProt plain text format which we call "swiss")
- xdna - DNA Strider's and SerialCloner's native format.

Note that while Bio.SeqIO can read all the above file formats, it cannot write to all of them.

You can also use any file format supported by Bio.AlignIO, such as "nexus", "phylip" and "stockholm", which gives you access to the individual sequences making up each alignment as SeqRecords.

Bio.SeqIO.**write**(*sequences: Iterable[*SeqRecord*]* | SeqRecord, *handle: IO[str]* | *PathLike* | *str* | *bytes*, *format: str*) → int

Write complete set of sequences to a file.

Arguments:

- sequences - A list (or iterator) of SeqRecord objects, or a single SeqRecord.
- handle - File handle object to write to, or filename as string.

- format - lower case string describing the file format to write.

Note if providing a file handle, your code should close the handle after calling this function (to ensure the data gets flushed to disk).

Returns the number of records written (as an integer).

Bio.SeqIO.**parse**(*handle*, *format*, *alphabet=None*)

Turn a sequence file into an iterator returning SeqRecords.

Arguments:

- handle - handle to the file, or the filename as a string (note older versions of Biopython only took a handle).
- format - lower case string describing the file format.
- alphabet - no longer used, should be None.

Typical usage, opening a file to read in, and looping over the record(s):

```
>>> from Bio import SeqIO
>>> filename = "Fasta/sweetpea.nu"
>>> for record in SeqIO.parse(filename, "fasta"):
...     print("ID %s" % record.id)
...     print("Sequence length %i" % len(record))
ID gi|3176602|gb|U78617.1|LOU78617
Sequence length 309
```

For lazy-loading file formats such as twobit, for which the file contents is read on demand only, ensure that the file remains open while extracting sequence data.

If you have a string 'data' containing the file contents, you must first turn this into a handle in order to parse it:

```
>>> data = ">Alpha\nACCGGATGTA\n>Beta\nAGGCTCGGTTA\n"
>>> from Bio import SeqIO
>>> from io import StringIO
>>> for record in SeqIO.parse(StringIO(data), "fasta"):
...     print("%s %s" % (record.id, record.seq))
Alpha ACCGGATGTA
Beta AGGCTCGGTTA
```

Use the Bio.SeqIO.read(...) function when you expect a single record only.

Bio.SeqIO.**read**(*handle*, *format*, *alphabet=None*)

Turn a sequence file into a single SeqRecord.

Arguments:

- handle - handle to the file, or the filename as a string (note older versions of Biopython only took a handle).
- format - string describing the file format.
- alphabet - no longer used, should be None.

This function is for use parsing sequence files containing exactly one record. For example, reading a GenBank file:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("GenBank/arab1.gb", "genbank")
>>> print("ID %s" % record.id)
ID AC007323.5
>>> print("Sequence length %i" % len(record))
Sequence length 86436
```

If the handle contains no records, or more than one record, an exception is raised. For example:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("GenBank/cor6_6.gb", "genbank")
Traceback (most recent call last):
    ...
ValueError: More than one record found in handle
```

If however you want the first record from a file containing multiple records this function would raise an exception (as shown in the example above). Instead use:

```
>>> from Bio import SeqIO
>>> record = next(SeqIO.parse("GenBank/cor6_6.gb", "genbank"))
>>> print("First record's ID %s" % record.id)
First record's ID X55053.1
```

Use the Bio.SeqIO.parse(handle, format) function if you want to read multiple records from the handle.

Bio.SeqIO.**to_dict**(*sequences*, *key_function=None*)

Turn a sequence iterator or list into a dictionary.

Arguments:

- sequences - An iterator that returns SeqRecord objects, or simply a list of SeqRecord objects.
- key_function - Optional callback function which when given a SeqRecord should return a unique key for the dictionary.

e.g. key_function = lambda rec : rec.name or, key_function = lambda rec : rec.description.split()[0]

If key_function is omitted then record.id is used, on the assumption that the records objects returned are SeqRecords with a unique id.

If there are duplicate keys, an error is raised.

Since Python 3.7, the default dict class maintains key order, meaning this dictionary will reflect the order of records given to it. For CPython and PyPy, this was already implemented for Python 3.6, so effectively you can always assume the record order is preserved.

Example usage, defaulting to using the record.id as key:

```
>>> from Bio import SeqIO
>>> filename = "GenBank/cor6_6.gb"
>>> format = "genbank"
>>> id_dict = SeqIO.to_dict(SeqIO.parse(filename, format))
>>> print(list(id_dict))
['X55053.1', 'X62281.1', 'M81224.1', 'AJ237582.1', 'L31939.1', 'AF297471.1']
>>> print(id_dict["L31939.1"].description)
Brassica rapa (clone bif72) kin mRNA, complete cds
```

A more complex example, using the key_function argument in order to use a sequence checksum as the dictionary key:

```
>>> from Bio import SeqIO
>>> from Bio.SeqUtils.CheckSum import seguid
>>> filename = "GenBank/cor6_6.gb"
>>> format = "genbank"
>>> seguid_dict = SeqIO.to_dict(SeqIO.parse(filename, format),
...                             key_function = lambda rec : seguid(rec.seq))
>>> for key, record in sorted(seguid_dict.items()):
...     print("%s %s" % (key, record.id))
/wQvmrl87QWcm9llO4/efg23Vgg AJ237582.1
BUg6YxXSKWEcFFH0L08JzaLGhQs L31939.1
SabZaA4V2eLE9/2Fm5FnyYy07J4 X55053.1
TtWsXo45S3ZclIBy4X/WJc39+CY M81224.1
l7gjJFE6W/S1jJn5+1ASrUKW/FA X62281.1
uVEYeAQSV5EDQOnFoeMmVea+Oow AF297471.1
```

This approach is not suitable for very large sets of sequences, as all the SeqRecord objects are held in memory. Instead, consider using the Bio.SeqIO.index() function (if it supports your particular file format).

This dictionary will reflect the order of records given to it.

Bio.SeqIO.**index**(*filename*, *format*, *alphabet=None*, *key_function=None*)

Indexes a sequence file and returns a dictionary like object.

Arguments:

- filename - string giving name of file to be indexed
- format - lower case string describing the file format
- alphabet - no longer used, leave as None
- key_function - Optional callback function which when given a SeqRecord identifier string should return a unique key for the dictionary.

This indexing function will return a dictionary like object, giving the SeqRecord objects as values.

As of Biopython 1.69, this will preserve the ordering of the records in file when iterating over the entries.

```
>>> from Bio import SeqIO
>>> records = SeqIO.index("Quality/example.fastq", "fastq")
>>> len(records)
3
>>> list(records)  # make a list of the keys
['EAS54_6_R1_2_1_413_324', 'EAS54_6_R1_2_1_540_792', 'EAS54_6_R1_2_1_443_348']
>>> print(records["EAS54_6_R1_2_1_540_792"].format("fasta"))
>EAS54_6_R1_2_1_540_792
TTGGCAGGCCAAGGCCGATGGATCA

>>> "EAS54_6_R1_2_1_540_792" in records
True
>>> print(records.get("Missing", None))
None
>>> records.close()
```

If the file is BGZF compressed, this is detected automatically. Ordinary GZIP files are not supported:

```
>>> from Bio import SeqIO
>>> records = SeqIO.index("Quality/example.fastq.bgz", "fastq")
>>> len(records)
3
>>> print(records["EAS54_6_R1_2_1_540_792"].seq)
TTGGCAGGCCAAGGCCGATGGATCA
>>> records.close()
```

When you call the index function, it will scan through the file, noting the location of each record. When you access a particular record via the dictionary methods, the code will jump to the appropriate part of the file and then parse that section into a SeqRecord.

Note that not all the input formats supported by Bio.SeqIO can be used with this index function. It is designed to work only with sequential file formats (e.g. "fasta", "gb", "fastq") and is not suitable for any interlaced file format (e.g. alignment formats such as "clustal").

For small files, it may be more efficient to use an in memory Python dictionary, e.g.

```
>>> from Bio import SeqIO
>>> records = SeqIO.to_dict(SeqIO.parse("Quality/example.fastq", "fastq"))
>>> len(records)
3
>>> list(records)  # make a list of the keys
['EAS54_6_R1_2_1_413_324', 'EAS54_6_R1_2_1_540_792', 'EAS54_6_R1_2_1_443_348']
>>> print(records["EAS54_6_R1_2_1_540_792"].format("fasta"))
>EAS54_6_R1_2_1_540_792
TTGGCAGGCCAAGGCCGATGGATCA
```

As with the to_dict() function, by default the id string of each record is used as the key. You can specify a callback function to transform this (the record identifier string) into your preferred key. For example:

```
>>> from Bio import SeqIO
>>> def make_tuple(identifier):
...     parts = identifier.split("_")
...     return int(parts[-2]), int(parts[-1])
>>> records = SeqIO.index("Quality/example.fastq", "fastq",
...                       key_function=make_tuple)
>>> len(records)
3
>>> list(records)  # make a list of the keys
[(413, 324), (540, 792), (443, 348)]
>>> print(records[(540, 792)].format("fasta"))
>EAS54_6_R1_2_1_540_792
TTGGCAGGCCAAGGCCGATGGATCA

>>> (540, 792) in records
True
>>> "EAS54_6_R1_2_1_540_792" in records
False
>>> print(records.get("Missing", None))
None
>>> records.close()
```

Another common use case would be indexing an NCBI style FASTA file, where you might want to extract the GI number from the FASTA identifier to use as the dictionary key.

Notice that unlike the to_dict() function, here the key_function does not get given the full SeqRecord to use to generate the key. Doing so would impose a severe performance penalty as it would require the file to be completely parsed while building the index. Right now this is usually avoided.

See Also: Bio.SeqIO.index_db() and Bio.SeqIO.to_dict()

Bio.SeqIO.**index_db**(*index_filename*, *filenames=None*, *format=None*, *alphabet=None*, *key_function=None*)

Index several sequence files and return a dictionary like object.

The index is stored in an SQLite database rather than in memory (as in the Bio.SeqIO.index(...) function).

Arguments:

- index_filename - Where to store the SQLite index
- filenames - list of strings specifying file(s) to be indexed, or when indexing a single file this can be given as a string. (optional if reloading an existing index, but must match)
- format - lower case string describing the file format (optional if reloading an existing index, but must match)
- alphabet - no longer used, leave as None.
- key_function - Optional callback function which when given a SeqRecord identifier string should return a unique key for the dictionary.

This indexing function will return a dictionary like object, giving the SeqRecord objects as values:

```
>>> from Bio import SeqIO
>>> files = ["GenBank/NC_000932.faa", "GenBank/NC_005816.faa"]
>>> def get_gi(name):
...     parts = name.split("|")
...     i = parts.index("gi")
...     assert i != -1
...     return parts[i+1]
>>> idx_name = ":memory:" #use an in memory SQLite DB for this test
>>> records = SeqIO.index_db(idx_name, files, "fasta", key_function=get_gi)
>>> len(records)
95
>>> records["7525076"].description
'gi|7525076|ref|NP_051101.1| Ycf2 [Arabidopsis thaliana]'
>>> records["45478717"].description
'gi|45478717|ref|NP_995572.1| pesticin [Yersinia pestis biovar Microtus str. 91001]'
>>> records.close()
```

In this example the two files contain 85 and 10 records respectively.

BGZF compressed files are supported, and detected automatically. Ordinary GZIP compressed files are not supported.

See Also: Bio.SeqIO.index() and Bio.SeqIO.to_dict(), and the Python module glob which is useful for building lists of files.

Bio.SeqIO.**convert**(*in_file*, *in_format*, *out_file*, *out_format*, *molecule_type=None*)

Convert between two sequence file formats, return number of records.

Arguments:

- in_file - an input handle or filename
- in_format - input file format, lower case string

- out_file - an output handle or filename
- out_format - output file format, lower case string
- molecule_type - optional molecule type to apply, string containing "DNA", "RNA" or "protein".

NOTE - If you provide an output filename, it will be opened which will overwrite any existing file without warning.

The idea here is that while doing this will work:

```
from Bio import SeqIO
records = SeqIO.parse(in_handle, in_format)
count = SeqIO.write(records, out_handle, out_format)
```

it is shorter to write:

```
from Bio import SeqIO
count = SeqIO.convert(in_handle, in_format, out_handle, out_format)
```

Also, Bio.SeqIO.convert is faster for some conversions as it can make some optimisations.

For example, going from a filename to a handle:

```
>>> from Bio import SeqIO
>>> from io import StringIO
>>> handle = StringIO("")
>>> SeqIO.convert("Quality/example.fastq", "fastq", handle, "fasta")
3
>>> print(handle.getvalue())
>EAS54_6_R1_2_1_413_324
CCCTTCTTGTCTTCAGCGTTTCTCC
>EAS54_6_R1_2_1_540_792
TTGGCAGGCCAAGGCCGATGGATCA
>EAS54_6_R1_2_1_443_348
GTTGCTTCTGGCGTGGGTGGGGGGG
```

Note some formats like SeqXML require you to specify the molecule type when it cannot be determined by the parser:

```
>>> from Bio import SeqIO
>>> from io import BytesIO
>>> handle = BytesIO()
>>> SeqIO.convert("Quality/example.fastq", "fastq", handle, "seqxml", "DNA")
3
```

28.1.30 Bio.SeqUtils package

Submodules

Bio.SeqUtils.CheckSum module

Functions to calculate assorted sequence checksums.

Bio.SeqUtils.CheckSum.**crc32**(*seq*)

Return the crc32 checksum for a sequence (string or Seq object).

Note that the case is important:

```
>>> crc32("ACGTACGTACGT")
20049947
>>> crc32("acgtACGTacgt")
1688586483
```

Bio.SeqUtils.CheckSum.**crc64**(*s*)

Return the crc64 checksum for a sequence (string or Seq object).

Note that the case is important:

```
>>> crc64("ACGTACGTACGT")
'CRC-C4FBB762C4A87EBD'
>>> crc64("acgtACGTacgt")
'CRC-DA4509DC64A87EBD'
```

Bio.SeqUtils.CheckSum.**gcg**(*seq*)

Return the GCG checksum (int) for a sequence (string or Seq object).

Given a nucleotide or amino-acid sequence (or any string), returns the GCG checksum (int). Checksum used by GCG program. seq type = str.

Based on BioPerl GCG_checksum. Adapted by Sebastian Bassi with the help of John Lenton, Pablo Ziliani, and Gabriel Genellina.

All sequences are converted to uppercase.

```
>>> gcg("ACGTACGTACGT")
5688
>>> gcg("acgtACGTacgt")
5688
```

Bio.SeqUtils.CheckSum.**seguid**(*seq*)

Return the SEGUID (string) for a sequence (string or Seq object).

Given a nucleotide or amino-acid sequence (or any string), returns the SEGUID string (A SEquence Globally Unique IDentifier). seq type = str.

Note that the case is not important:

```
>>> seguid("ACGTACGTACGT")
'If6HIvcnRSQDVNiAoefAzySc6i4'
>>> seguid("acgtACGTacgt")
'If6HIvcnRSQDVNiAoefAzySc6i4'
```

For more information about SEGUID, see: http://bioinformatics.anl.gov/seguid/ https://doi.org/10.1002/pmic.200600032

Bio.SeqUtils.IsoelectricPoint module

Calculate isoelectric points of polypeptides using methods of Bjellqvist.

pK values and the methods are taken from:

```
* Bjellqvist, B.,Hughes, G.J., Pasquali, Ch., Paquet, N., Ravier, F.,
Sanchez, J.-Ch., Frutiger, S. & Hochstrasser, D.F.
The focusing positions of polypeptides in immobilized pH gradients can be
predicted from their amino acid sequences. Electrophoresis 1993, 14,
1023-1031.

* Bjellqvist, B., Basse, B., Olsen, E. and Celis, J.E.
Reference points for comparisons of two-dimensional maps of proteins from
different human cell types defined in a pH scale where isoelectric points
correlate with polypeptide compositions. Electrophoresis 1994, 15, 529-539.
```

I designed the algorithm according to a note by David L. Tabb, available at: http://fields.scripps.edu/DTASelect/20010710-pI-Algorithm.pdf

class Bio.SeqUtils.IsoelectricPoint.**IsoelectricPoint**(*protein_sequence*, *aa_content=None*)

Bases: `object`

A class for calculating the IEP or charge at given pH of a protein.

Parameters

:protein_sequence: A ``Bio.Seq`` or string object containing a protein
sequence.

:aa_content: A dictionary with amino acid letters as keys and its
occurrences as integers, e.g. `{"A": 3, "C": 0, ...}`. Default: `None`. If `None`, the dic will be calculated from the given sequence.

Examples

The methods of this class can either be accessed from the class itself or from a `ProtParam.ProteinAnalysis` object (with partially different names):

```
>>> from Bio.SeqUtils.IsoelectricPoint import IsoelectricPoint as IP
>>> protein = IP("INGAR")
>>> print(f"IEP of peptide {protein.sequence} is {protein.pi():.2f}")
IEP of peptide INGAR is 9.75
>>> print(f"Its charge at pH 7 is {protein.charge_at_pH(7.0):.2f}")
Its charge at pH 7 is 0.76
```

```
>>> from Bio.SeqUtils.ProtParam import ProteinAnalysis as PA
>>> protein = PA("PETER")
>>> print(f"IEP of {protein.sequence}: {protein.isoelectric_point():.2f}")
IEP of PETER: 4.53
>>> print(f"Charge at pH 4.53: {protein.charge_at_pH(4.53):.2f}")
Charge at pH 4.53: 0.00
```

Methods

:charge_at_pH(pH): Calculates the charge of the protein for a given pH
:pi(): Calculates the isoelectric point

__init__(*protein_sequence*, *aa_content=None*)

Initialize the class.

charge_at_pH(*pH*)

Calculate the charge of a protein at given pH.

pi(*pH=7.775*, *min_=4.05*, *max_=12*)

Calculate and return the isoelectric point as float.

This is a recursive function that uses bisection method. Wiki on bisection: https://en.wikipedia.org/wiki/Bisection_method

Arguments:

- pH: the pH at which the current charge of the protein is computed. This pH lies at the centre of the interval (mean of *min_* and *max_*).
- min_: the minimum of the interval. Initial value defaults to 4.05, which is below the theoretical minimum, when the protein is composed exclusively of aspartate.
- max_: the maximum of the the interval. Initial value defaults to 12, which is above the theoretical maximum, when the protein is composed exclusively of arginine.

Bio.SeqUtils.MeltingTemp module

Calculate the melting temperature of nucleotide sequences.

This module contains three different methods to calculate the melting temperature of oligonucleotides:

1. Tm_Wallace: 'Rule of thumb'
2. Tm_GC: Empirical formulas based on GC content. Salt and mismatch corrections can be included.
3. Tm_NN: Calculation based on nearest neighbor thermodynamics. Several tables for DNA/DNA, DNA/RNA and RNA/RNA hybridizations are included. Correction for mismatches, dangling ends, salt concentration and other additives are available.

General parameters for most Tm methods:

- seq – A Biopython sequence object or a string.
- check – Checks if the sequence is valid for the given method (default= True). In general, whitespaces and non-base characters are removed and characters are converted to uppercase. RNA will be backtranscribed.
- strict – Do not allow base characters or neighbor duplex keys (e.g. 'AT/NA') that could not or not unambiguously be evaluated for the respective method (default=True). Note that W (= A or T) and S (= C or G) are not ambiguous for Tm_Wallace and Tm_GC. If 'False', average values (if applicable) will be used.

This module is not able to detect self-complementary and it will not use alignment tools to align an oligonucleotide sequence to its target sequence. Thus it can not detect dangling-ends and mismatches by itself (don't even think about bulbs and loops). These parameters have to be handed over to the respective method.

Other public methods of this module:

- make_table : To create a table with thermodynamic data.

- salt_correction: To adjust Tm to a given salt concentration by different formulas. This method is called from Tm_GC and Tm_NN but may also be accessed 'manually'. It returns a correction term, not a corrected Tm!
- chem_correction: To adjust Tm regarding the chemical additives DMSO and formaldehyde. The method returns a corrected Tm. Chemical correction is not an integral part of the Tm methods and must be called additionally.

For example:

```
>>> from Bio.SeqUtils import MeltingTemp as mt
>>> from Bio.Seq import Seq
>>> mystring = 'CGTTCCAAAGATGTGGGCATGAGCTTAC'
>>> myseq = Seq(mystring)
>>> print('%0.2f' % mt.Tm_Wallace(mystring))
84.00
>>> print('%0.2f' % mt.Tm_Wallace(myseq))
84.00
>>> print('%0.2f' % mt.Tm_GC(myseq))
58.97
>>> print('%0.2f' % mt.Tm_NN(myseq))
60.32
```

Using different thermodynamic tables, e.g. from Breslauer '86 or Sugimoto '96:

```
>>> print('%0.2f' % mt.Tm_NN(myseq, nn_table=mt.DNA_NN1))  # Breslauer '86
72.19
>>> print('%0.2f' % mt.Tm_NN(myseq, nn_table=mt.DNA_NN2))  # Sugimoto '96
65.47
```

Tables for RNA and RNA/DNA hybrids are included:

```
>>> print('%0.2f' % mt.Tm_NN(myseq, nn_table=mt.RNA_NN1))  # Freier '86
73.35
>>> print('%0.2f' % mt.Tm_NN(myseq, nn_table=mt.R_DNA_NN1))  # Sugimoto '95
58.45
```

Several types of salc correction (for Tm_NN and Tm_GC):

```
>>> for i in range(1, 8):
...     print("Type: %d, Tm: %0.2f" % (i, Tm_NN(myseq, saltcorr=i)))
...
Type: 1, Tm: 54.27
Type: 2, Tm: 54.02
Type: 3, Tm: 59.60
Type: 4, Tm: 60.64
Type: 5, Tm: 60.32
Type: 6, Tm: 59.78
Type: 7, Tm: 59.78
```

Correction for other monovalent cations (K+, Tris), Mg2+ and dNTPs according to von Ahsen et al. (2001) or Owczarzy et al. (2008) (for Tm_NN and Tm_GC):

```
>>> print('%0.2f' % mt.Tm_NN(myseq, Na=50, Tris=10))
60.79
```

(continues on next page)

(continued from previous page)

```
>>> print('%0.2f' % mt.Tm_NN(myseq, Na=50, Tris=10, Mg=1.5))
67.39
>>> print('%0.2f' % mt.Tm_NN(myseq, Na=50, Tris=10, Mg=1.5, saltcorr=7))
66.81
>>> print('%0.2f' % mt.Tm_NN(myseq, Na=50, Tris=10, Mg=1.5, dNTPs=0.6,
...                          saltcorr=7))
66.04
```

Dangling ends and mismatches, e.g.:

```
Oligo:     CGTTCCaAAGATGTGGGCATGAGCTTAC       CGTTCCaAAGATGTGGGCATGAGCTTAC
           ::::::X:::::::::::::::::::::  or   ::::::X:::::::::::::::::::::
Template:  GCAAGGcTTCTACACCCGTACTCGAATG      TGCAAGGcTTCTACACCCGTACTCGAATGC
```

Here:

```
>>> print('%0.2f' % mt.Tm_NN('CGTTCCAAAGATGTGGGCATGAGCTTAC'))
60.32
>>> print('%0.2f' % mt.Tm_NN('CGTTCCAAAGATGTGGGCATGAGCTTAC',
...                          c_seq='GCAAGGcTTCTACACCCGTACTCGAATG'))
55.39
>>> print('%0.2f' % mt.Tm_NN('CGTTCCAAAGATGTGGGCATGAGCTTAC', shift=1,
...                          c_seq='TGCAAGGcTTCTACACCCGTACTCGAATGC'))
55.69
```

The same for RNA:

```
>>> print('%0.2f' % mt.Tm_NN('CGUUCCAAAGAUGUGGGCAUGAGCUUAC',
...                          c_seq='UGCAAGGcUUCUACACCCGUACUCGAAUGC',
...                          shift=1, nn_table=mt.RNA_NN3,
...                          de_table=mt.RNA_DE1))
73.00
```

Note, that thermodynamic data are not available for all kind of mismatches, e.g. most double mismatches or terminal mismatches combined with dangling ends:

```
>>> print('%0.2f' % mt.Tm_NN('CGTTCCAAAGATGTGGGCATGAGCTTAC',
...                          c_seq='TtCAAGGcTTCTACACCCGTACTCGAATGC',
...                          shift=1))
Traceback (most recent call last):
ValueError: no thermodynamic data for neighbors '.C/TT' available
```

Make your own tables, or update/extend existing tables. E.g., add values for locked nucleotides. Here, 'locked A' (and its complement) should be represented by '1':

```
>>> mytable = mt.make_table(oldtable=mt.DNA_NN3,
...                         values={'A1/T1':(-6.608, -17.235),
...                         '1A/1T':(-6.893, -15.923)})
>>> print('%0.2f' % mt.Tm_NN('CGTTCCAAAGATGTGGGCATGAGCTTAC'))
60.32
>>> print('%0.2f' % mt.Tm_NN('CGTTCCA1AGATGTGGGCATGAGCTTAC',
...                          nn_table=mytable, check=False))
```

(continues on next page)

(continued from previous page)

```
... # 'check' must be False, otherwise '1' would be discarded
62.53
```

Bio.SeqUtils.MeltingTemp.**make_table**(*oldtable=None*, *values=None*)

Return a table with thermodynamic parameters (as dictionary).

Arguments:

- oldtable: An existing dictionary with thermodynamic parameters.
- values: A dictionary with new or updated values.

E.g., to replace the initiation parameters in the Sugimoto '96 dataset with the initiation parameters from Allawi & SantaLucia '97:

```
>>> from Bio.SeqUtils.MeltingTemp import make_table, DNA_NN2
>>> table = DNA_NN2                                    # Sugimoto '96
>>> table['init_A/T']
(0, 0)
>>> newtable = make_table(oldtable=DNA_NN2, values={'init': (0, 0),
...                       'init_A/T': (2.3, 4.1),
...                       'init_G/C': (0.1, -2.8)})
>>> print("%0.1f, %0.1f" % newtable['init_A/T'])
2.3, 4.1
```

Bio.SeqUtils.MeltingTemp.**salt_correction**(*Na=0*, *K=0*, *Tris=0*, *Mg=0*, *dNTPs=0*, *method=1*, *seq=None*)

Calculate a term to correct Tm for salt ions.

Depending on the Tm calculation, the term will correct Tm or entropy. To calculate corrected Tm values, different operations need to be applied:

- methods 1-4: Tm(new) = Tm(old) + corr
- method 5: deltaS(new) = deltaS(old) + corr
- methods 6+7: Tm(new) = 1/(1/Tm(old) + corr)

Arguments:

- Na, K, Tris, Mg, dNTPS: Millimolar concentration of respective ion. To have a simple 'salt correction', just pass Na. If any of K, Tris, Mg and dNTPS is non-zero, a 'sodium-equivalent' concentration is calculated according to von Ahsen et al. (2001, Clin Chem 47: 1956-1961): [Na_eq] = [Na+] + [K+] + [Tris]/2 + 120*([Mg2+] - [dNTPs])^0.5 If [dNTPs] >= [Mg2+]: [Na_eq] = [Na+] + [K+] + [Tris]/2
- method: Which method to be applied. Methods 1-4 correct Tm, method 5 corrects deltaS, methods 6 and 7 correct 1/Tm. The methods are:
  1. 16.6 x log[Na+] (Schildkraut & Lifson (1965), Biopolymers 3: 195-208)
  2. 16.6 x log([Na+]/(1.0 + 0.7*[Na+])) (Wetmur (1991), Crit Rev Biochem Mol Biol 126: 227-259)
  3. 12.5 x log(Na+] (SantaLucia et al. (1996), Biochemistry 35: 3555-3562
  4. 11.7 x log[Na+] (SantaLucia (1998), Proc Natl Acad Sci USA 95: 1460-1465
  5. Correction for deltaS: 0.368 x (N-1) x ln[Na+] (SantaLucia (1998), Proc Natl Acad Sci USA 95: 1460-1465)
  6. (4.29(%GC)-3.95)x1e-5 x ln[Na+] + 9.40e-6 x ln[Na+]^2 (Owczarzy et al. (2004), Biochemistry 43: 3537-3554)

7. Complex formula with decision tree and 7 empirical constants. Mg2+ is corrected for dNTPs binding (if present) (Owczarzy et al. (2008), Biochemistry 47: 5336-5353)

Examples

```
>>> from Bio.SeqUtils.MeltingTemp import salt_correction
>>> print('%0.2f' % salt_correction(Na=50, method=1))
-21.60
>>> print('%0.2f' % salt_correction(Na=50, method=2))
-21.85
>>> print('%0.2f' % salt_correction(Na=100, Tris=20, method=2))
-16.45
>>> print('%0.2f' % salt_correction(Na=100, Tris=20, Mg=1.5, method=2))
-10.99
```

Bio.SeqUtils.MeltingTemp.**chem_correction**(*melting_temp*, *DMSO=0*, *fmd=0*, *DMSOfactor=0.75*, *fmdfactor=0.65*, *fmdmethod=1*, *GC=None*)

Correct a given Tm for DMSO and formamide.

Please note that these corrections are +/- rough approximations.

Arguments:

- melting_temp: Melting temperature.
- DMSO: Percent DMSO.
- fmd: Formamide concentration in %(fmdmethod=1) or molar (fmdmethod=2).
- DMSOfactor: How much should Tm decreases per percent DMSO. Default=0.65 (von Ahsen et al. 2001). Other published values are 0.5, 0.6 and 0.675.
- fmdfactor: How much should Tm decrease per percent formamide. Default=0.65. Several papers report factors between 0.6 and 0.72.
- fmdmethod:
  1. Tm = Tm - factor(%formamide) (Default)
  2. Tm = Tm + (0.453(f(GC)) - 2.88) x [formamide]

  Here f(GC) is fraction of GC. Note (again) that in fmdmethod=1 formamide concentration is given in %, while in fmdmethod=2 it is given in molar.
- GC: GC content in percent.

Examples:

```
>>> from Bio.SeqUtils import MeltingTemp as mt
>>> mt.chem_correction(70)
70
>>> print('%0.2f' % mt.chem_correction(70, DMSO=3))
67.75
>>> print('%0.2f' % mt.chem_correction(70, fmd=5))
66.75
>>> print('%0.2f' % mt.chem_correction(70, fmdmethod=2, fmd=1.25,
...                                    GC=50))
66.68
```

Bio.SeqUtils.MeltingTemp.**Tm_Wallace**(*seq*, *check=True*, *strict=True*)

Calculate and return the Tm using the 'Wallace rule'.

Tm = 4 degC * (G + C) + 2 degC * (A+T)

The Wallace rule (Thein & Wallace 1986, in Human genetic diseases: a practical approach, 33-50) is often used as rule of thumb for approximate Tm calculations for primers of 14 to 20 nt length.

Non-DNA characters (e.g., E, F, J, !, 1, etc) are ignored by this method.

Examples:

```
>>> from Bio.SeqUtils import MeltingTemp as mt
>>> mt.Tm_Wallace('ACGTTGCAATGCCGTA')
48.0
>>> mt.Tm_Wallace('ACGT TGCA ATGC CGTA')
48.0
>>> mt.Tm_Wallace('1ACGT2TGCA3ATGC4CGTA')
48.0
```

Bio.SeqUtils.MeltingTemp.**Tm_GC**(*seq*, *check=True*, *strict=True*, *valueset=7*, *userset=None*, *Na=50*, *K=0*, *Tris=0*, *Mg=0*, *dNTPs=0*, *saltcorr=0*, *mismatch=True*)

Return the Tm using empirical formulas based on GC content.

General format: Tm = A + B(%GC) - C/N + salt correction - D(%mismatch)

A, B, C, D: empirical constants, N: primer length D (amount of decrease in Tm per % mismatch) is often 1, but sometimes other values have been used (0.6-1.5). Use 'X' to indicate the mismatch position in the sequence. Note that this mismatch correction is a rough estimate.

```
>>> from Bio.SeqUtils import MeltingTemp as mt
>>> print("%0.2f" % mt.Tm_GC('CTGCTGATXGCACGAGGTTATGG', valueset=2))
69.20
```

Arguments:

- valueset: A few often cited variants are included:
  1. Tm = 69.3 + 0.41(%GC) - 650/N (Marmur & Doty 1962, J Mol Biol 5: 109-118; Chester & Marshak 1993), Anal Biochem 209: 284-290)
  2. Tm = 81.5 + 0.41(%GC) - 675/N - %mismatch 'QuikChange' formula. Recommended (by the manufacturer) for the design of primers for QuikChange mutagenesis.
  3. Tm = 81.5 + 0.41(%GC) - 675/N + 16.6 x log[Na+] (Marmur & Doty 1962, J Mol Biol 5: 109-118; Schildkraut & Lifson 1965, Biopolymers 3: 195-208)
  4. Tm = 81.5 + 0.41(%GC) - 500/N + 16.6 x log([Na+]/(1.0 + 0.7 x [Na+])) - %mismatch (Wetmur 1991, Crit Rev Biochem Mol Biol 126: 227-259). This is the standard formula in approximative mode of MELTING 4.3.
  5. Tm = 78 + 0.7(%GC) - 500/N + 16.6 x log([Na+]/(1.0 + 0.7 x [Na+])) - %mismatch (Wetmur 1991, Crit Rev Biochem Mol Biol 126: 227-259). For RNA.
  6. Tm = 67 + 0.8(%GC) - 500/N + 16.6 x log([Na+]/(1.0 + 0.7 x [Na+])) - %mismatch (Wetmur 1991, Crit Rev Biochem Mol Biol 126: 227-259). For RNA/DNA hybrids.
  7. Tm = 81.5 + 0.41(%GC) - 600/N + 16.6 x log[Na+] Used by Primer3Plus to calculate the product Tm. Default set.

8. Tm = 77.1 + 0.41(%GC) - 528/N + 11.7 x log[Na+] (von Ahsen et al. 2001, Clin Chem 47: 1956-1961). Recommended 'as a tradeoff between accuracy and ease of use'.

- userset: Tuple of four values for A, B, C, and D. Usersets override valuesets.
- Na, K, Tris, Mg, dNTPs: Concentration of the respective ions [mM]. If any of K, Tris, Mg and dNTPS is non-zero, a 'sodium-equivalent' concentration is calculated and used for salt correction (von Ahsen et al., 2001).
- saltcorr: Type of salt correction (see method salt_correction). Default=0. 0 or None means no salt correction.
- mismatch: If 'True' (default) every 'X' in the sequence is counted as mismatch.

Bio.SeqUtils.MeltingTemp.**Tm_NN**(*seq*, *check=True*, *strict=True*, *c_seq=None*, *shift=0*, *nn_table=None*, *tmm_table=None*, *imm_table=None*, *de_table=None*, *dnac1=25*, *dnac2=25*, *selfcomp=False*, *Na=50*, *K=0*, *Tris=0*, *Mg=0*, *dNTPs=0*, *saltcorr=5*)

Return the Tm using nearest neighbor thermodynamics.

Arguments:

- seq: The primer/probe sequence as string or Biopython sequence object. For RNA/DNA hybridizations seq must be the RNA sequence.
- c_seq: Complementary sequence. The sequence of the template/target in 3'->5' direction. c_seq is necessary for mismatch correction and dangling-ends correction. Both corrections will automatically be applied if mismatches or dangling ends are present. Default=None.
- shift: Shift of the primer/probe sequence on the template/target sequence, e.g.:

```
                  shift=0       shift=1       shift= -1
Primer (seq):     5' ATGC...    5'  ATGC...    5' ATGC...
Template (c_seq): 3' TACG...    3' CTACG...    3'  ACG...
```

  The shift parameter is necessary to align seq and c_seq if they have different lengths or if they should have dangling ends. Default=0

- table: Thermodynamic NN values, eight tables are implemented: For DNA/DNA hybridizations:
  - DNA_NN1: values from Breslauer et al. (1986)
  - DNA_NN2: values from Sugimoto et al. (1996)
  - DNA_NN3: values from Allawi & SantaLucia (1997) (default)
  - DNA_NN4: values from SantaLucia & Hicks (2004)

  For RNA/RNA hybridizations:
  - RNA_NN1: values from Freier et al. (1986)
  - RNA_NN2: values from Xia et al. (1998)
  - RNA_NN3: values from Chen et al. (2012)

  For RNA/DNA hybridizations:
  - R_DNA_NN1: values from Sugimoto et al. (1995) Note that `seq` must be the RNA sequence.

  Use the module's maketable method to make a new table or to update one one of the implemented tables.
- tmm_table: Thermodynamic values for terminal mismatches. Default: DNA_TMM1 (SantaLucia & Peyret, 2001)

- imm_table: Thermodynamic values for internal mismatches, may include insosine mismatches. Default: DNA_IMM1 (Allawi & SantaLucia, 1997-1998; Peyret et al., 1999; Watkins & SantaLucia, 2005)
- de_table: Thermodynamic values for dangling ends:
  - DNA_DE1: for DNA. Values from Bommarito et al. (2000) (default)
  - RNA_DE1: for RNA. Values from Turner & Mathews (2010)
- dnac1: Concentration of the higher concentrated strand [nM]. Typically this will be the primer (for PCR) or the probe. Default=25.
- dnac2: Concentration of the lower concentrated strand [nM]. In PCR this is the template strand which concentration is typically very low and may be ignored (dnac2=0). In oligo/oligo hybridization experiments, dnac1 equals dnac1. Default=25. MELTING and Primer3Plus use k = [Oligo(Total)]/4 by default. To mimic this behaviour, you have to divide [Oligo(Total)] by 2 and assign this concentration to dnac1 and dnac2. E.g., Total oligo concentration of 50 nM in Primer3Plus means dnac1=25, dnac2=25.
- selfcomp: Is the sequence self-complementary? Default=False. If 'True' the primer is thought binding to itself, thus dnac2 is not considered.
- Na, K, Tris, Mg, dNTPs: See method 'Tm_GC' for details. Defaults: Na=50, K=0, Tris=0, Mg=0, dNTPs=0.
- saltcorr: See method 'Tm_GC'. Default=5. 0 means no salt correction.

Bio.SeqUtils.ProtParam module

Simple protein analysis.

Examples

```
>>> from Bio.SeqUtils.ProtParam import ProteinAnalysis
>>> X = ProteinAnalysis("MAEGEITTFTALTEKFNLPPGNYKKPKLLYCSNGGHFLRILPDGTVDGT"
...                     "RDRSDQHIQLQLSAESVGEVYIKSTETGQYLAMDTSGLLYGSQTPSEEC"
...                     "LFLERLEENHYNTYTSKKHAEKNWFVGLKKNGSCKRGPRTHYGQKAILF"
...                     "LPLPV")
>>> print(X.count_amino_acids()['A'])
6
>>> print(X.count_amino_acids()['E'])
12
>>> print("%0.2f" % X.get_amino_acids_percent()['A'])
0.04
>>> print("%0.2f" % X.get_amino_acids_percent()['L'])
0.12
>>> print("%0.2f" % X.molecular_weight())
17103.16
>>> print("%0.2f" % X.aromaticity())
0.10
>>> print("%0.2f" % X.instability_index())
41.98
>>> print("%0.2f" % X.isoelectric_point())
7.72
```

(continues on next page)

(continued from previous page)

```
>>> sec_struc = X.secondary_structure_fraction()  # [helix, turn, sheet]
>>> print("%0.2f" % sec_struc[0])  # helix
0.33
>>> print("%0.2f" % sec_struc[1])  # turn
0.29
>>> print("%0.2f" % sec_struc[2])  # sheet
0.37
>>> epsilon_prot = X.molar_extinction_coefficient()  # [reduced, oxidized]
>>> print(epsilon_prot[0])  # with reduced cysteines
17420
>>> print(epsilon_prot[1])  # with disulfid bridges
17545
```

Other public methods are:

- gravy
- protein_scale
- flexibility
- charge_at_pH

class Bio.SeqUtils.ProtParam.**ProteinAnalysis**(*prot_sequence*, *monoisotopic=False*)

Bases: object

Class containing methods for protein analysis.

The constructor takes two arguments. The first is the protein sequence as a string or a Seq object.

The second argument is optional. If set to True, the weight of the amino acids will be calculated using their monoisotopic mass (the weight of the most abundant isotopes for each element), instead of the average molecular mass (the averaged weight of all stable isotopes for each element). If set to false (the default value) or left out, the IUPAC average molecular mass will be used for the calculation.

__init__(*prot_sequence*, *monoisotopic=False*)

Initialize the class.

count_amino_acids()

Count standard amino acids, return a dict.

Counts the number times each amino acid is in the protein sequence. Returns a dictionary {AminoAcid:Number}.

The return value is cached in self.amino_acids_content. It is not recalculated upon subsequent calls.

get_amino_acids_percent()

Calculate the amino acid content in percentages.

The same as count_amino_acids only returns the Number in percentage of entire sequence. Returns a dictionary of {AminoAcid:percentage}.

The return value is cached in self.amino_acids_percent.

input is the dictionary self.amino_acids_content. output is a dictionary with amino acids as keys.

molecular_weight()

Calculate MW from Protein sequence.

aromaticity()

Calculate the aromaticity according to Lobry, 1994.

Calculates the aromaticity value of a protein according to Lobry, 1994. It is simply the relative frequency of Phe+Trp+Tyr.

instability_index()

Calculate the instability index according to Guruprasad et al 1990.

Implementation of the method of Guruprasad et al. 1990 to test a protein for stability. Any value above 40 means the protein is unstable (has a short half life).

See: Guruprasad K., Reddy B.V.B., Pandit M.W. Protein Engineering 4:155-161(1990).

flexibility()

Calculate the flexibility according to Vihinen, 1994.

No argument to change window size because parameters are specific for a window=9. The parameters used are optimized for determining the flexibility.

gravy(*scale='KyteDoolitle'*)

Calculate the GRAVY (Grand Average of Hydropathy) according to Kyte and Doolitle, 1982.

Utilizes the given Hydrophobicity scale, by default uses the original proposed by Kyte and Doolittle (Kyte-Doolitle). Other options are: Aboderin, AbrahamLeo, Argos, BlackMould, BullBreese, Casari, Cid, Cowan3.4, Cowan7.5, Eisenberg, Engelman, Fasman, Fauchere, GoldSack, Guy, Jones, Juretic, Kidera, Miyazawa, Parker,Ponnuswamy, Rose, Roseman, Sweet, Tanford, Wilson and Zimmerman.

New scales can be added in ProtParamData.

protein_scale(*param_dict*, *window*, *edge=1.0*)

Compute a profile by any amino acid scale.

An amino acid scale is defined by a numerical value assigned to each type of amino acid. The most frequently used scales are the hydrophobicity or hydrophilicity scales and the secondary structure conformational parameters scales, but many other scales exist which are based on different chemical and physical properties of the amino acids. You can set several parameters that control the computation of a scale profile, such as the window size and the window edge relative weight value.

WindowSize: The window size is the length of the interval to use for the profile computation. For a window size n, we use the i-(n-1)/2 neighboring residues on each side to compute the score for residue i. The score for residue i is the sum of the scaled values for these amino acids, optionally weighted according to their position in the window.

Edge: The central amino acid of the window always has a weight of 1. By default, the amino acids at the remaining window positions have the same weight, but you can make the residue at the center of the window have a larger weight than the others by setting the edge value for the residues at the beginning and end of the interval to a value between 0 and 1. For instance, for Edge=0.4 and a window size of 5 the weights will be: 0.4, 0.7, 1.0, 0.7, 0.4.

The method returns a list of values which can be plotted to view the change along a protein sequence. Many scales exist. Just add your favorites to the ProtParamData modules.

Similar to expasy's ProtScale: http://www.expasy.org/cgi-bin/protscale.pl

isoelectric_point()

Calculate the isoelectric point.

Uses the module IsoelectricPoint to calculate the pI of a protein.

charge_at_pH(*pH*)

Calculate the charge of a protein at given pH.

secondary_structure_fraction()

Calculate fraction of helix, turn and sheet.

Returns a list of the fraction of amino acids which tend to be in Helix, Turn or Sheet, according to Haimov and Srebnik, 2016; Hutchinson and Thornton, 1994; and Kim and Berg, 1993, respectively.

Amino acids in helix: E, M, A, L, K. Amino acids in turn: N, P, G, S, D. Amino acids in sheet: V, I, Y, F, W, L, T.

Note that, prior to v1.82, this method wrongly returned (Sheet, Turn, Helix) while claiming to return (Helix, Turn, Sheet).

Returns a tuple of three floats (Helix, Turn, Sheet).

molar_extinction_coefficient()

Calculate the molar extinction coefficient.

Calculates the molar extinction coefficient assuming cysteines (reduced) and cystines residues (Cys-Cys-bond)

Bio.SeqUtils.ProtParamData module

Indices to be used with ProtParam.

Bio.SeqUtils.lcc module

Local Composition Complexity.

Bio.SeqUtils.lcc.**lcc_mult**(*seq*, *wsize*)

Calculate Local Composition Complexity (LCC) values over sliding window.

Returns a list of floats, the LCC values for a sliding window over the sequence.

seq - an unambiguous DNA sequence (a string or Seq object) wsize - window size, integer

The result is the same as applying lcc_simp multiple times, but this version is optimized for speed. The optimization works by using the value of previous window as a base to compute the next one.

Bio.SeqUtils.lcc.**lcc_simp**(*seq*)

Calculate Local Composition Complexity (LCC) for a sequence.

seq - an unambiguous DNA sequence (a string or Seq object)

Returns the Local Composition Complexity (LCC) value for the entire sequence (as a float).

Reference: Andrzej K Konopka (2005) Sequence Complexity and Composition https://doi.org/10.1038/npg.els.0005260

Module contents

Miscellaneous functions for dealing with sequences.

`Bio.SeqUtils.gc_fraction`(*seq*, *ambiguous='remove'*)

Calculate G+C percentage in seq (float between 0 and 1).

Copes with mixed case sequences. Ambiguous Nucleotides in this context are those different from ATCGSWU (S is G or C, and W is A or T).

If ambiguous equals "remove" (default), will only count GCS and will only include ACTGSWU when calculating the sequence length. Equivalent to removing all characters in the set BDHKMNRVXY before calculating the GC content, as each of these ambiguous nucleotides can either be in (A,T) or in (C,G).

If ambiguous equals "ignore", it will treat only unambiguous nucleotides (GCS) as counting towards the GC percentage, but will include all ambiguous and unambiguous nucleotides when calculating the sequence length.

If ambiguous equals "weighted", will use a "mean" value when counting the ambiguous characters, for example, G and C will be counted as 1, N and X will be counted as 0.5, D will be counted as 0.33 etc. See Bio.SeqUtils._gc_values for a full list.

Will raise a ValueError for any other value of the ambiguous parameter.

```
>>> from Bio.SeqUtils import gc_fraction
>>> seq = "ACTG"
>>> print(f"GC content of {seq} : {gc_fraction(seq):.2f}")
GC content of ACTG : 0.50
```

Example with an RNA sequence:

```
>>> seq = "GGAUCUUCGGAUCU"
>>> print(f"GC content of {seq} : {gc_fraction(seq):.2f}")
GC content of GGAUCUUCGGAUCU : 0.50
```

S and W are ambiguous for the purposes of calculating the GC content.

```
>>> seq = "ACTGSSSS"
>>> gc = gc_fraction(seq, "remove")
>>> print(f"GC content of {seq} : {gc:.2f}")
GC content of ACTGSSSS : 0.75
>>> gc = gc_fraction(seq, "ignore")
>>> print(f"GC content of {seq} : {gc:.2f}")
GC content of ACTGSSSS : 0.75
>>> gc = gc_fraction(seq, "weighted")
>>> print(f"GC content with ambiguous counting: {gc:.2f}")
GC content with ambiguous counting: 0.75
```

Some examples with ambiguous nucleotides.

```
>>> seq = "ACTGN"
>>> gc = gc_fraction(seq, "ignore")
>>> print(f"GC content of {seq} : {gc:.2f}")
GC content of ACTGN : 0.40
>>> gc = gc_fraction(seq, "weighted")
>>> print(f"GC content with ambiguous counting: {gc:.2f}")
GC content with ambiguous counting: 0.50
>>> gc = gc_fraction(seq, "remove")
```

(continues on next page)

(continued from previous page)

```
>>> print(f"GC content with ambiguous removing: {gc:.2f}")
GC content with ambiguous removing: 0.50
```

Ambiguous nucleotides are also removed from the length of the sequence.

```
>>> seq = "GDVV"
>>> gc = gc_fraction(seq, "ignore")
>>> print(f"GC content of {seq} : {gc:.2f}")
GC content of GDVV : 0.25
>>> gc = gc_fraction(seq, "weighted")
>>> print(f"GC content with ambiguous counting: {gc:.4f}")
GC content with ambiguous counting: 0.6667
>>> gc = gc_fraction(seq, "remove")
>>> print(f"GC content with ambiguous removing: {gc:.2f}")
GC content with ambiguous removing: 1.00
```

Note that this will return zero for an empty sequence.

Bio.SeqUtils.**GC123**(*seq*)

Calculate G+C content: total, for first, second and third positions.

Returns a tuple of four floats (percentages between 0 and 100) for the entire sequence, and the three codon positions. e.g.

```
>>> from Bio.SeqUtils import GC123
>>> GC123("ACTGTN")
(40.0, 50.0, 50.0, 0.0)
```

Copes with mixed case sequences, but does NOT deal with ambiguous nucleotides.

Bio.SeqUtils.**GC_skew**(*seq*, *window=100*)

Calculate GC skew (G-C)/(G+C) for multiple windows along the sequence.

Returns a list of ratios (floats), controlled by the length of the sequence and the size of the window.

Returns 0 for windows without any G/C by handling zero division errors.

Does NOT look at any ambiguous nucleotides.

Bio.SeqUtils.**xGC_skew**(*seq*, *window=1000*, *zoom=100*, *r=300*, *px=100*, *py=100*)

Calculate and plot normal and accumulated GC skew (GRAPHICS !!!).

Bio.SeqUtils.**nt_search**(*seq*, *subseq*)

Search for a DNA subseq in seq, return list of [subseq, positions].

Use ambiguous values (like N = A or T or C or G, R = A or G etc.), searches only on forward strand.

Bio.SeqUtils.**seq3**(*seq*, *custom_map=None*, *undef_code='Xaa'*)

Convert protein sequence from one-letter to three-letter code.

The single required input argument 'seq' should be a protein sequence using single letter codes, either as a Python string or as a Seq or MutableSeq object.

This function returns the amino acid sequence as a string using the three letter amino acid codes. Output follows the IUPAC standard (including ambiguous characters B for "Asx", J for "Xle" and X for "Xaa", and also U for "Sel" and O for "Pyl") plus "Ter" for a terminator given as an asterisk. Any unknown character (including possible gap characters), is changed into 'Xaa' by default.

e.g.

```
>>> from Bio.SeqUtils import seq3
>>> seq3("MAIVMGRWKGAR*")
'MetAlaIleValMetGlyArgTrpLysGlyAlaArgTer'
```

You can set a custom translation of the codon termination code using the dictionary "custom_map" argument (which defaults to {'*': 'Ter'}), e.g.

```
>>> seq3("MAIVMGRWKGAR*", custom_map={"*": "***"})
'MetAlaIleValMetGlyArgTrpLysGlyAlaArg***'
```

You can also set a custom translation for non-amino acid characters, such as '-', using the "undef_code" argument, e.g.

```
>>> seq3("MAIVMGRWKGA--R*", undef_code='---')
'MetAlaIleValMetGlyArgTrpLysGlyAla------ArgTer'
```

If not given, "undef_code" defaults to "Xaa", e.g.

```
>>> seq3("MAIVMGRWKGA--R*")
'MetAlaIleValMetGlyArgTrpLysGlyAlaXaaXaaArgTer'
```

This function was inspired by BioPerl's seq3.

Bio.SeqUtils.**seq1**(*seq*, *custom_map=None*, *undef_code='X'*)

Convert protein sequence from three-letter to one-letter code.

The single required input argument 'seq' should be a protein sequence using three-letter codes, either as a Python string or as a Seq or MutableSeq object.

This function returns the amino acid sequence as a string using the one letter amino acid codes. Output follows the IUPAC standard (including ambiguous characters "B" for "Asx", "J" for "Xle", "X" for "Xaa", "U" for "Sel", and "O" for "Pyl") plus "*" for a terminator given the "Ter" code. Any unknown character (including possible gap characters), is changed into '-' by default.

e.g.

```
>>> from Bio.SeqUtils import seq1
>>> seq1("MetAlaIleValMetGlyArgTrpLysGlyAlaArgTer")
'MAIVMGRWKGAR*'
```

The input is case insensitive, e.g.

```
>>> from Bio.SeqUtils import seq1
>>> seq1("METalaIlEValMetGLYArgtRplysGlyAlaARGTer")
'MAIVMGRWKGAR*'
```

You can set a custom translation of the codon termination code using the dictionary "custom_map" argument (defaulting to {'Ter': '*'}), e.g.

```
>>> seq1("MetAlaIleValMetGlyArgTrpLysGlyAla***", custom_map={"***": "*"})
'MAIVMGRWKGA*'
```

You can also set a custom translation for non-amino acid characters, such as '-', using the "undef_code" argument, e.g.

```
>>> seq1("MetAlaIleValMetGlyArgTrpLysGlyAla------ArgTer", undef_code='?')
'MAIVMGRWKGA??R*'
```

If not given, "undef_code" defaults to "X", e.g.

```
>>> seq1("MetAlaIleValMetGlyArgTrpLysGlyAla------ArgTer")
'MAIVMGRWKGAXXR*'
```

Bio.SeqUtils.**molecular_weight**(*seq*, *seq_type='DNA'*, *double_stranded=False*, *circular=False*, *monoisotopic=False*)

Calculate the molecular mass of DNA, RNA or protein sequences as float.

Only unambiguous letters are allowed. Nucleotide sequences are assumed to have a 5' phosphate.

Arguments:

- seq: string, Seq, or SeqRecord object.
- seq_type: The default is to assume DNA; override this with a string "DNA", "RNA", or "protein".
- double_stranded: Calculate the mass for the double stranded molecule?
- circular: Is the molecule circular (has no ends)?
- monoisotopic: Use the monoisotopic mass tables?

```
>>> print("%0.2f" % molecular_weight("AGC"))
949.61
>>> print("%0.2f" % molecular_weight(Seq("AGC")))
949.61
```

However, it is better to be explicit - for example with strings:

```
>>> print("%0.2f" % molecular_weight("AGC", "DNA"))
949.61
>>> print("%0.2f" % molecular_weight("AGC", "RNA"))
997.61
>>> print("%0.2f" % molecular_weight("AGC", "protein"))
249.29
```

Bio.SeqUtils.**six_frame_translations**(*seq*, *genetic_code=1*)

Return pretty string showing the 6 frame translations and GC content.

Nice looking 6 frame translation with GC content - code from xbbtools similar to DNA Striders six-frame translation

```
>>> from Bio.SeqUtils import six_frame_translations
>>> print(six_frame_translations("AUGGCCAUUGUAAUGGGCCGCUGA"))
GC_Frame: a:5 t:0 g:8 c:5
Sequence: auggccauug ... gggccgcuga, 24 nt, 54.17 %GC


1/1
  G  H  C  N  G  P  L
 W  P  L  *  W  A  A
M  A  I  V  M  G  R  *
auggccauuguaaugggccgcuga  54 %
```

(continues on next page)

(continued from previous page)

```
uaccgguaacauuacccggcgacu
A  M  T  I  P  R  Q
 H  G  N  Y  H  A  A  S
  P  W  Q  L  P  G  S
```

class Bio.SeqUtils.**CodonAdaptationIndex**(*sequences*, *table=standard_dna_table*)

Bases: dict

A codon adaptation index (CAI) implementation.

Implements the codon adaptation index (CAI) described by Sharp and Li (Nucleic Acids Res. 1987 Feb 11;15(3):1281-95).

__init__(*sequences*, *table=standard_dna_table*)

Generate a codon adaptiveness table from the coding DNA sequences.

This calculates the relative adaptiveness of each codon (w_ij) as defined by Sharp & Li (Nucleic Acids Research 15(3): 1281-1295 (1987)) from the provided codon DNA sequences.

Arguments:

- **sequences: An iterable over DNA sequences, which may be plain**
  strings, Seq objects, MutableSeq objects, or SeqRecord objects.
- **table: A Bio.Data.CodonTable.CodonTable object defining the**
  genetic code. By default, the standard genetic code is used.

calculate(*sequence*)

Calculate and return the CAI (float) for the provided DNA sequence.

optimize(*sequence*, *seq_type='DNA'*, *strict=True*)

Return a new DNA sequence with preferred codons only.

Uses the codon adaptiveness table defined by the CodonAdaptationIndex object to generate DNA sequences with only preferred codons. May be useful when designing DNA sequences for transgenic protein expression or codon-optimized proteins like fluorophores.

Arguments:

- **sequence: DNA, RNA, or protein sequence to codon-optimize.**
  Supplied as a str, Seq, or SeqRecord object.
- **seq_type: String specifying type of sequence provided.**
  Options are “DNA”, “RNA”, and “protein”. Default is “DNA”.
- **strict: Determines whether an exception should be raised when**
  two codons are equally preferred for a given amino acid.

Returns:

Seq object with DNA encoding the same protein as the sequence argument, but using only preferred codons as defined by the codon adaptation index. If multiple codons are equally preferred, a warning is issued and one codon is chosen for use in the optimized sequence.

__str__()

Return str(self).

28.1.31 Bio.Sequencing package

Subpackages

Bio.Sequencing.Applications package

Module contents

Sequencing related command line application wrappers (OBSOLETE).

We have decided to remove this module in future, and instead recommend building your command and invoking it via the subprocess module directly.

class Bio.Sequencing.Applications.**BwaIndexCommandline**(*cmd='bwa'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for Burrows Wheeler Aligner (BWA) index.

Index database sequences in the FASTA format, equivalent to:

```
$ bwa index [-p prefix] [-a algoType] [-c] <in.db.fasta>
```

See http://bio-bwa.sourceforge.net/bwa.shtml for details.

Examples

```
>>> from Bio.Sequencing.Applications import BwaIndexCommandline
>>> reference_genome = "/path/to/reference_genome.fasta"
>>> index_cmd = BwaIndexCommandline(infile=reference_genome, algorithm="bwtsw")
>>> print(index_cmd)
bwa index -a bwtsw /path/to/reference_genome.fasta
```

You would typically run the command using index_cmd() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='bwa'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property algorithm

Algorithm for constructing BWT index.

Available options are:

- is: IS linear-time algorithm for constructing suffix array. It requires 5.37N memory where N is the size of the database. IS is moderately fast, but does not work with database larger than 2GB. IS is the default algorithm due to its simplicity.
- bwtsw: Algorithm implemented in BWT-SW. This method works with the whole human genome, but it does not work with database smaller than 10MB and it is usually slower than IS.

This controls the addition of the -a parameter and its associated value. Set this property to the argument value required.

property c

Build color-space index. The input fasta should be in nucleotide space.

This property controls the addition of the -c switch, treat this property as a boolean.

property infile

Input file name

This controls the addition of the infile parameter and its associated value. Set this property to the argument value required.

property prefix

Prefix of the output database [same as db filename]

This controls the addition of the -p parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**BwaAlignCommandline**(*cmd='bwa'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for Burrows Wheeler Aligner (BWA) aln.

Run a BWA alignment, equivalent to:

```
$ bwa aln [...] <in.db.fasta> <in.query.fq> > <out.sai>
```

See http://bio-bwa.sourceforge.net/bwa.shtml for details.

Examples

```
>>> from Bio.Sequencing.Applications import BwaAlignCommandline
>>> reference_genome = "/path/to/reference_genome.fasta"
>>> read_file = "/path/to/read_1.fq"
>>> output_sai_file = "/path/to/read_1.sai"
>>> align_cmd = BwaAlignCommandline(reference=reference_genome, read_file=read_file)
>>> print(align_cmd)
bwa aln /path/to/reference_genome.fasta /path/to/read_1.fq
```

You would typically run the command line using align_cmd(stdout=output_sai_file) or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='bwa'*, ***kwargs*)

Initialize the class.

property B

Length of barcode starting from the 5-end. When INT is positive, the barcode of each read will be trimmed before mapping and will be written at the BC SAM tag. For paired-end reads, the barcode from both ends are concatenated. [0]

This controls the addition of the -B parameter and its associated value. Set this property to the argument value required.

property E

Gap extension penalty [4]

This controls the addition of the -E parameter and its associated value. Set this property to the argument value required.

property I

The input is in the Illumina 1.3+ read format (quality equals ASCII-64).

This property controls the addition of the -I switch, treat this property as a boolean.

property M

Mismatch penalty. BWA will not search for suboptimal hits with a score lower than (bestScore-misMsc). [3]

This controls the addition of the -M parameter and its associated value. Set this property to the argument value required.

property N

Disable iterative search. All hits with no more than maxDiff differences will be found. This mode is much slower than the default.

This property controls the addition of the -N switch, treat this property as a boolean.

property O

Gap open penalty [11]

This controls the addition of the -O parameter and its associated value. Set this property to the argument value required.

property R

Proceed with suboptimal alignments if there are no more than INT equally best hits.

> This option only affects paired-end mapping. Increasing this threshold helps to improve the pairing accuracy at the cost of speed, especially for short reads (~32bp).

This controls the addition of the -R parameter and its associated value. Set this property to the argument value required.

__annotations__ = {}

property b

Specify the input read sequence file is the BAM format

This property controls the addition of the -b switch, treat this property as a boolean.

property b1

When -b is specified, only use the first read in a read pair in mapping (skip single-end reads and the second reads).

This property controls the addition of the -b1 switch, treat this property as a boolean.

property b2

When -b is specified, only use the second read in a read pair in mapping.

This property controls the addition of the -b2 switch, treat this property as a boolean.

property c

Reverse query but not complement it, which is required for alignment in the color space.

This property controls the addition of the -c switch, treat this property as a boolean.

property d

Disallow a long deletion within INT bp towards the 3-end [16]

This controls the addition of the -d parameter and its associated value. Set this property to the argument value required.

property e

Maximum number of gap extensions, -1 for k-difference mode (disallowing long gaps) [-1]

This controls the addition of the -e parameter and its associated value. Set this property to the argument value required.

property i

Disallow an indel within INT bp towards the ends [5]

This controls the addition of the -i parameter and its associated value. Set this property to the argument value required.

property k

Maximum edit distance in the seed [2]

This controls the addition of the -k parameter and its associated value. Set this property to the argument value required.

property l

Take the first INT subsequence as seed.

If INT is larger than the query sequence, seeding will be disabled. For long reads, this option is typically ranged from 25 to 35 for -k 2. [inf]

This controls the addition of the -l parameter and its associated value. Set this property to the argument value required.

property n

Maximum edit distance if the value is INT, or the fraction of missing alignments given 2% uniform base error rate if FLOAT. In the latter case, the maximum edit distance is automatically chosen for different read lengths. [0.04]

This controls the addition of the -n parameter and its associated value. Set this property to the argument value required.

property o

Maximum edit distance if the value is INT, or the fraction of missing alignments given 2% uniform base error rate if FLOAT. In the latter case, the maximum edit distance is automatically chosen for different read lengths. [0.04]

This controls the addition of the -o parameter and its associated value. Set this property to the argument value required.

property q

Parameter for read trimming [0].

BWA trims a read down to argmax_x{sum_{i=x+1}^l(INT-q_i)} if q_l<INT where l is the original read length.

This controls the addition of the -q parameter and its associated value. Set this property to the argument value required.

property read_file

Read file name

This controls the addition of the read_file parameter and its associated value. Set this property to the argument value required.

property reference

Reference file name

This controls the addition of the reference parameter and its associated value. Set this property to the argument value required.

property t

Number of threads (multi-threading mode) [1]

This controls the addition of the -t parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**BwaSamseCommandline**(*cmd='bwa'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for Burrows Wheeler Aligner (BWA) samse.

Generate alignments in the SAM format given single-end reads. Equvialent to:

```
$ bwa samse [-n maxOcc] <in.db.fasta> <in.sai> <in.fq> > <out.sam>
```

See http://bio-bwa.sourceforge.net/bwa.shtml for details.

Examples

```
>>> from Bio.Sequencing.Applications import BwaSamseCommandline
>>> reference_genome = "/path/to/reference_genome.fasta"
>>> read_file = "/path/to/read_1.fq"
>>> sai_file = "/path/to/read_1.sai"
>>> output_sam_file = "/path/to/read_1.sam"
>>> samse_cmd = BwaSamseCommandline(reference=reference_genome,
...                                 read_file=read_file, sai_file=sai_file)
>>> print(samse_cmd)
bwa samse /path/to/reference_genome.fasta /path/to/read_1.sai /path/to/read_1.fq
```

You would typically run the command line using samse_cmd(stdout=output_sam_file) or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='bwa'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property n

Maximum number of alignments to output in the XA tag for reads paired properly.

If a read has more than INT hits, the XA tag will not be written. [3]

This controls the addition of the -n parameter and its associated value. Set this property to the argument value required.

property r

Specify the read group in a format like '@RG ID:foo SM:bar'. [null]

This controls the addition of the -r parameter and its associated value. Set this property to the argument value required.

property read_file

Read file name

This controls the addition of the read_file parameter and its associated value. Set this property to the argument value required.

property reference

Reference file name

This controls the addition of the reference parameter and its associated value. Set this property to the argument value required.

property sai_file

Sai file name

This controls the addition of the sai_file parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**BwaSampeCommandline**(*cmd='bwa'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for Burrows Wheeler Aligner (BWA) sampe.

Generate alignments in the SAM format given paired-end reads. Equivalent to:

```
$ bwa sampe [...] <in.db.fasta> <in1.sai> <in2.sai> <in1.fq> <in2.fq> > <out.sam>
```

See http://bio-bwa.sourceforge.net/bwa.shtml for details.

Examples

```
>>> from Bio.Sequencing.Applications import BwaSampeCommandline
>>> reference_genome = "/path/to/reference_genome.fasta"
>>> read_file1 = "/path/to/read_1.fq"
>>> read_file2 = "/path/to/read_2.fq"
>>> sai_file1 = "/path/to/read_1.sai"
>>> sai_file2 = "/path/to/read_2.sai"
>>> output_sam_file = "/path/to/output.sam"
>>> read_group = r"@RG\tID:foo\tSM:bar"  # BWA will turn backslash-t into tab
>>> sampe_cmd = BwaSampeCommandline(reference=reference_genome,
...                                 sai_file1=sai_file1, sai_file2=sai_file2,
...                                 read_file1=read_file1, read_file2=read_file2,
...                                 r=read_group)
>>> print(sampe_cmd)
bwa sampe /path/to/reference_genome.fasta /path/to/read_1.sai /path/to/read_2.sai /
↪path/to/read_1.fq /path/to/read_2.fq -r @RG\tID:foo\tSM:bar
```

You would typically run the command line using sampe_cmd(stdout=output_sam_file) or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='bwa'*, ***kwargs*)

Initialize the class.

property N

Maximum number of alignments to output in the XA tag for disconcordant read pairs (excluding singletons) [10].

If a read has more than INT hits, the XA tag will not be written.

This controls the addition of the -N parameter and its associated value. Set this property to the argument value required.

`__annotations__ = {}`

`property a`

Maximum insert size for a read pair to be considered being mapped properly [500].

Since 0.4.5, this option is only used when there are not enough good alignments to infer the distribution of insert sizes.

This controls the addition of the -a parameter and its associated value. Set this property to the argument value required.

`property n`

Maximum number of alignments to output in the XA tag for reads paired properly [3].

If a read has more than INT hits, the XA tag will not be written.

This controls the addition of the -n parameter and its associated value. Set this property to the argument value required.

`property o`

Maximum occurrences of a read for pairing [100000].

A read with more occurrences will be treated as a single-end read. Reducing this parameter helps faster pairing.

This controls the addition of the -o parameter and its associated value. Set this property to the argument value required.

`property r`

Specify the read group in a format like '@RG ID:foo SM:bar'. [null]

This controls the addition of the -r parameter and its associated value. Set this property to the argument value required.

`property read_file1`

Read file 1

This controls the addition of the read_file1 parameter and its associated value. Set this property to the argument value required.

`property read_file2`

Read file 2

This controls the addition of the read_file2 parameter and its associated value. Set this property to the argument value required.

`property reference`

Reference file name

This controls the addition of the reference parameter and its associated value. Set this property to the argument value required.

`property sai_file1`

Sai file 1

This controls the addition of the sai_file1 parameter and its associated value. Set this property to the argument value required.

property sai_file2

Sai file 2

This controls the addition of the sai_file2 parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**BwaBwaswCommandline**(*cmd='bwa'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for Burrows Wheeler Aligner (BWA) bwasw.

Align query sequences from FASTQ files. Equivalent to:

```
$ bwa bwasw [...] <in.db.fasta> <in.fq>
```

See http://bio-bwa.sourceforge.net/bwa.shtml for details.

Examples

```
>>> from Bio.Sequencing.Applications import BwaBwaswCommandline
>>> reference_genome = "/path/to/reference_genome.fasta"
>>> read_file = "/path/to/read_1.fq"
>>> bwasw_cmd = BwaBwaswCommandline(reference=reference_genome, read_file=read_file)
>>> print(bwasw_cmd)
bwa bwasw /path/to/reference_genome.fasta /path/to/read_1.fq
```

You would typically run the command line using bwasw_cmd() or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='bwa'*, ***kwargs*)

Initialize the class.

property N

Minimum number of seeds supporting the resultant alignment to skip reverse alignment. [5]

This controls the addition of the -N parameter and its associated value. Set this property to the argument value required.

property T

Minimum score threshold divided by a [37]

This controls the addition of the -T parameter and its associated value. Set this property to the argument value required.

__annotations__ = {}

property a

Score of a match [1]

This controls the addition of the -a parameter and its associated value. Set this property to the argument value required.

property b

Mismatch penalty [3]

This controls the addition of the -b parameter and its associated value. Set this property to the argument value required.

property c

Coefficient for threshold adjustment according to query length [5.5].

Given an l-long query, the threshold for a hit to be retained is a*max{T,c*log(l)}.

This controls the addition of the -c parameter and its associated value. Set this property to the argument value required.

property mate_file

Mate file

This controls the addition of the mate_file parameter and its associated value. Set this property to the argument value required.

property q

Gap open penalty [5]

This controls the addition of the -q parameter and its associated value. Set this property to the argument value required.

property r

Gap extension penalty. The penalty for a contiguous gap of size k is q+k*r. [2]

This controls the addition of the -r parameter and its associated value. Set this property to the argument value required.

property read_file

Read file

This controls the addition of the read_file parameter and its associated value. Set this property to the argument value required.

property reference

Reference file name

This controls the addition of the reference parameter and its associated value. Set this property to the argument value required.

property s

Maximum SA interval size for initiating a seed [3].

Higher -s increases accuracy at the cost of speed.

This controls the addition of the -s parameter and its associated value. Set this property to the argument value required.

property t

Number of threads in the multi-threading mode [1]

This controls the addition of the -t parameter and its associated value. Set this property to the argument value required.

property w

Band width in the banded alignment [33]

This controls the addition of the -w parameter and its associated value. Set this property to the argument value required.

property z

Z-best heuristics. Higher -z increases accuracy at the cost of speed. [1]

This controls the addition of the -z parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**BwaMemCommandline**(*cmd='bwa'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for Burrows Wheeler Aligner (BWA) mem.

Run a BWA-MEM alignment, with single- or paired-end reads, equivalent to:

```
$ bwa mem [...] <in.db.fasta> <in1.fq> <in2.fq> > <out.sam>
```

See http://bio-bwa.sourceforge.net/bwa.shtml for details.

Examples

```
>>> from Bio.Sequencing.Applications import BwaMemCommandline
>>> reference_genome = "/path/to/reference_genome.fasta"
>>> read_file = "/path/to/read_1.fq"
>>> output_sam_file = "/path/to/output.sam"
>>> align_cmd = BwaMemCommandline(reference=reference_genome, read_file1=read_file)
>>> print(align_cmd)
bwa mem /path/to/reference_genome.fasta /path/to/read_1.fq
```

You would typically run the command line using align_cmd(stdout=output_sam_file) or via the Python subprocess module, as described in the Biopython tutorial.

__init__(*cmd='bwa'*, ***kwargs*)

Initialize the class.

property A

Matching score. [1]

This controls the addition of the -A parameter and its associated value. Set this property to the argument value required.

property B

Mismatch penalty. The sequence error rate is approximately: {.75 * exp[-log(4) * B/A]}. [4]

This controls the addition of the -B parameter and its associated value. Set this property to the argument value required.

property C

Append FASTA/Q comment to SAM output. This option can be used to transfer read meta information (e.g. barcode) to the SAM output. Note that the FASTA/Q comment (the string after a space in the header line) must conform the SAM spec (e.g. BC:Z:CGTAC). Malformated comments lead to incorrect SAM output.

This property controls the addition of the -C switch, treat this property as a boolean.

property E

Gap extension penalty. A gap of length k costs O + k*E (i.e. -O is for opening a zero-length gap). [1]

This controls the addition of the -E parameter and its associated value. Set this property to the argument value required.

property H

Use hard clipping 'H' in the SAM output. This option may dramatically reduce the redundancy of output when mapping long contig or BAC sequences.

This property controls the addition of the -H switch, treat this property as a boolean.

property L

Clipping penalty. When performing SW extension, BWA-MEM keeps track of the best score reaching the end of query. If this score is larger than the best SW score minus the clipping penalty, clipping will not be applied. Note that in this case, the SAM AS tag reports the best SW score; clipping penalty is not deducted. [5]

This controls the addition of the -L parameter and its associated value. Set this property to the argument value required.

property M

Mark shorter split hits as secondary (for Picard compatibility).

This property controls the addition of the -M switch, treat this property as a boolean.

property O

Gap open penalty. [6]

This controls the addition of the -O parameter and its associated value. Set this property to the argument value required.

property P

In the paired-end mode, perform SW to rescue missing hits only but do not try to find hits that fit a proper pair.

This property controls the addition of the -P switch, treat this property as a boolean.

property R

Complete read group header line. 't' can be used in STR and will be converted to a TAB in the output SAM. The read group ID will be attached to every read in the output. An example is '@RG ID:foo SM:bar'. [null]

This controls the addition of the -R parameter and its associated value. Set this property to the argument value required.

property T

Don't output alignment with score lower than INT. This option only affects output. [30]

This controls the addition of the -T parameter and its associated value. Set this property to the argument value required.

property U

Penalty for an unpaired read pair. BWA-MEM scores an unpaired read pair as scoreRead1+scoreRead2-INT and scores a paired as scoreRead1+scoreRead2-insertPenalty. It compares these two scores to determine whether we should force pairing. [9]

This controls the addition of the -U parameter and its associated value. Set this property to the argument value required.

__annotations__ = {}

property a

Output all found alignments for single-end or unpaired paired-end reads. These alignments will be flagged as secondary alignments.

This property controls the addition of the -a switch, treat this property as a boolean.

property c

Discard a MEM if it has more than INT occurrence in the genome. This is an insensitive parameter. [10000]

This controls the addition of the -c parameter and its associated value. Set this property to the argument value required.

property d

Off-diagonal X-dropoff (Z-dropoff). Stop extension when the difference between the best and the current extension score is above |i-j|*A+INT, where i and j are the current positions of the query and reference, respectively, and A is the matching score. Z-dropoff is similar to BLAST's X-dropoff except that it doesn't penalize gaps in one of the sequences in the alignment. Z-dropoff not only avoids unnecessary extension, but also reduces poor alignments inside a long good alignment. [100]

This controls the addition of the -d parameter and its associated value. Set this property to the argument value required.

property k

Minimum seed length. Matches shorter than INT will be missed. The alignment speed is usually insensitive to this value unless it significantly deviates 20. [19]

This controls the addition of the -k parameter and its associated value. Set this property to the argument value required.

property p

Assume the first input query file is interleaved paired-end FASTA/Q. See the command description for details.

This property controls the addition of the -p switch, treat this property as a boolean.

property r

Trigger re-seeding for a MEM longer than minSeedLen*FLOAT. This is a key heuristic parameter for tuning the performance. Larger value yields fewer seeds, which leads to faster alignment speed but lower accuracy. [1.5]

This controls the addition of the -r parameter and its associated value. Set this property to the argument value required.

property read_file1

Read 1 file name

This controls the addition of the read_file1 parameter and its associated value. Set this property to the argument value required.

property read_file2

Read 2 file name

This controls the addition of the read_file2 parameter and its associated value. Set this property to the argument value required.

property reference

Reference file name

This controls the addition of the reference parameter and its associated value. Set this property to the argument value required.

property t

Number of threads [1]

This controls the addition of the -t parameter and its associated value. Set this property to the argument value required.

property v

Control the verbose level of the output. This option has not been fully supported throughout BWA. Ideally, a value 0 for disabling all the output to stderr; 1 for outputting errors only; 2 for warnings and errors; 3 for all normal messages; 4 or higher for debugging. When this option takes value 4, the output is not SAM. [3]

This controls the addition of the -v parameter and its associated value. Set this property to the argument value required.

property w

Band width. Essentially, gaps longer than INT will not be found. Note that the maximum gap length is also affected by the scoring matrix and the hit length, not solely determined by this option. [100]

This controls the addition of the -w parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**NovoalignCommandline**(*cmd='novoalign'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for novoalign by Novocraft.

See www.novocraft.com - novoalign is a short read alignment program.

Examples

```
>>> from Bio.Sequencing.Applications import NovoalignCommandline
>>> novoalign_cline = NovoalignCommandline(database='some_db',
...                                        readfile='some_seq.txt')
>>> print(novoalign_cline)
novoalign -d some_db -f some_seq.txt
```

As with all the Biopython application wrappers, you can also add or change options after creating the object:

```
>>> novoalign_cline.format = 'PRBnSEQ'
>>> novoalign_cline.r_method='0.99' # limited valid values
>>> novoalign_cline.fragment = '250 20' # must be given as a string
>>> novoalign_cline.miRNA = 100
>>> print(novoalign_cline)
novoalign -d some_db -f some_seq.txt -F PRBnSEQ -r 0.99 -i 250 20 -m 100
```

You would typically run the command line with novoalign_cline() or via the Python subprocess module, as described in the Biopython tutorial.

Last checked against version: 2.05.04

__init__(*cmd='novoalign'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property adapter3

Strips a 3' adapter sequence prior to alignment.

With paired ends two adapters can be specified

This controls the addition of the -a parameter and its associated value. Set this property to the argument value required.

property adapter5

Strips a 5' adapter sequence.

Similar to -a (adaptor3), but on the 5' end.

This controls the addition of the -5 parameter and its associated value. Set this property to the argument value required.

property cores

Number of threads, disabled on free versions [default: number of cores]

This controls the addition of the -c parameter and its associated value. Set this property to the argument value required.

property database

database filename

This controls the addition of the -d parameter and its associated value. Set this property to the argument value required.

property format

Format of read files.

Allowed values: FA, SLXFQ, STDFQ, ILMFQ, PRB, PRBnSEQ

This controls the addition of the -F parameter and its associated value. Set this property to the argument value required.

property fragment

Fragment length (2 reads + insert) and standard deviation [default: 250 30]

This controls the addition of the -i parameter and its associated value. Set this property to the argument value required.

property gap_extend

Gap extend penalty [default: 15]

This controls the addition of the -x parameter and its associated value. Set this property to the argument value required.

property gap_open

Gap opening penalty [default: 40]

This controls the addition of the -g parameter and its associated value. Set this property to the argument value required.

property good_bases

Minimum number of good quality bases [default: log(N_g, 4) + 5]

This controls the addition of the -l parameter and its associated value. Set this property to the argument value required.

property homopolymer

Homopolymer read filter [default: 20; disable: negative value]

This controls the addition of the -h parameter and its associated value. Set this property to the argument value required.

property miRNA

Sets miRNA mode and optionally sets a value for the region scanned [default: off]

This controls the addition of the -m parameter and its associated value. Set this property to the argument value required.

property qual_digits

Decimal digits for quality scores [default: 0]

This controls the addition of the -q parameter and its associated value. Set this property to the argument value required.

property quality

Lower threshold for an alignment to be reported [default: 0]

This controls the addition of the -Q parameter and its associated value. Set this property to the argument value required.

property r_method

Methods to report reads with multiple matches.

Allowed values: None, Random, All, Exhaustive, 0.99 'All' and 'Exhaustive' accept limits.

This controls the addition of the -r parameter and its associated value. Set this property to the argument value required.

property read_cal

Read quality calibration from file (mismatch counts)

This controls the addition of the -k parameter and its associated value. Set this property to the argument value required.

property readfile

read file

This controls the addition of the -f parameter and its associated value. Set this property to the argument value required.

property recorded

Alignments recorded with score equal to the best.

Default: 1000 in default read method, otherwise no limit.

This controls the addition of the -e parameter and its associated value. Set this property to the argument value required.

property repeats

If score difference is higher, report repeats.

Otherwise -r read method applies [default: 5]

This controls the addition of the -R parameter and its associated value. Set this property to the argument value required.

property report

Specifies the report format.

Allowed values: Native, Pairwise, SAM Default: Native

This controls the addition of the -o parameter and its associated value. Set this property to the argument value required.

property threshold

Threshold for alignment score

This controls the addition of the -t parameter and its associated value. Set this property to the argument value required.

property trimming

If fail to align, trim by s bases until they map or become shorter than l.

Ddefault: 2

This controls the addition of the -s parameter and its associated value. Set this property to the argument value required.

property truncate

Truncate to specific length before alignment

This controls the addition of the -n parameter and its associated value. Set this property to the argument value required.

property unconverted

Experimental: unconverted cytosines penalty in bisulfite mode

Default: no penalty

This controls the addition of the -u parameter and its associated value. Set this property to the argument value required.

property variation

Structural variation penalty [default: 70]

This controls the addition of the -v parameter and its associated value. Set this property to the argument value required.

property write_cal

Accumulate mismatch counts and write to file

This controls the addition of the -K parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**SamtoolsViewCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools view.

Extract/print all or sub alignments in SAM or BAM format, equivalent to:

```
$ samtools view [-bchuHS] [-t in.refList] [-o output] [-f reqFlag]
                [-F skipFlag] [-q minMapQ] [-l library] [-r readGroup]
                [-R rgFile] <in.bam>|<in.sam> [region1 [...]]
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsViewCommandline
>>> input_file = "/path/to/sam_or_bam_file"
>>> samtools_view_cmd = SamtoolsViewCommandline(input_file=input_file)
>>> print(samtools_view_cmd)
samtools view /path/to/sam_or_bam_file
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

property F

Skip alignments with bits present in INT

This controls the addition of the -F parameter and its associated value. Set this property to the argument value required.

property H

Output the header only

This property controls the addition of the -H switch, treat this property as a boolean.

property R

Output reads in read groups listed in FILE

This controls the addition of the -R parameter and its associated value. Set this property to the argument value required.

property S

Input is in SAM.
If @SQ header lines are absent, the '-t' option is required.

This property controls the addition of the -S switch, treat this property as a boolean.

__annotations__ = {}

property b

Output in the BAM format

This property controls the addition of the -b switch, treat this property as a boolean.

property c

Instead of printing the alignments, only count them and
print the total number.

All filter options, such as '-f', '-F' and '-q', are taken into account

This property controls the addition of the -c switch, treat this property as a boolean.

property f

Only output alignments with all bits in
INT present in the FLAG field

This controls the addition of the -f parameter and its associated value. Set this property to the argument value required.

property fast_bam

Use zlib compression level 1 to compress the output

This property controls the addition of the -1 switch, treat this property as a boolean.

property h

Include the header in the output

This property controls the addition of the -h switch, treat this property as a boolean.

property input_file

Input File Name

This controls the addition of the input parameter and its associated value. Set this property to the argument value required.

property l

Only output reads in library STR

This controls the addition of the -l parameter and its associated value. Set this property to the argument value required.

property o

Output file

This controls the addition of the -o parameter and its associated value. Set this property to the argument value required.

property q

Skip alignments with MAPQ smaller than INT

This controls the addition of the -q parameter and its associated value. Set this property to the argument value required.

property r

Only output reads in read group STR

This controls the addition of the -r parameter and its associated value. Set this property to the argument value required.

property region

Region

This controls the addition of the region parameter and its associated value. Set this property to the argument value required.

property t

This file is TAB-delimited.
Each line must contain the reference name and the length of the reference, one line for each distinct reference; additional fields are ignored.

This file also defines the order of the reference sequences in sorting. If you run 'samtools faidx <ref.fa>', the resultant index file <ref.fa>.fai can be used as this <in.ref_list> file.

This controls the addition of the -t parameter and its associated value. Set this property to the argument value required.

property u

Output uncompressed BAM.

This option saves time spent on compression/decompression and is thus preferred when the output is piped to another samtools command

This property controls the addition of the -u switch, treat this property as a boolean.

class Bio.Sequencing.Applications.**SamtoolsCalmdCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools calmd.

Generate the MD tag, equivalent to:

```
$ samtools calmd [-EeubSr] [-C capQcoef] <aln.bam> <ref.fasta>
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsCalmdCommandline
>>> input_bam = "/path/to/aln.bam"
>>> reference_fasta = "/path/to/reference.fasta"
>>> calmd_cmd = SamtoolsCalmdCommandline(input_bam=input_bam,
...                                      reference=reference_fasta)
>>> print(calmd_cmd)
samtools calmd /path/to/aln.bam /path/to/reference.fasta
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

property A

When used jointly with -r this option overwrites
the original base quality

This property controls the addition of the -A switch, treat this property as a boolean.

property C

Coefficient to cap mapping quality
of poorly mapped reads.

See the pileup command for details.

This controls the addition of the -C parameter and its associated value. Set this property to the argument value required.

property E

Extended BAQ calculation.
This option trades specificity for sensitivity, though the effect is minor.

This property controls the addition of the -E switch, treat this property as a boolean.

property S

The input is SAM with header lines

This property controls the addition of the -S switch, treat this property as a boolean.

__annotations__ = {}

property b

Output compressed BAM

This property controls the addition of the -b switch, treat this property as a boolean.

property e

Convert the read base to = if it is
identical to the aligned reference base.

Indel caller does not support the = bases at the moment.

This property controls the addition of the -e switch, treat this property as a boolean.

property input_bam

Input BAM

This controls the addition of the input parameter and its associated value. Set this property to the argument value required.

property r

Compute the BQ tag (without -A)
or cap base quality by BAQ (with -A).

This property controls the addition of the -r switch, treat this property as a boolean.

property ref

Reference FASTA to be indexed

This controls the addition of the reference parameter and its associated value. Set this property to the argument value required.

property u

Output uncompressed BAM

This property controls the addition of the -u switch, treat this property as a boolean.

class Bio.Sequencing.Applications.**SamtoolsCatCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools cat.

Concatenate BAMs, equivalent to:

```
$ samtools cat [-h header.sam] [-o out.bam] <in1.bam> <in2.bam> [ ... ]
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsCatCommandline
>>> input_bam1 = "/path/to/input_bam1"
>>> input_bam2 = "/path/to/input_bam2"
>>> input_bams = [input_bam1, input_bam2]
>>> samtools_cat_cmd = SamtoolsCatCommandline(input_bam=input_bams)
>>> print(samtools_cat_cmd)
samtools cat /path/to/input_bam1 /path/to/input_bam2
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property bams

Input BAM files

This controls the addition of the input parameter and its associated value. Set this property to the argument value required.

property h

Header SAM file

This controls the addition of the -h parameter and its associated value. Set this property to the argument value required.

property o

Output SAM file

This controls the addition of the -o parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**SamtoolsFaidxCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools faidx.

Retrieve and print stats in the index file, equivalent to:

```
$ samtools faidx <ref.fasta> [region1 [...]]
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsFaidxCommandline
>>> reference = "/path/to/reference.fasta"
>>> samtools_faidx_cmd = SamtoolsFaidxCommandline(reference=reference)
>>> print(samtools_faidx_cmd)
samtools faidx /path/to/reference.fasta
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property ref

Reference FASTA to be indexed

This controls the addition of the reference parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**SamtoolsFixmateCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools fixmate.

Fill in mate coordinates, ISIZE and mate related flags from a name-sorted alignment, equivalent to:

```
$ samtools fixmate <in.nameSrt.bam> <out.bam>
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsFixmateCommandline
>>> in_bam = "/path/to/in.nameSrt.bam"
>>> out_bam = "/path/to/out.bam"
>>> fixmate_cmd = SamtoolsFixmateCommandline(input_bam=in_bam,
...                                          out_bam=out_bam)
>>> print(fixmate_cmd)
samtools fixmate /path/to/in.nameSrt.bam /path/to/out.bam
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property input_file

Name Sorted Alignment File

This controls the addition of the in_bam parameter and its associated value. Set this property to the argument value required.

property output_file

Output file

This controls the addition of the out_bam parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**SamtoolsIdxstatsCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools idxstats.

Retrieve and print stats in the index file, equivalent to:

```
$ samtools idxstats <aln.bam>
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsIdxstatsCommandline
>>> input = "/path/to/aln_bam"
>>> samtools_idxstats_cmd = SamtoolsIdxstatsCommandline(input_bam=input)
>>> print(samtools_idxstats_cmd)
samtools idxstats /path/to/aln_bam
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property input_bam

BAM file to be indexed

This controls the addition of the input parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**SamtoolsIndexCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools index.

Index sorted alignment for fast random access, equivalent to:

```
$ samtools index <aln.bam>
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsIndexCommandline
>>> input = "/path/to/aln_bam"
>>> samtools_index_cmd = SamtoolsIndexCommandline(input_bam=input)
>>> print(samtools_index_cmd)
samtools index /path/to/aln_bam
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property input_bam

BAM file to be indexed

This controls the addition of the input parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**SamtoolsMergeCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools merge.

Merge multiple sorted alignments, equivalent to:

```
$ samtools merge [-nur1f] [-h inh.sam] [-R reg]
                 <out.bam> <in1.bam> <in2.bam> [...]
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsMergeCommandline
>>> out_bam = "/path/to/out_bam"
>>> in_bam = ["/path/to/input_bam1", "/path/to/input_bam2"]
>>> merge_cmd = SamtoolsMergeCommandline(out_bam=out_bam,
...                                      input_bam=in_bam)
>>> print(merge_cmd)
samtools merge /path/to/out_bam /path/to/input_bam1 /path/to/input_bam2
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

property R

Merge files in the specified region indicated by STR

This controls the addition of the -R parameter and its associated value. Set this property to the argument value required.

__annotations__ = {}

property bam

Input BAM

This controls the addition of the input_bam parameter and its associated value. Set this property to the argument value required.

property f

Force to overwrite the
output file if present

This property controls the addition of the -f switch, treat this property as a boolean.

property fast_bam

Use zlib compression level 1
to compress the output

This property controls the addition of the -1 switch, treat this property as a boolean.

property h

Use the lines of FILE as '@'
headers to be copied to out.bam

This controls the addition of the -h parameter and its associated value. Set this property to the argument value required.

property n

The input alignments are sorted by read names
rather than by chromosomal coordinates

This property controls the addition of the -n switch, treat this property as a boolean.

property output

Output BAM file

This controls the addition of the output_bam parameter and its associated value. Set this property to the argument value required.

property r

Attach an RG tag to each alignment.
The tag value is inferred from file names

This property controls the addition of the -r switch, treat this property as a boolean.

property u

Uncompressed BAM output

This property controls the addition of the -u switch, treat this property as a boolean.

class Bio.Sequencing.Applications.**SamtoolsMpileupCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools mpileup.

Generate BCF or pileup for one or multiple BAM files, equivalent to:

```
$ samtools mpileup [-EBug] [-C capQcoef] [-r reg] [-f in.fa]
                   [-l list] [-M capMapQ] [-Q minBaseQ]
                   [-q minMapQ] in.bam [in2.bam [...]]
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsMpileupCommandline
>>> input = ["/path/to/sam_or_bam_file"]
>>> samtools_mpileup_cmd = SamtoolsMpileupCommandline(input_file=input)
>>> print(samtools_mpileup_cmd)
samtools mpileup /path/to/sam_or_bam_file
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

property A

Do not skip anomalous read pairs in variant calling.

This property controls the addition of the -A switch, treat this property as a boolean.

property B

Disable probabilistic realignment for the
computation of base alignment quality (BAQ).

BAQ is the Phred-scaled probability of a read base being misaligned. Applying this option greatly helps to reduce false SNPs caused by misalignments

This property controls the addition of the -B switch, treat this property as a boolean.

property C

Coefficient for downgrading mapping quality for
reads containing excessive mismatches.

Given a read with a phred-scaled probability q of being generated from the mapped position, the new mapping quality is about sqrt((INT-q)/INT)*INT. A zero value disables this functionality; if enabled, the recommended value for BWA is 50

This controls the addition of the -C parameter and its associated value. Set this property to the argument value required.

property D

Output per-sample read depth

This property controls the addition of the -D switch, treat this property as a boolean.

property E

Extended BAQ computation.
This option helps sensitivity especially for MNPs, but may hurt specificity a little bit

This property controls the addition of the -E switch, treat this property as a boolean.

property I

Do not perform INDEL calling

This property controls the addition of the -I switch, treat this property as a boolean.

property L

Skip INDEL calling if the average per-sample
depth is above INT

This controls the addition of the -L parameter and its associated value. Set this property to the argument value required.

property M

Cap Mapping Quality at M

This controls the addition of the -M parameter and its associated value. Set this property to the argument value required.

property Q

Minimum base quality for a base to be considered

This controls the addition of the -Q parameter and its associated value. Set this property to the argument value required.

property S

Output per-sample Phred-scaled
strand bias P-value

This property controls the addition of the -S switch, treat this property as a boolean.

__annotations__ = {}

property b

List of input BAM files, one file per line

This controls the addition of the -b parameter and its associated value. Set this property to the argument value required.

property d

At a position, read maximally INT reads per input BAM

This controls the addition of the -d parameter and its associated value. Set this property to the argument value required.

property e

Phred-scaled gap extension sequencing error probability.

Reducing INT leads to longer indels

This controls the addition of the -e parameter and its associated value. Set this property to the argument value required.

property f

The faidx-indexed reference file in the FASTA format.

The file can be optionally compressed by razip

This controls the addition of the -f parameter and its associated value. Set this property to the argument value required.

property g

Compute genotype likelihoods and output them in the
binary call format (BCF)

This property controls the addition of the -g switch, treat this property as a boolean.

property h

Coefficient for modeling homopolymer errors.

Given an l-long homopolymer run, the sequencing error of an indel of size s is modeled as INT*s/l

This controls the addition of the -h parameter and its associated value. Set this property to the argument value required.

property illumina_13

Assume the quality is in the Illumina 1.3+ encoding

This property controls the addition of the -6 switch, treat this property as a boolean.

property input_file

Input File for generating mpileup

This controls the addition of the input_file parameter and its associated value. Set this property to the argument value required.

property l

BED or position list file containing a list of regions
or sites where pileup or BCF should be generated

This controls the addition of the -l parameter and its associated value. Set this property to the argument value required.

property o

Phred-scaled gap open sequencing error probability.

Reducing INT leads to more indel calls.

This controls the addition of the -o parameter and its associated value. Set this property to the argument value required.

property p

Comma delimited list of platforms (determined by @RG-PL)
from which indel candidates are obtained.

It is recommended to collect indel candidates from sequencing technologies that have low indel error rate such as ILLUMINA

This controls the addition of the -p parameter and its associated value. Set this property to the argument value required.

property q

Minimum mapping quality for an alignment to be used

This controls the addition of the -q parameter and its associated value. Set this property to the argument value required.

property r

Only generate pileup in region STR

This controls the addition of the -r parameter and its associated value. Set this property to the argument value required.

property u

Similar to -g except that the output is
uncompressed BCF, which is preferred for piping

This property controls the addition of the -u switch, treat this property as a boolean.

class Bio.Sequencing.Applications.**SamtoolsPhaseCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools phase.

Call and phase heterozygous SNPs, equivalent to:

```
$ samtools phase [-AF] [-k len] [-b prefix]
                 [-q minLOD] [-Q minBaseQ] <in.bam>
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsPhaseCommandline
>>> input_bam = "/path/to/in.bam"
>>> samtools_phase_cmd = SamtoolsPhaseCommandline(input_bam=input_bam)
>>> print(samtools_phase_cmd)
samtools phase /path/to/in.bam
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

property A

Drop reads with ambiguous phase

This property controls the addition of the -A switch, treat this property as a boolean.

property F

Do not attempt to fix chimeric reads

This property controls the addition of the -F switch, treat this property as a boolean.

property Q

Minimum base quality to be
used in het calling

This controls the addition of the -Q parameter and its associated value. Set this property to the argument value required.

__annotations__ = {}

property b

Prefix of BAM output

This controls the addition of the -b parameter and its associated value. Set this property to the argument value required.

property in_bam

Input file

This controls the addition of the input parameter and its associated value. Set this property to the argument value required.

property k

Maximum length for local phasing

This controls the addition of the -k parameter and its associated value. Set this property to the argument value required.

property q

Minimum Phred-scaled LOD to
call a heterozygote

This controls the addition of the -q parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**SamtoolsReheaderCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools reheader.

Replace the header in in.bam with the header in in.header.sam, equivalent to:

```
$ samtools reheader <in.header.sam> <in.bam>
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsReheaderCommandline
>>> input_header = "/path/to/header_sam_file"
>>> input_bam = "/path/to/input_bam_file"
>>> reheader_cmd = SamtoolsReheaderCommandline(input_header=input_header,
...                                            input_bam=input_bam)
>>> print(reheader_cmd)
samtools reheader /path/to/header_sam_file /path/to/input_bam_file
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property bam_file

BAM file for writing header to

This controls the addition of the input_bam parameter and its associated value. Set this property to the argument value required.

property sam_file

Sam file with header

This controls the addition of the input_header parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**SamtoolsRmdupCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools rmdup.

Remove potential PCR duplicates, equivalent to:

```
$ samtools rmdup [-sS] <input.srt.bam> <out.bam>
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsRmdupCommandline
>>> input_sorted_bam = "/path/to/input.srt.bam"
>>> out_bam = "/path/to/out.bam"
>>> rmdup_cmd = SamtoolsRmdupCommandline(input_bam=input_sorted_bam,
...                                      out_bam=out_bam)
>>> print(rmdup_cmd)
samtools rmdup /path/to/input.srt.bam /path/to/out.bam
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

property S

Treat paired-end reads
as single-end reads

This property controls the addition of the -S switch, treat this property as a boolean.

__annotations__ = {}

property input_file

Name Sorted Alignment File

This controls the addition of the in_bam parameter and its associated value. Set this property to the argument value required.

property output_file

Output file

This controls the addition of the out_bam parameter and its associated value. Set this property to the argument value required.

property s

Remove duplicates for single-end reads.

By default, the command works for paired-end reads only

This property controls the addition of the -s switch, treat this property as a boolean.

Bio.Sequencing.Applications.**SamtoolsSortCommandline**

alias of *SamtoolsVersion0xSortCommandline*

class Bio.Sequencing.Applications.**SamtoolsVersion0xSortCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools version 0.1.x sort.

Concatenate BAMs, equivalent to:

```
$ samtools sort [-no] [-m maxMem] <in.bam> <out.prefix>
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsVersion0xSortCommandline
>>> input_bam = "/path/to/input_bam"
>>> out_prefix = "/path/to/out_prefix"
>>> samtools_sort_cmd = SamtoolsVersion0xSortCommandline(input=input_bam, out_
↪prefix=out_prefix)
>>> print(samtools_sort_cmd)
samtools sort /path/to/input_bam /path/to/out_prefix
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

__annotations__ = {}

property input

Input BAM file

This controls the addition of the input parameter and its associated value. Set this property to the argument value required.

property m

Approximately the maximum required memory

This controls the addition of the -m parameter and its associated value. Set this property to the argument value required.

property n

Sort by read names rather
than by chromosomal coordinates

This property controls the addition of the -n switch, treat this property as a boolean.

property o

Output the final alignment
to the standard output

This property controls the addition of the -o switch, treat this property as a boolean.

property out_prefix

Output prefix

This controls the addition of the out_prefix parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**SamtoolsVersion1xSortCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools version 1.3.x sort.

Concatenate BAMs, equivalent to:

```
$ samtools sort [-n] [-T FREFIX] [-o file] [-I INT] [-m maxMem] <in.bam>
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsVersion1xSortCommandline
>>> input_bam = "/path/to/input_bam"
>>> FREFIX = "/path/to/out_prefix"
>>> file_name = "/path/to/out_file"
>>> samtools_sort_cmd = SamtoolsVersion1xSortCommandline(input=input_bam, T=FREFIX,␣
↪o=file_name)
>>> print(samtools_sort_cmd)
samtools sort -o /path/to/out_file -T /path/to/out_prefix /path/to/input_bam
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

property I

(INT) Set the desired compression level for the final output file,
ranging from 0 (uncompressed) or 1 (fastest but minimal compression) to 9 (best compression but slowest to write), similarly to gzip(1)'s compression level setting.

This controls the addition of the -I parameter and its associated value. Set this property to the argument value required.

property O

(FORMAT) Write the final output as sam, bam, or cram

This controls the addition of the -O parameter and its associated value. Set this property to the argument value required.

property T

(PREFIX) Write temporary files to PREFIX.nnnn.bam, or if the specified PREFIX
is an existing directory, to PREFIX/samtools.mmm.mmm.tmp.nnnn.bam, where mmm is unique to this invocation of the sort command

This controls the addition of the -T parameter and its associated value. Set this property to the argument value required.

__annotations__ = {}

property input

Input SAM/BAM/CRAM file

This controls the addition of the input parameter and its associated value. Set this property to the argument value required.

property m

Approximately the maximum required memory

This controls the addition of the -m parameter and its associated value. Set this property to the argument value required.

property n

Sort by read names rather
than by chromosomal coordinates

This property controls the addition of the -n switch, treat this property as a boolean.

property o

(file) Write the final sorted output to FILE,
rather than to standard output

This controls the addition of the -o parameter and its associated value. Set this property to the argument value required.

class Bio.Sequencing.Applications.**SamtoolsTargetcutCommandline**(*cmd='samtools'*, ***kwargs*)

Bases: *AbstractCommandline*

Command line wrapper for samtools targetcut.

This command identifies target regions by examining the continuity of read depth, computes haploid consensus sequences of targets and outputs a SAM with each sequence corresponding to a target, equivalent to:

```
$ samtools targetcut [-Q minBaseQ] [-i inPenalty] [-0 em0]
                     [-1 em1] [-2 em2] [-f ref] <in.bam>
```

See http://samtools.sourceforge.net/samtools.shtml for more details

Examples

```
>>> from Bio.Sequencing.Applications import SamtoolsTargetcutCommandline
>>> input_bam = "/path/to/aln.bam"
>>> samtools_targetcut_cmd = SamtoolsTargetcutCommandline(input_bam=input_bam)
>>> print(samtools_targetcut_cmd)
samtools targetcut /path/to/aln.bam
```

__init__(*cmd='samtools'*, ***kwargs*)

Initialize the class.

property Q

Minimum Base Quality

This controls the addition of the -Q parameter and its associated value. Set this property to the argument value required.

__annotations__ = {}

property em0

This controls the addition of the -0 parameter and its associated value. Set this property to the argument value required.

property em1

This controls the addition of the -1 parameter and its associated value. Set this property to the argument value required.

property em2

This controls the addition of the -2 parameter and its associated value. Set this property to the argument value required.

property f

Reference Filename

This controls the addition of the -f parameter and its associated value. Set this property to the argument value required.

property i

Insertion Penalty

This controls the addition of the -i parameter and its associated value. Set this property to the argument value required.

property in_bam

Input file

This controls the addition of the input parameter and its associated value. Set this property to the argument value required.

Submodules

Bio.Sequencing.Ace module

Parser for ACE files output by PHRAP.

Written by Frank Kauff (fkauff@duke.edu) and Cymon J. Cox (cymon@duke.edu)

Usage:

There are two ways of reading an ace file:

1. The function 'read' reads the whole file at once;
2. The function 'parse' reads the file contig after contig.

First option, parse whole ace file at once:

```
from Bio.Sequencing import Ace
acefilerecord = Ace.read(open('my_ace_file.ace'))
```

This gives you:

- acefilerecord.ncontigs (the number of contigs in the ace file)
- acefilerecord.nreads (the number of reads in the ace file)
- acefilerecord.contigs[] (one instance of the Contig class for each contig)

The Contig class holds the info of the CO tag, CT and WA tags, and all the reads used for this contig in a list of instances of the Read class, e.g.:

```
contig3 = acefilerecord.contigs[2]
read4 = contig3.reads[3]
RD_of_read4 = read4.rd
DS_of_read4 = read4.ds
```

CT, WA, RT tags from the end of the file can appear anywhere are automatically sorted into the right place.

see _RecordConsumer for details.

The second option is to iterate over the contigs of an ace file one by one in the usual way:

```
from Bio.Sequencing import Ace
contigs = Ace.parse(open('my_ace_file.ace'))
for contig in contigs:
    print(contig.name)
    ...
```

Please note that for memory efficiency, when using the iterator approach, only one contig is kept in memory at once. However, there can be a footer to the ACE file containing WA, CT, RT or WR tags which contain additional meta-data on the contigs. Because the parser doesn't see this data until the final record, it cannot be added to the appropriate records. Instead these tags will be returned with the last contig record. Thus an ace file does not entirerly suit the concept of iterating. If WA, CT, RT, WR tags are needed, the 'read' function rather than the 'parse' function might be more appropriate.

class Bio.Sequencing.Ace.**rd**

Bases: object

RD (reads), store a read with its name, sequence etc.

The location and strand each read is mapped to is held in the AF lines.

__init__()

Initialize the class.

class Bio.Sequencing.Ace.**qa**(*line=None*)

Bases: object

QA (read quality), including which part if any was used as the consensus.

__init__(*line=None*)

Initialize the class.

class Bio.Sequencing.Ace.**ds**(*line=None*)

Bases: object

DS lines, include file name of a read's chromatogram file.

__init__(*line=None*)

Initialize the class.

class Bio.Sequencing.Ace.**af**(*line=None*)

Bases: object

AF lines, define the location of the read within the contig.

Note attribute coru is short for complemented (C) or uncomplemented (U), since the strand information is stored in an ACE file using either the C or U character.

__init__(*line=None*)

Initialize the class.

class Bio.Sequencing.Ace.**bs**(*line=None*)

Bases: object

BS (base segment), which read was chosen as the consensus at each position.

__init__(*line=None*)

Initialize the class.

class Bio.Sequencing.Ace.**rt**(*line=None*)

Bases: object

RT (transient read tags), generated by crossmatch and phrap.

__init__(*line=None*)

Initialize the class.

class Bio.Sequencing.Ace.**ct**(*line=None*)

Bases: object

CT (consensus tags).

__init__(*line=None*)

Initialize the class.

class Bio.Sequencing.Ace.**wa**(*line=None*)

Bases: object

WA (whole assembly tag), holds the assembly program name, version, etc.

__init__(*line=None*)

Initialize the class.

class Bio.Sequencing.Ace.**wr**(*line=None*)

Bases: object

WR lines.

__init__(*line=None*)

Initialize the class.

class Bio.Sequencing.Ace.**Reads**(*line=None*)

Bases: object

Holds information about a read supporting an ACE contig.

__init__(*line=None*)

Initialize the class.

class Bio.Sequencing.Ace.**Contig**(*line=None*)

Bases: object

Holds information about a contig from an ACE record.

__init__(*line=None*)

Initialize the class.

Bio.Sequencing.Ace.**parse**(*source*)

Iterate of ACE file contig by contig.

Argument source is a file-like object or a path to a file.

This function returns an iterator that allows you to iterate over the ACE file record by record:

```
records = parse(source)
for record in records:
    # do something with the record
```

where each record is a Contig object.

class Bio.Sequencing.Ace.**ACEFileRecord**

Bases: object

Holds data of an ACE file.

__init__()

Initialize the class.

sort()

Sorts wr, rt and ct tags into the appropriate contig / read instance, if possible.

Bio.Sequencing.Ace.**read**(*handle*)

Parse a full ACE file into a list of contigs.

Bio.Sequencing.Phd module

Parser for PHD files output by PHRED and used by PHRAP and CONSED.

This module can be used directly, which will return Record objects containing all the original data in the file.

Alternatively, using Bio.SeqIO with the "phd" format will call this module internally. This will give SeqRecord objects for each contig sequence.

class Bio.Sequencing.Phd.**Record**

Bases: object

Hold information from a PHD file.

__init__()

Initialize the class.

Bio.Sequencing.Phd.**read**(*source*)

Read one PHD record from the file and return it as a Record object.

Argument source is a file-like object opened in text mode, or a path to a file.

This function reads PHD file data line by line from the source, and returns a single Record object. A ValueError is raised if more than one record is found in the file.

Bio.Sequencing.Phd.**parse**(*source*)

Iterate over a file yielding multiple PHD records.

Argument source is a file-like object opened in text mode, or a path to a file.

The data is read line by line from the source.

Typical usage:

```
records = parse(handle)
for record in records:
    # do something with the record object
```

Module contents

Code to deal with various programs for sequencing and assembly.

This code deals with programs such as Phred, Phrap and Consed – which provide utilities for calling bases from sequencing reads, and assembling sequences into contigs.

28.1.32 Bio.SwissProt package

Submodules

Bio.SwissProt.KeyWList module

Code to parse the keywlist.txt file from SwissProt/UniProt.

See:

- https://www.uniprot.org/docs/keywlist.txt

Classes:

- Record Stores the information about one keyword or one category in the keywlist.txt file.

Functions:

- parse Parses the keywlist.txt file and returns an iterator to the records it contains.

class Bio.SwissProt.KeyWList.**Record**

Bases: dict

Store information of one keyword or category from the keywords list.

This record stores the information of one keyword or category in the keywlist.txt as a Python dictionary. The keys in this dictionary are the line codes that can appear in the keywlist.txt file:

```
---------  --------------------------     ----------------------
Line code  Content                        Occurrence in an entry
---------  --------------------------     ----------------------
ID         Identifier (keyword)           Once; starts a keyword entry
IC         Identifier (category)          Once; starts a category entry
AC         Accession (KW-xxxx)            Once
DE         Definition                     Once or more
SY         Synonyms                       Optional; once or more
GO         Gene ontology (GO) mapping     Optional; once or more
HI         Hierarchy                      Optional; once or more
WW         Relevant WWW site              Optional; once or more
CA         Category                       Once per keyword entry; absent
                                          in category entries
```

__init__()

Initialize the class.

Bio.SwissProt.KeyWList.**parse**(*handle*)

Parse the keyword list from file handle.

Returns a generator object which yields keyword entries as Bio.SwissProt.KeyWList.Record() object.

Module contents

Code to work with the sprotXX.dat file from SwissProt.

https://web.expasy.org/docs/userman.html

Classes:

- Record Holds SwissProt data.
- Reference Holds reference data from a SwissProt record.

Functions:

- read Read one SwissProt record
- parse Read multiple SwissProt records

exception Bio.SwissProt.**SwissProtParserError**(**args*, *line=None*)

Bases: ValueError

An error occurred while parsing a SwissProt file.

__init__(**args*, *line=None*)

Create a SwissProtParserError object with the offending line.

class Bio.SwissProt.**Record**

Bases: object

Holds information from a SwissProt record.

Attributes:

- entry_name Name of this entry, e.g. RL1_ECOLI.
- data_class Either 'STANDARD' or 'PRELIMINARY'.
- molecule_type Type of molecule, 'PRT',
- sequence_length Number of residues.
- accessions List of the accession numbers, e.g. ['P00321']
- created A tuple of (date, release).
- sequence_update A tuple of (date, release).
- annotation_update A tuple of (date, release).
- description Free-format description.
- **gene_name A list of dictionaries with keys 'Name', 'Synonyms',** 'OrderedLocusNames' and 'ORFNames'.
- organism The source of the sequence.
- organelle The origin of the sequence.
- organism_classification The taxonomy classification. List of strings. (http://www.ncbi.nlm.nih.gov/Taxonomy/)

- taxonomy_id A list of NCBI taxonomy id's.
- host_organism A list of names of the hosts of a virus, if any.
- host_taxonomy_id A list of NCBI taxonomy id's of the hosts, if any.
- references List of Reference objects.
- comments List of strings.
- cross_references List of tuples (db, id1[, id2][, id3]). See the docs.
- keywords List of the keywords.
- features List of tuples (key name, from, to, description). from and to can be either integers for the residue numbers, '<', '>', or '?'
- protein_existence Numerical value describing the evidence for the existence of the protein.
- seqinfo tuple of (length, molecular weight, CRC32 value)
- sequence The sequence.

Examples

```
>>> from Bio import SwissProt
>>> example_filename = "SwissProt/P68308.txt"
>>> with open(example_filename) as handle:
...     records = SwissProt.parse(handle)
...     for record in records:
...         print(record.entry_name)
...         print(record.accessions)
...         print(record.keywords)
...         print(record.organism)
...         print(record.sequence[:20] + "...")
...
NU3M_BALPH
['P68308', 'P24973']
['Electron transport', 'Membrane', 'Mitochondrion', 'Mitochondrion inner membrane',
↪'NAD', 'Respiratory chain', 'Translocase', 'Transmembrane', 'Transmembrane helix',
↪ 'Transport', 'Ubiquinone']
Balaenoptera physalus (Fin whale) (Balaena physalus).
MNLLLTLLTNTTLALLLVFI...
```

__init__()

Initialize the class.

class Bio.SwissProt.**Reference**

Bases: object

Holds information from one reference in a SwissProt entry.

Attributes:

- number Number of reference in an entry.
- evidence Evidence code. List of strings.
- positions Describes extent of work. List of strings.
- comments Comments. List of (token, text).

- references References. List of (dbname, identifier).
- authors The authors of the work.
- title Title of the work.
- location A citation for the work.

__init__()

Initialize the class.

class Bio.SwissProt.**FeatureTable**(*location=None*, *type=''*, *id='<unknown id>'*, *qualifiers=None*, *sub_features=None*)

Bases: *SeqFeature*

Stores feature annotations for specific regions of the sequence.

This is a subclass of SeqFeature, defined in Bio.SeqFeature, where the attributes are used as follows:

- `location`: location of the feature on the canonical or isoform sequence; the location is stored as an instance of SimpleLocation, defined in Bio.SeqFeature, with the ref attribute set to the isoform ID referring to the canonical or isoform sequence on which the feature is defined
- `id`: unique and stable identifier (FTId), only provided for features belonging to the types CARBOHYD, CHAIN, PEPTIDE, PROPEP, VARIANT, or VAR_SEQ
- `type`: indicates the type of feature, as defined by the UniProt Knowledgebase documentation:
  - ACT_SITE: amino acid(s) involved in the activity of an enzyme
  - BINDING: binding site for any chemical group
  - CARBOHYD: glycosylation site; an FTId identifier to the GlyConnect database is provided if annotated there
  - CA_BIND: calcium-binding region
  - CHAIN: polypeptide chain in the mature protein
  - COILED: coiled-coil region
  - COMPBIAS: compositionally biased region
  - CONFLICT: different sources report differing sequences
  - CROSSLNK: posttransationally formed amino acid bond
  - DISULFID: disulfide bond
  - DNA_BIND: DNA-binding region
  - DOMAIN: domain, defined as a specific combination of secondary structures organized into a characteristic three-dimensional structure or fold
  - INIT_MET: initiator methionine
  - INTRAMEM: region located in a membrane without crossing it
  - HELIX: alpha-, 3(10)-, or pi-helix secondary structure
  - LIPID: covalent binding of a lipid moiety
  - METAL: binding site for a metal ion
  - MOD_RES: posttranslational modification (PTM) of a residue, annotated by the controlled vocabulary defined by the ptmlist.txt document on the UniProt website
  - MOTIF: short sequence motif of biological interest

- MUTAGEN: site experimentally altered by mutagenesis
- NON_CONS: non-consecutive residues
- NON_STD: non-standard amino acid
- NON_TER: the residue at an extremity of the sequence is not the terminal residue
- NP_BIND: nucleotide phosphate-binding region
- PEPTIDE: released active mature polypeptide
- PROPEP: any processed propeptide
- REGION: region of interest in the sequence
- REPEAT: internal sequence repetition
- SIGNAL: signal sequence (prepeptide)
- SITE: amino-acid site of interest not represented by another feature key
- STRAND: beta-strand secondary structure; either a hydrogen-bonded extended beta strand or a residue in an isolated beta-bridge
- TOPO_DOM: topological domain
- TRANSIT: transit peptide (mitochondrion, chloroplast, thylakoid, cyanelle, peroxisome, etc.)
- TRANSMEM: transmembrane region
- TURN: H-bonded turn (3-, 4-, or 5-turn)
- UNSURE: uncertainties in the sequence
- VARIANT: sequence variant; an FTId is provided for protein sequence variants of Hominidae (great apes and humans)
- VAR_SEQ: sequence variant produced by alternative splicing, alternative promoter usage, alternative initiation, or ribosomal frameshifting
- ZN_FING: zinc finger region

`qualifiers`: a dictionary of additional information, which may include the feature evidence and free-text notes. While SwissProt includes the feature identifier code (FTId) as a qualifier, it is stored as the attribute ID of the FeatureTable object.

__annotations__ = {}

Bio.SwissProt.**parse**(*source*)

Read multiple SwissProt records from file.

Argument source is a file-like object or a path to a file.

Returns a generator object which yields Bio.SwissProt.Record() objects.

Bio.SwissProt.**read**(*source*)

Read one SwissProt record from file.

Argument source is a file-like object or a path to a file.

Returns a Record() object.

28.1.33 Bio.TogoWS package

Module contents

Provides code to access the TogoWS integrated websevices of DBCLS, Japan.

This module aims to make the TogoWS (from DBCLS, Japan) easier to use. See: http://togows.dbcls.jp/

The TogoWS REST service provides simple access to a range of databases, acting as a proxy to shield you from all the different provider APIs. This works using simple URLs (which this module will construct for you). For more details, see http://togows.dbcls.jp/site/en/rest.html

The functionality is somewhat similar to Biopython's Bio.Entrez module which provides access to the NCBI's Entrez Utilities (E-Utils) which also covers a wide range of databases.

Currently TogoWS does not provide any usage guidelines (unlike the NCBI whose requirements are reasonably clear). To avoid risking overloading the service, Biopython will only allow three calls per second.

The TogoWS SOAP service offers a more complex API for calling web services (essentially calling remote functions) provided by DDBJ, KEGG and PDBj. For example, this allows you to run a remote BLAST search at the DDBJ. This is not yet covered by this module, however there are lots of Python examples on the TogoWS website using the SOAPpy python library. See: http://togows.dbcls.jp/site/en/soap.html http://soapy.sourceforge.net/

Bio.TogoWS.**entry**(*db*, *id*, *format=None*, *field=None*)

Call TogoWS 'entry' to fetch a record.

Arguments:

- db - database (string), see list below.
- id - identifier (string) or a list of identifiers (either as a list of strings or a single string with comma separators).
- format - return data file format (string), options depend on the database e.g. "xml", "json", "gff", "fasta", "ttl" (RDF Turtle)
- field - specific field from within the database record (string) e.g. "au" or "authors" for pubmed.

At the time of writing, this includes the following:

```
KEGG: compound, drug, enzyme, genes, glycan, orthology, reaction,
      module, pathway
DDBj: ddbj, dad, pdb
NCBI: nuccore, nucest, nucgss, nucleotide, protein, gene, onim,
      homologue, snp, mesh, pubmed
EBI:  embl, uniprot, uniparc, uniref100, uniref90, uniref50
```

For the current list, please see http://togows.dbcls.jp/entry/

This function is essentially equivalent to the NCBI Entrez service EFetch, available in Biopython as Bio.Entrez.efetch(...), but that does not offer field extraction.

Bio.TogoWS.**search_count**(*db*, *query*)

Call TogoWS search count to see how many matches a search gives.

Arguments:

- db - database (string), see http://togows.dbcls.jp/search
- query - search term (string)

You could then use the count to download a large set of search results in batches using the offset and limit options to Bio.TogoWS.search(). In general however the Bio.TogoWS.search_iter() function is simpler to use.

Bio.TogoWS.**search_iter**(*db*, *query*, *limit=None*, *batch=100*)

Call TogoWS search iterating over the results (generator function).

Arguments:

- db - database (string), see http://togows.dbcls.jp/search
- query - search term (string)
- limit - optional upper bound on number of search results
- batch - number of search results to pull back each time talk to TogoWS (currently limited to 100).

You would use this function within a for loop, e.g.

```
>>> from Bio import TogoWS
>>> for id in TogoWS.search_iter("pubmed", "diabetes+human", limit=10):
...     print("PubMed ID: %s" %id) # maybe fetch data with entry?
PubMed ID: ...
```

Internally this first calls the Bio.TogoWS.search_count() and then uses Bio.TogoWS.search() to get the results in batches.

Bio.TogoWS.**search**(*db*, *query*, *offset=None*, *limit=None*, *format=None*)

Call TogoWS search.

This is a low level wrapper for the TogoWS search function, which can return results in a several formats. In general, the search_iter function is more suitable for end users.

Arguments:

- db - database (string), see http://togows.dbcls.jp/search/
- query - search term (string)
- offset, limit - optional integers specifying which result to start from (1 based) and the number of results to return.
- format - return data file format (string), e.g. "json", "ttl" (RDF) By default plain text is returned, one result per line.

At the time of writing, TogoWS applies a default count limit of 100 search results, and this is an upper bound. To access more results, use the offset argument or the search_iter(...) function.

TogoWS supports a long list of databases, including many from the NCBI (e.g. "ncbi-pubmed" or "pubmed", "ncbi-genbank" or "genbank", and "ncbi-taxonomy"), EBI (e.g. "ebi-ebml" or "embl", "ebi-uniprot" or "uniprot, "ebi-go"), and KEGG (e.g. "kegg-compound" or "compound"). For the current list, see http://togows.dbcls.jp/search/

The NCBI provide the Entrez Search service (ESearch) which is similar, available in Biopython as the Bio.Entrez.esearch() function.

See also the function Bio.TogoWS.search_count() which returns the number of matches found, and the Bio.TogoWS.search_iter() function which allows you to iterate over the search results (taking care of batching for you).

Bio.TogoWS.**convert**(*data*, *in_format*, *out_format*)

Call TogoWS for file format conversion.

Arguments:

- data - string or handle containing input record(s)
- in_format - string describing the input file format (e.g. "genbank")
- out_format - string describing the requested output format (e.g. "fasta")

For a list of supported conversions (e.g. "genbank" to "fasta"), see http://togows.dbcls.jp/convert/

Note that Biopython has built in support for conversion of sequence and alignnent file formats (functions Bio.SeqIO.convert and Bio.AlignIO.convert)

28.1.34 Bio.UniGene package

Module contents

Parse Unigene flat file format files such as the Hs.data file.

Here is an overview of the flat file format that this parser deals with:

Line types/qualifiers:

```
ID          UniGene cluster ID
TITLE       Title for the cluster
GENE        Gene symbol
CYTOBAND    Cytological band
EXPRESS     Tissues of origin for ESTs in cluster
RESTR_EXPR  Single tissue or development stage contributes
            more than half the total EST frequency for this gene.
GNM_TERMINUS genomic confirmation of presence of a 3' terminus;
            T if a non-templated polyA tail is found among
            a cluster's sequences; else
            I if templated As are found in genomic sequence or
            S if a canonical polyA signal is found on
              the genomic sequence
GENE_ID     Entrez gene identifier associated with at least one
            sequence in this cluster;
            to be used instead of LocusLink.
LOCUSLINK   LocusLink identifier associated with at least one
            sequence in this cluster;
            deprecated in favor of GENE_ID
HOMOL       Homology;
CHROMOSOME  Chromosome.  For plants, CHROMOSOME refers to mapping
            on the arabidopsis genome.
STS         STS
     ACC=        GenBank/EMBL/DDBJ accession number of STS
                 [optional field]
     UNISTS=     identifier in NCBI's UNISTS database
TXMAP       Transcript map interval
     MARKER=     Marker found on at least one sequence in this
                 cluster
     RHPANEL=    Radiation Hybrid panel used to place marker
PROTSIM     Protein Similarity data for the sequence with
            highest-scoring protein similarity in this cluster
     ORG=        Organism
     PROTGI=     Sequence GI of protein
```

(continues on next page)

(continued from previous page)

```
     PROTID=      Sequence ID of protein
     PCT=         Percent alignment
     ALN=         length of aligned region (aa)
SCOUNT       Number of sequences in the cluster
SEQUENCE     Sequence
     ACC=         GenBank/EMBL/DDBJ accession number of sequence
     NID=         Unique nucleotide sequence identifier (gi)
     PID=         Unique protein sequence identifier (used for
                  non-ESTs)
     CLONE=       Clone identifier (used for ESTs only)
     END=         End (5'/3') of clone insert read (used for
                  ESTs only)
     LID=         Library ID; see Hs.lib.info for library name
                  and tissue
     MGC=         5' CDS-completeness indicator; if present, the
                  clone associated with this sequence is believed
                  CDS-complete. A value greater than 511 is the gi
                  of the CDS-complete mRNA matched by the EST,
                  otherwise the value is an indicator of the
                  reliability of the test indicating CDS
                  completeness; higher values indicate more
                  reliable CDS-completeness predictions.
    SEQTYPE=      Description of the nucleotide sequence.
                  Possible values are mRNA, EST and HTC.
    TRACE=        The Trace ID of the EST sequence, as provided by
                  NCBI Trace Archive
```

class Bio.UniGene.**SequenceLine**(*text=None*)

Bases: object

Store the information for one SEQUENCE line from a Unigene file.

Initialize with the text part of the SEQUENCE line, or nothing.

Attributes and descriptions (access as LOWER CASE):

- ACC= GenBank/EMBL/DDBJ accession number of sequence
- NID= Unique nucleotide sequence identifier (gi)
- PID= Unique protein sequence identifier (used for non-ESTs)
- CLONE= Clone identifier (used for ESTs only)
- END= End (5'/3') of clone insert read (used for ESTs only)
- LID= Library ID; see Hs.lib.info for library name and tissue
- MGC= 5' CDS-completeness indicator; if present, the clone associated with this sequence is believed CDS-complete. A value greater than 511 is the gi of the CDS-complete mRNA matched by the EST, otherwise the value is an indicator of the reliability of the test indicating CDS completeness; higher values indicate more reliable CDS-completeness predictions.
- SEQTYPE= Description of the nucleotide sequence. Possible values are mRNA, EST and HTC.
- TRACE= The Trace ID of the EST sequence, as provided by NCBI Trace Archive

__init__(*text=None*)

Initialize the class.

__repr__()

Return UniGene SequenceLine object as a string.

class Bio.UniGene.**ProtsimLine**(*text=None*)

Bases: object

Store the information for one PROTSIM line from a Unigene file.

Initialize with the text part of the PROTSIM line, or nothing.

Attributes and descriptions (access as LOWER CASE) ORG= Organism PROTGI= Sequence GI of protein PROTID= Sequence ID of protein PCT= Percent alignment ALN= length of aligned region (aa)

__init__(*text=None*)

Initialize the class.

__repr__()

Return UniGene ProtsimLine object as a string.

class Bio.UniGene.**STSLine**(*text=None*)

Bases: object

Store the information for one STS line from a Unigene file.

Initialize with the text part of the STS line, or nothing.

Attributes and descriptions (access as LOWER CASE)

ACC= GenBank/EMBL/DDBJ accession number of STS [optional field] UNISTS= identifier in NCBI's UNISTS database

__init__(*text=None*)

Initialize the class.

__repr__()

Return UniGene STSLine object as a string.

class Bio.UniGene.**Record**

Bases: object

Store a Unigene record.

Here is what is stored:

```
self.ID           = ''  # ID line
self.species      = ''  # Hs, Bt, etc.
self.title        = ''  # TITLE line
self.symbol       = ''  # GENE line
self.cytoband     = ''  # CYTOBAND line
self.express      = []  # EXPRESS line, parsed on ';'
                        # Will be an array of strings
self.restr_expr   = ''  # RESTR_EXPR line
self.gnm_terminus = ''  # GNM_TERMINUS line
self.gene_id      = ''  # GENE_ID line
self.locuslink    = ''  # LOCUSLINK line
self.homol        = ''  # HOMOL line
self.chromosome   = ''  # CHROMOSOME line
self.protsim      = []  # PROTSIM entries, array of Protsims
                        # Type ProtsimLine
```

(continues on next page)

(continued from previous page)

```
self.sequence     = []  # SEQUENCE entries, array of Sequence entries
                        # Type SequenceLine
self.sts          = []  # STS entries, array of STS entries
                        # Type STSLine
self.txmap        = []  # TXMAP entries, array of TXMap entries
```

__init__()

Initialize the class.

__repr__()

Represent the UniGene Record object as a string for debugging.

Bio.UniGene.**parse**(*handle*)

Read and load a UniGene records, for files containing multiple records.

Bio.UniGene.**read**(*handle*)

Read and load a UniGene record, one record per file.

28.1.35 Bio.UniProt package

Submodules

Bio.UniProt.GOA module

Parsers for the GAF, GPA and GPI formats from UniProt-GOA.

Uniprot-GOA README + GAF format description: ftp://ftp.ebi.ac.uk/pub/databases/GO/goa/UNIPROT/README

Gene Association File, GAF formats: http://geneontology.org/docs/go-annotation-file-gaf-format-2.2/ http://geneontology.org/docs/go-annotation-file-gaf-format-2.1/ http://geneontology.org/docs/go-annotation-file-gaf-format-2.0/

Gene Product Association Data (GPA format) README: http://geneontology.org/docs/gene-product-association-data-gpad-format/

Gene Product Information (GPI format) README: http://geneontology.org/docs/gene-product-information-gpi-format/

Go Annotation files are located here: ftp://ftp.ebi.ac.uk/pub/databases/GO/goa/

Bio.UniProt.GOA.**gpi_iterator**(*handle*)

Read GPI format files.

This function should be called to read a gp_information.goa_uniprot file. At the moment, there is only one format, but this may change, so this function is a placeholder a future wrapper.

Bio.UniProt.GOA.**gpa_iterator**(*handle*)

Read GPA format files.

This function should be called to read a gene_association.goa_uniprot file. Reads the first record and returns a gpa 1.1 or a gpa 1.0 iterator as needed

Bio.UniProt.GOA.**gafbyproteiniterator**(*handle*)

Iterate over records in a gene association file.

Returns a list of all consecutive records with the same DB_Object_ID This function should be called to read a gene_association.goa_uniprot file. Reads the first record and returns a gaf 2.0 or a gaf 1.0 iterator as needed

2016-04-09: added GAF 2.1 iterator & fixed bug in iterator assignment In the meantime GAF 2.1 uses the GAF 2.0 iterator

Bio.UniProt.GOA.**gafiterator**(*handle*)

Iterate over a GAF 1.0 or 2.x file.

This function should be called to read a gene_association.goa_uniprot file. Reads the first record and returns a gaf 2.x or a gaf 1.0 iterator as needed

Example: open, read, interat and filter results.

Original data file has been trimmed to ~600 rows.

Original source ftp://ftp.ebi.ac.uk/pub/databases/GO/goa/YEAST/goa_yeast.gaf.gz

```
>>> from Bio.UniProt.GOA import gafiterator, record_has
>>> Evidence = {'Evidence': set(['ND'])}
>>> Synonym = {'Synonym': set(['YA19A_YEAST', 'YAL019W-A'])}
>>> Taxon_ID = {'Taxon_ID': set(['taxon:559292'])}
>>> with open('UniProt/goa_yeast.gaf', 'r') as handle:
...     for rec in gafiterator(handle):
...         if record_has(rec, Taxon_ID) and record_has(rec, Evidence) and record_
↪has(rec, Synonym):
...             for key in ('DB_Object_Name', 'Evidence', 'Synonym', 'Taxon_ID'):
...                 print(rec[key])
...
Putative uncharacterized protein YAL019W-A
ND
['YA19A_YEAST', 'YAL019W-A']
['taxon:559292']
Putative uncharacterized protein YAL019W-A
ND
['YA19A_YEAST', 'YAL019W-A']
['taxon:559292']
Putative uncharacterized protein YAL019W-A
ND
['YA19A_YEAST', 'YAL019W-A']
['taxon:559292']
```

Bio.UniProt.GOA.**writerec**(*outrec*, *handle*, *fields=GAF20FIELDS*)

Write a single UniProt-GOA record to an output stream.

Caller should know the format version. Default: gaf-2.0 If header has a value, then it is assumed this is the first record, a header is written.

Bio.UniProt.GOA.**writebyproteinrec**(*outprotrec*, *handle*, *fields=GAF20FIELDS*)

Write a list of GAF records to an output stream.

Caller should know the format version. Default: gaf-2.0 If header has a value, then it is assumed this is the first record, a header is written. Typically the list is the one read by fafbyproteinrec, which contains all consecutive lines with the same DB_Object_ID

Bio.UniProt.GOA.**record_has**(*inrec*, *fieldvals*)

Accept a record, and a dictionary of field values.

The format is {'field_name': set([val1, val2])}. If any field in the record has a matching value, the function returns True. Otherwise, returns False.

Module contents

Code for dealing with assorted UniProt file formats and interacting with the UniProt database.

This currently include parsers for the GAF, GPA and GPI formats from UniProt-GOA as the module Bio.UniProt.GOA.

See also Bio.SwissProt and the "swiss" support in Bio.SeqIO for the legacy plain text sequence format still used in UniProt.

See also Bio.SeqIO.SwissIO for the "uniprot-xml" support in Bio.SeqIO.

`Bio.UniProt.`**`search`**(*query: str*, *fields: list[str] | None = None*, *batch_size: int = 500*) → _UniProtSearchResults

Search the UniProt database.

Consider using query syntax and query fields to refine your search.

See the API details here.

```
>>> from Bio import UniProt
>>> from itertools import islice
>>> # Get the first 10 results
>>> results = UniProt.search("(organism_id:2697049) AND (reviewed:true)")[:10]
```

Parameters

- **query** (*str*) – The query string to search UniProt with
- **fields** (*List[str], optional*) – The columns to retrieve in the results, defaults to all fields
- **batch_size** (*int*) – The number of results to retrieve in each batch, defaults to 500

Returns
An iterator over the search results

Return type
_UniProtSearchResults

28.1.36 Bio.codonalign package

Submodules

Bio.codonalign.codonalignment module

Code for dealing with Codon Alignment.

CodonAlignment class is inherited from MultipleSeqAlignment class. This is the core class to deal with codon alignment in biopython.

class `Bio.codonalign.codonalignment.`**`CodonAlignment`**(*records=''*, *name=None*)

Bases: *MultipleSeqAlignment*

Codon Alignment class that inherits from MultipleSeqAlignment.

```
>>> from Bio.SeqRecord import SeqRecord
>>> a = SeqRecord(CodonSeq("AAAACGTCG"), id="Alpha")
>>> b = SeqRecord(CodonSeq("AAA---TCG"), id="Beta")
>>> c = SeqRecord(CodonSeq("AAAAGGTGG"), id="Gamma")
```

(continues on next page)

(continued from previous page)

```
>>> print(CodonAlignment([a, b, c]))
CodonAlignment with 3 rows and 9 columns (3 codons)
AAAACGTCG Alpha
AAA---TCG Beta
AAAAGGTGG Gamma
```

__init__(*records=''*, *name=None*)

Initialize the class.

__str__()

Return a multi-line string summary of the alignment.

This output is indicated to be readable, but large alignment is shown truncated. A maximum of 20 rows (sequences) and 60 columns (20 codons) are shown, with the record identifiers. This should fit nicely on a single screen. e.g.

__getitem__(*index*)

Return a CodonAlignment object for single indexing.

__add__(*other*)

Combine two codonalignments with the same number of rows by adding them.

The method also allows to combine a CodonAlignment object with a MultipleSeqAlignment object. The following rules apply:

- CodonAlignment + CodonAlignment -> CodonAlignment
- CodonAlignment + MultipleSeqAlignment -> MultipleSeqAlignment

get_aln_length()

Get alignment length.

toMultipleSeqAlignment()

Convert the CodonAlignment to a MultipleSeqAlignment.

Return a MultipleSeqAlignment containing all the SeqRecord in the CodonAlignment using Seq to store sequences

get_dn_ds_matrix(*method='NG86'*, *codon_table=None*)

Available methods include NG86, LWL85, YN00 and ML.

Argument:

- method - Available methods include NG86, LWL85, YN00 and ML.
- codon_table - Codon table to use for forward translation.

get_dn_ds_tree(*dn_ds_method='NG86'*, *tree_method='UPGMA'*, *codon_table=None*)

Construct dn tree and ds tree.

Argument:

- dn_ds_method - Available methods include NG86, LWL85, YN00 and ML.
- tree_method - Available methods include UPGMA and NJ.

classmethod from_msa(*align*)

Convert a MultipleSeqAlignment to CodonAlignment.

Function to convert a MultipleSeqAlignment to CodonAlignment. It is the user's responsibility to ensure all the requirement needed by CodonAlignment is met.

__annotations__ = {}

Bio.codonalign.codonalignment.**mktest**(*codon_alns*, *codon_table=None*, *alpha=0.05*)

McDonald-Kreitman test for neutrality.

Implement the McDonald-Kreitman test for neutrality (PMID: 1904993) This method counts changes rather than sites (http://mkt.uab.es/mkt/help_mkt.asp).

Arguments:

- codon_alns - list of CodonAlignment to compare (each CodonAlignment object corresponds to gene sampled from a species)

Return the p-value of test result.

Bio.codonalign.codonseq module

Code for dealing with coding sequence.

CodonSeq class is inherited from Seq class. This is the core class to deal with sequences in CodonAlignment in biopython.

class Bio.codonalign.codonseq.**CodonSeq**(*data=''*, *gap_char='-'*, *rf_table=None*)

Bases: *Seq*

CodonSeq is designed to be within the SeqRecords of a CodonAlignment class.

CodonSeq is useful as it allows the user to specify reading frame when translate CodonSeq

CodonSeq also accepts codon style slice by calling get_codon() method.

Important: Ungapped CodonSeq can be any length if you specify the rf_table. Gapped CodonSeq should be a multiple of three.

```
>>> codonseq = CodonSeq("AAATTTGGGCCAAATTT", rf_table=(0,3,6,8,11,14))
>>> print(codonseq.translate())
KFGAKF
```

test get_full_rf_table method

```
>>> p = CodonSeq('AAATTTCCCGG-TGGGTTTAA', rf_table=(0, 3, 6, 9, 11, 14, 17))
>>> full_rf_table = p.get_full_rf_table()
>>> print(full_rf_table)
[0, 3, 6, 9, 12, 15, 18]
>>> print(p.translate(rf_table=full_rf_table, ungap_seq=False))
KFPPWV*
>>> p = CodonSeq('AAATTTCCCGGGAA-TTTTAA', rf_table=(0, 3, 6, 9, 14, 17))
>>> print(p.get_full_rf_table())
[0, 3, 6, 9, 12.0, 15, 18]
>>> p = CodonSeq('AAA----------TAA', rf_table=(0, 3))
>>> print(p.get_full_rf_table())
[0, 3.0, 6.0, 9.0, 12.0, 15]
```

__init__(*data=''*, *gap_char='-'*, *rf_table=None*)

Initialize the class.

get_codon(*index*)

Get the index codon from the sequence.

get_codon_num()

Return the number of codons in the CodonSeq.

translate(*codon_table=None*, *stop_symbol='*'*, *rf_table=None*, *ungap_seq=True*)

Translate the CodonSeq based on the reading frame in rf_table.

It is possible for the user to specify a rf_table at this point. If you want to include gaps in the translated sequence, this is the only way. ungap_seq should be set to true for this purpose.

toSeq()

Convert DNA to seq object.

get_full_rf_table()

Return full rf_table of the CodonSeq records.

A full rf_table is different from a normal rf_table in that it translate gaps in CodonSeq. It is helpful to construct alignment containing frameshift.

full_translate(*codon_table=None*, *stop_symbol='*'*)

Apply full translation with gaps considered.

ungap(*gap='-'*)

Return a copy of the sequence without the gap character(s).

classmethod from_seq(*seq*, *rf_table=None*)

Get codon sequence from sequence data.

__abstractmethods__ = frozenset({})

__annotations__ = {'_data': 'Union[bytes, SequenceDataAbstractBaseClass]'}

Bio.codonalign.codonseq.**cal_dn_ds**(*codon_seq1*, *codon_seq2*, *method='NG86'*, *codon_table=None*, *k=1*, *cfreq=None*)

Calculate dN and dS of the given two sequences.

Available methods:

- NG86 - Nei and Gojobori (1986) (PMID 3444411).
- LWL85 - Li et al. (1985) (PMID 3916709).
- ML - Goldman and Yang (1994) (PMID 7968486).
- YN00 - Yang and Nielsen (2000) (PMID 10666704).

Arguments:

- codon_seq1 - CodonSeq or or SeqRecord that contains a CodonSeq
- codon_seq2 - CodonSeq or or SeqRecord that contains a CodonSeq
- w - transition/transversion ratio
- cfreq - Current codon frequency vector can only be specified when you are using ML method. Possible ways of getting cfreq are: F1x4, F3x4 and F61.

Module contents

Code for dealing with Codon Alignments.

Bio.codonalign.**build**(*pro_align*, *nucl_seqs*, *corr_dict=None*, *gap_char='-'*, *unknown='X'*, *codon_table=None*, *complete_protein=False*, *anchor_len=10*, *max_score=10*)

Build a codon alignment from protein alignment and corresponding nucleotides.

Arguments:

- pro_align - a protein MultipleSeqAlignment object
- nucl_seqs - an object returned by SeqIO.parse or SeqIO.index or a collection of SeqRecord.
- corr_dict - a dict that maps protein id to nucleotide id
- complete_protein - whether the sequence begins with a start codon

Return a CodonAlignment object.

The example below answers this Biostars question: https://www.biostars.org/p/89741/

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> from Bio.codonalign import build
>>> seq1 = SeqRecord(Seq('ATGTCTCGT'), id='pro1')
>>> seq2 = SeqRecord(Seq('ATGCGT'), id='pro2')
>>> pro1 = SeqRecord(Seq('MSR'), id='pro1')
>>> pro2 = SeqRecord(Seq('M-R'), id='pro2')
>>> aln = MultipleSeqAlignment([pro1, pro2])
>>> codon_aln = build(aln, [seq1, seq2])
>>> print(codon_aln)
CodonAlignment with 2 rows and 9 columns (3 codons)
ATGTCTCGT pro1
ATG---CGT pro2
```

28.1.37 Bio.motifs package

Subpackages

Bio.motifs.applications package

Module contents

Motif command line tool wrappers (OBSOLETE).

We have decided to remove this module in future, and instead recommend building your command and invoking it via the subprocess module directly.

Bio.motifs.jaspar package

Submodules

Bio.motifs.jaspar.db module

Provides read access to a JASPAR5 formatted database.

This modules requires MySQLdb to be installed.

Example, substitute the your database credentials as appropriate:

```
    from Bio.motifs.jaspar.db import JASPAR5
    JASPAR_DB_HOST = "hostname.example.org"
    JASPAR_DB_NAME = "JASPAR2018"
    JASPAR_DB_USER = "guest"
    JASPAR_DB_PASS = "guest"

    jdb = JASPAR5(
        host=JASPAR_DB_HOST,
        name=JASPAR_DB_NAME,
        user=JASPAR_DB_USER,
        password=JASPAR_DB_PASS
    )
    ets1 = jdb.fetch_motif_by_id('MA0098')
    print(ets1)
TF name ETS1
Matrix ID   MA0098.3
Collection  CORE
TF class    ['Tryptophan cluster factors']
TF family   ['Ets-related factors']
Species 9606
Taxonomic group vertebrates
Accession   ['P14921']
Data type used  HT-SELEX
Medline 20517297
PAZAR ID    TF0000070
Comments    Data is from Taipale HTSELEX DBD (2013)
Matrix:
        0      1      2      3      4      5      6      7      8      9
A: 2683.00 180.00 425.00   0.00   0.00 2683.00 2683.00 1102.00  89.00 803.00
C: 210.00 2683.00 2683.00  21.00   0.00   0.00   9.00  21.00 712.00 401.00
G: 640.00 297.00   7.00 2683.00 2683.00   0.00  31.00 1580.00 124.00 1083.00
T: 241.00  22.00   0.00   0.00  12.00   0.00 909.00  12.00 1970.00 396.00

    motifs = jdb.fetch_motifs(
        collection = 'CORE',
        tax_group = ['vertebrates', 'insects'],
        tf_class = 'Homeo domain factors',
        tf_family = ['TALE-type homeo domain factors', 'POU domain factors'],
        min_ic = 12
    )
    for motif in motifs:
        pass # do something with the motif
```

class Bio.motifs.jaspar.db.**JASPAR5**(*host=None*, *name=None*, *user=None*, *password=None*)

Bases: object

Class representing a JASPAR5 database.

Class representing a JASPAR5 DB. The methods within are loosely based on the perl TFBS::DB::JASPAR5 module.

Note: We will only implement reading of JASPAR motifs from the DB. Unlike the perl module, we will not attempt to implement any methods to store JASPAR motifs or create a new DB at this time.

__init__(*host=None*, *name=None*, *user=None*, *password=None*)

Construct a JASPAR5 instance and connect to specified DB.

Arguments:

- host - host name of the the JASPAR DB server
- name - name of the JASPAR database
- user - user name to connect to the JASPAR DB
- password - JASPAR DB password

__str__()

Return a string representation of the JASPAR5 DB connection.

fetch_motif_by_id(*id*)

Fetch a single JASPAR motif from the DB by its JASPAR matrix ID.

Example id 'MA0001.1'.

Arguments:

- **id - JASPAR matrix ID. This may be a fully specified ID including**
  the version number (e.g. MA0049.2) or just the base ID (e.g. MA0049). If only a base ID is provided, the latest version is returned.

Returns:

- A Bio.motifs.jaspar.Motif object

NOTE: The perl TFBS module allows you to specify the type of matrix to return (PFM, PWM, ICM) but matrices are always stored in JASPAR as PFMs so this does not really belong here. Once a PFM is fetched the pwm() and pssm() methods can be called to return the normalized and log-odds matrices.

fetch_motifs_by_name(*name*)

Fetch a list of JASPAR motifs from a JASPAR DB by the given TF name(s).

Arguments: name - a single name or list of names Returns: A list of Bio.motifs.jaspar.Motif objects

Notes: Names are not guaranteed to be unique. There may be more than one motif with the same name. Therefore even if name specifies a single name, a list of motifs is returned. This just calls self.fetch_motifs(collection = None, tf_name = name).

This behaviour is different from the TFBS perl module's get_Matrix_by_name() method which always returns a single matrix, issuing a warning message and returning the first matrix retrieved in the case where multiple matrices have the same name.

fetch_motifs(*collection=JASPAR_DFLT_COLLECTION*, *tf_name=None*, *tf_class=None*, *tf_family=None*, *matrix_id=None*, *tax_group=None*, *species=None*, *pazar_id=None*, *data_type=None*, *medline=None*, *min_ic=0*, *min_length=0*, *min_sites=0*, *all=False*, *all_versions=False*)

Fetch jaspar.Record (list) of motifs using selection criteria.

Arguments:

```
Except where obvious, all selection criteria arguments may be
specified as a single value or a list of values. Motifs must
meet ALL the specified selection criteria to be returned with
the precedent exceptions noted below.

all          - Takes precedent of all other selection criteria.
               Every motif is returned. If 'all_versions' is also
               specified, all versions of every motif are returned,
               otherwise just the latest version of every motif is
               returned.
matrix_id    - Takes precedence over all other selection criteria
               except 'all'.  Only motifs with the given JASPAR
               matrix ID(s) are returned. A matrix ID may be
               specified as just a base ID or full JASPAR IDs
               including version number. If only a base ID is
               provided for specific motif(s), then just the latest
               version of those motif(s) are returned unless
               'all_versions' is also specified.
collection   - Only motifs from the specified JASPAR collection(s)
               are returned. NOTE - if not specified, the collection
               defaults to CORE for all other selection criteria
               except 'all' and 'matrix_id'. To apply the other
               selection criteria across all JASPAR collections,
               explicitly set collection=None.
tf_name      - Only motifs with the given name(s) are returned.
tf_class     - Only motifs of the given TF class(es) are returned.
tf_family    - Only motifs from the given TF families are returned.
tax_group    - Only motifs belonging to the given taxonomic
               supergroups are returned (e.g. 'vertebrates',
               'insects', 'nematodes' etc.)
species      - Only motifs derived from the given species are
               returned.  Species are specified as taxonomy IDs.
data_type    - Only motifs generated with the given data type (e.g.
               ('ChIP-seq', 'PBM', 'SELEX' etc.) are returned.
               NOTE - must match exactly as stored in the database.
pazar_id     - Only motifs with the given PAZAR TF ID are returned.
medline      - Only motifs with the given medline (PubmMed IDs) are
               returned.
min_ic       - Only motifs whose profile matrices have at least this
               information content (specificty) are returned.
min_length   - Only motifs whose profiles are of at least this
               length are returned.
min_sites    - Only motifs compiled from at least these many binding
               sites are returned.
all_versions- Unless specified, just the latest version of motifs
               determined by the other selection criteria are
               returned. Otherwise all versions of the selected
               motifs are returned.
```

Returns:

- A Bio.motifs.jaspar.Record (list) of motifs.

Module contents

JASPAR2014 module.

class Bio.motifs.jaspar.**Motif**(*matrix_id*, *name*, *alphabet='ACGT'*, *alignment=None*, *counts=None*, *collection=None*, *tf_class=None*, *tf_family=None*, *species=None*, *tax_group=None*, *acc=None*, *data_type=None*, *medline=None*, *pazar_id=None*, *comment=None*)

Bases: *Motif*

A subclass of Bio.motifs.Motif used to represent a JASPAR profile.

Additional metadata information are stored if available. The metadata availability depends on the source of the JASPAR motif (a 'pfm' format file, a 'jaspar' format file or a JASPAR database).

__init__(*matrix_id*, *name*, *alphabet='ACGT'*, *alignment=None*, *counts=None*, *collection=None*, *tf_class=None*, *tf_family=None*, *species=None*, *tax_group=None*, *acc=None*, *data_type=None*, *medline=None*, *pazar_id=None*, *comment=None*)

Construct a JASPAR Motif instance.

property base_id

Return the JASPAR base matrix ID.

property version

Return the JASPAR matrix version.

__str__()

Return a string representation of the JASPAR profile.

We choose to provide only the filled metadata information.

__hash__()

Return the hash key corresponding to the JASPAR profile.

Note
We assume the unicity of matrix IDs

__eq__(*other*)

Return True if matrix IDs are the same.

__annotations__ = {}

class Bio.motifs.jaspar.**Record**

Bases: list

Represent a list of jaspar motifs.

Attributes:

- version: The JASPAR version used

__init__()

Initialize the class.

__str__()

Return a string of all motifs in the Record.

to_dict()

Return the list of matrices as a dictionary of matrices.

Bio.motifs.jaspar.**read**(*handle*, *format*)

Read motif(s) from a file in one of several different JASPAR formats.

Return the record of PFM(s). Call the appropriate routine based on the format passed.

Bio.motifs.jaspar.**write**(*motifs*, *format*)

Return the representation of motifs in "pfm" or "jaspar" format.

Bio.motifs.jaspar.**calculate_pseudocounts**(*motif*)

Calculate pseudocounts.

Computes the root square of the total number of sequences multiplied by the background nucleotide.

Bio.motifs.jaspar.**split_jaspar_id**(*id*)

Split a JASPAR matrix ID into its component.

Components are base ID and version number, e.g. 'MA0047.2' is returned as ('MA0047', 2).

Submodules

Bio.motifs.alignace module

Parsing AlignACE output files.

class Bio.motifs.alignace.**Record**

Bases: list

AlignACE record (subclass of Python list).

__init__()

Initialize the class.

Bio.motifs.alignace.**read**(*handle*)

Parse an AlignACE format handle as a Record object.

Bio.motifs.clusterbuster module

Parse Cluster Buster position frequency matrix files.

class Bio.motifs.clusterbuster.**Record**(*iterable=()*, */*)

Bases: list

Class to store the information in a Cluster Buster matrix table.

The record inherits from a list containing the individual motifs.

__str__()

Return a string representation of the motifs in the Record object.

Bio.motifs.clusterbuster.**read**(*handle*)

Read motifs in Cluster Buster position frequency matrix format from a file handle.

Cluster Buster motif format: http://zlab.bu.edu/cluster-buster/help/cis-format.html

Bio.motifs.clusterbuster.**write**(*motifs*)

Return the representation of motifs in Cluster Buster position frequency matrix format.

Bio.motifs.mast module

Module for the support of Motif Alignment and Search Tool (MAST).

class Bio.motifs.mast.**Record**

Bases: `list`

The class for holding the results from a MAST run.

A mast.Record holds data about matches between motifs and sequences. The motifs held by the Record are objects of the class meme.Motif.

The mast.Record class inherits from list, so you can access individual motifs in the record by their index. Alternatively, you can find a motif by its name:

```
>>> from Bio import motifs
>>> with open("motifs/mast.crp0.de.oops.txt.xml") as f:
...     record = motifs.parse(f, 'MAST')
>>> motif = record[0]
>>> print(motif.name)
1
>>> motif = record['1']
>>> print(motif.name)
1
```

__init__()

Initialize the class.

__getitem__(*key*)

Return the motif of index key.

Bio.motifs.mast.**read**(*handle*)

Parse a MAST XML format handle as a Record object.

Bio.motifs.matrix module

Support for various forms of sequence motif matrices.

Implementation of frequency (count) matrices, position-weight matrices, and position-specific scoring matrices.

class Bio.motifs.matrix.**GenericPositionMatrix**(*alphabet*, *values*)

Bases: `dict`

Base class for the support of position matrix operations.

__init__(*alphabet*, *values*)

Initialize the class.

__str__()

Return a string containing nucleotides and counts of the alphabet in the Matrix.

__getitem__(*key*)

Return the position matrix of index key.

property consensus

Return the consensus sequence.

property anticonsensus

Return the anticonsensus sequence.

property degenerate_consensus

Return the degenerate consensus sequence.

calculate_consensus(*substitution_matrix=None*, *plurality=None*, *identity=0*, *setcase=None*)

Return the consensus sequence (as a string) for the given parameters.

This function largely follows the conventions of the EMBOSS *cons* tool.

Arguments:

- substitution_matrix - the scoring matrix used when comparing sequences. By default, it is None, in which case we simply count the frequency of each letter. Instead of the default value, you can use the substitution matrices available in Bio.Align.substitution_matrices. Common choices are BLOSUM62 (also known as EBLOSUM62) for protein, and NUC.4.4 (also known as EDNAFULL) for nucleotides. NOTE: This has not yet been implemented.
- plurality - threshold value for the number of positive matches, divided by the total count in a column, required to reach consensus. If substitution_matrix is None, then this argument must be None, and is ignored; a ValueError is raised otherwise. If substitution_matrix is not None, then the default value of the plurality is 0.5.
- identity - number of identities, divided by the total count in a column, required to define a consensus value. If the number of identities is less than identity * total count in a column, then the undefined character ('N' for nucleotides and 'X' for amino acid sequences) is used in the consensus sequence. If identity is 1.0, then only columns of identical letters contribute to the consensus. Default value is zero.
- setcase - threshold for the positive matches, divided by the total count in a column, above which the consensus is is upper-case and below which the consensus is in lower-case. By default, this is equal to 0.5.

property gc_content

Compute the fraction GC content.

reverse_complement()

Compute reverse complement.

class Bio.motifs.matrix.FrequencyPositionMatrix(*alphabet*, *values*)

Bases: *GenericPositionMatrix*

Class for the support of frequency calculations on the Position Matrix.

normalize(*pseudocounts=None*)

Create and return a position-weight matrix by normalizing the counts matrix.

If pseudocounts is None (default), no pseudocounts are added to the counts.

If pseudocounts is a number, it is added to the counts before calculating the position-weight matrix.

Alternatively, the pseudocounts can be a dictionary with a key for each letter in the alphabet associated with the motif.

__annotations__ = {}

class Bio.motifs.matrix.**PositionWeightMatrix**(*alphabet*, *counts*)

Bases: *GenericPositionMatrix*

Class for the support of weight calculations on the Position Matrix.

__init__(*alphabet*, *counts*)

Initialize the class.

log_odds(*background=None*)

Return the Position-Specific Scoring Matrix.

The Position-Specific Scoring Matrix (PSSM) contains the log-odds scores computed from the probability matrix and the background probabilities. If the background is None, a uniform background distribution is assumed.

__annotations__ = {}

class Bio.motifs.matrix.**PositionSpecificScoringMatrix**(*alphabet*, *values*)

Bases: *GenericPositionMatrix*

Class for the support of Position Specific Scoring Matrix calculations.

calculate(*sequence*)

Return the PWM score for a given sequence for all positions.

Notes:

- the sequence can only be a DNA sequence
- the search is performed only on one strand
- if the sequence and the motif have the same length, a single number is returned
- otherwise, the result is a one-dimensional numpy array

search(*sequence*, *threshold=0.0*, *both=True*, *chunksize=10**6*)

Find hits with PWM score above given threshold.

A generator function, returning found hits in the given sequence with the pwm score higher than the threshold.

property max

Maximal possible score for this motif.

returns the score computed for the consensus sequence.

property min

Minimal possible score for this motif.

returns the score computed for the anticonsensus sequence.

property gc_content

Compute the GC-ratio.

mean(*background=None*)

Return expected value of the score of a motif.

std(*background=None*)

Return standard deviation of the score of a motif.

dist_pearson(*other*)

Return the similarity score based on pearson correlation for the given motif against self.

We use the Pearson's correlation of the respective probabilities.

dist_pearson_at(*other*, *offset*)

Return the similarity score based on pearson correlation at the given offset.

__annotations__ = {}

distribution(*background=None*, *precision=10**3*)

Calculate the distribution of the scores at the given precision.

Bio.motifs.meme module

Module for the support of MEME motif format.

Bio.motifs.meme.**read**(*handle*)

Parse the text output of the MEME program into a meme.Record object.

Examples

```
>>> from Bio.motifs import meme
>>> with open("motifs/meme.INO_up800.classic.oops.xml") as f:
...     record = meme.read(f)
>>> for motif in record:
...     for sequence in motif.alignment.sequences:
...         print(sequence.motif_name, sequence.sequence_name, sequence.sequence_id,
↪ sequence.strand, sequence.pvalue)
GSKGCATGTGAAA INO1 sequence_5 + 1.21e-08
GSKGCATGTGAAA FAS1 sequence_2 - 1.87e-08
GSKGCATGTGAAA ACC1 sequence_4 - 6.62e-08
GSKGCATGTGAAA CHO2 sequence_1 - 1.05e-07
GSKGCATGTGAAA CHO1 sequence_0 - 1.69e-07
GSKGCATGTGAAA FAS2 sequence_3 - 5.62e-07
GSKGCATGTGAAA OPI3 sequence_6 + 1.08e-06
TTGACWCYTGCYCWG CHO2 sequence_1 + 7.2e-10
TTGACWCYTGCYCWG OPI3 sequence_6 - 2.56e-08
TTGACWCYTGCYCWG ACC1 sequence_4 - 1.59e-07
TTGACWCYTGCYCWG CHO1 sequence_0 + 2.05e-07
TTGACWCYTGCYCWG FAS1 sequence_2 + 3.85e-07
TTGACWCYTGCYCWG FAS2 sequence_3 - 5.11e-07
TTGACWCYTGCYCWG INO1 sequence_5 + 8.01e-07
```

class Bio.motifs.meme.**Motif**(*alphabet=None*, *alignment=None*)

Bases: *Motif*

A subclass of Motif used in parsing MEME (and MAST) output.

This subclass defines functions and data specific to MEME motifs. This includes the motif name, the evalue for a motif, and its number of occurrences.

__init__(*alphabet=None*, *alignment=None*)

Initialize the class.

__annotations__ = {}

class Bio.motifs.meme.**Instance**(**args*, ***kwds*)

Bases: *Seq*

A class describing the instances of a MEME motif, and the data thereof.

__init__(**args*, ***kwds*)

Initialize the class.

__abstractmethods__ = frozenset({})

__annotations__ = {'_data': 'Union[bytes, SequenceDataAbstractBaseClass]'}

class Bio.motifs.meme.**Record**

Bases: list

A class for holding the results of a MEME run.

A meme.Record is an object that holds the results from running MEME. It implements no methods of its own.

The meme.Record class inherits from list, so you can access individual motifs in the record by their index. Alternatively, you can find a motif by its name:

```
>>> from Bio import motifs
>>> with open("motifs/meme.INO_up800.classic.oops.xml") as f:
...     record = motifs.parse(f, 'MEME')
>>> motif = record[0]
>>> print(motif.name)
GSKGCATGTGAAA
>>> motif = record['GSKGCATGTGAAA']
>>> print(motif.name)
GSKGCATGTGAAA
```

__init__()

Initialize the class.

__getitem__(*key*)

Return the motif of index key.

Bio.motifs.minimal module

Module for the support of MEME minimal motif format.

Bio.motifs.minimal.**read**(*handle*)

Parse the text output of the MEME program into a meme.Record object.

Examples

```
>>> from Bio.motifs import minimal
>>> with open("motifs/meme.out") as f:
...     record = minimal.read(f)
...
>>> for motif in record:
...     print(motif.name, motif.evalue)
...
1 1.1e-22
```

You can access individual motifs in the record by their index or find a motif by its name:

```
>>> from Bio import motifs
>>> with open("motifs/minimal_test.meme") as f:
...     record = motifs.parse(f, 'minimal')
...
>>> motif = record[0]
>>> print(motif.name)
KRP
>>> motif = record['IFXA']
>>> print(motif.name)
IFXA
```

This function won't retrieve instances, as there are none in minimal meme format.

class Bio.motifs.minimal.**Record**

Bases: list

Class for holding the results of a minimal MEME run.

__init__()

Initialize record class values.

__getitem__(*key*)

Return the motif of index key.

Bio.motifs.pfm module

Parse various position frequency matrix format files.

class Bio.motifs.pfm.**Record**(*iterable=()*, */*)

Bases: list

Class to store the information in a position frequency matrix table.

The record inherits from a list containing the individual motifs.

__str__()

Return a string representation of the motifs in the Record object.

Bio.motifs.pfm.**read**(*handle*, *pfm_format*)

Read motif(s) from a file in various position frequency matrix formats.

Return the record of PFM(s). Call the appropriate routine based on the format passed.

Bio.motifs.pfm.**write**(*motifs*)

Return the representation of motifs in Cluster Buster position frequency matrix format.

Bio.motifs.thresholds module

Approximate calculation of appropriate thresholds for motif finding.

class Bio.motifs.thresholds.**ScoreDistribution**(*motif=None*, *precision=10**3*, *pssm=None*, *background=None*)

Bases: object

Class representing approximate score distribution for a given motif.

Utilizes a dynamic programming approach to calculate the distribution of scores with a predefined precision. Provides a number of methods for calculating thresholds for motif occurrences.

__init__(*motif=None*, *precision=10**3*, *pssm=None*, *background=None*)

Initialize the class.

modify(*scores*, *mo_probs*, *bg_probs*)

Modify motifs and background density.

threshold_fpr(*fpr*)

Approximate the log-odds threshold which makes the type I error (false positive rate).

threshold_fnr(*fnr*)

Approximate the log-odds threshold which makes the type II error (false negative rate).

threshold_balanced(*rate_proportion=1.0*, *return_rate=False*)

Approximate log-odds threshold making FNR equal to FPR times rate_proportion.

threshold_patser()

Threshold selection mimicking the behaviour of patser (Hertz, Stormo 1999) software.

It selects such a threshold that the log(fpr)=-ic(M) note: the actual patser software uses natural logarithms instead of log_2, so the numbers are not directly comparable.

Bio.motifs.transfac module

Parsing TRANSFAC files.

class Bio.motifs.transfac.**Motif**(*alphabet='ACGT'*, *alignment=None*, *counts=None*, *instances=None*)

Bases: *Motif*, dict

Store the information for one TRANSFAC motif.

This class inherits from the Bio.motifs.Motif base class, as well as from a Python dictionary. All motif information found by the parser is stored as attributes of the base class when possible; see the Bio.motifs.Motif base class for a description of these attributes. All other information associated with the motif is stored as (key, value) pairs in the dictionary, where the key is the two-letter fields as found in the TRANSFAC file. References are an exception: These are stored in the .references attribute.

These fields are commonly found in TRANSFAC files:

```
AC:    Accession number
AS:    Accession numbers, secondary
BA:    Statistical basis
BF:    Binding factors
BS:    Factor binding sites underlying the matrix
       [sequence; SITE accession number; start position for matrix
       sequence; length of sequence used; number of gaps inserted;
       strand orientation.]
CC:    Comments
CO:    Copyright notice
DE:    Short factor description
DR:    External databases
       [database name: database accession number]
DT:    Date created/updated
HC:    Subfamilies
HP:    Superfamilies
ID:    Identifier
NA:    Name of the binding factor
OC:    Taxonomic classification
OS:    Species/Taxon
OV:    Older version
PV:    Preferred version
TY:    Type
XX:    Empty line; these are not stored in the Record.
```

References are stored in an .references attribute, which is a list of dictionaries with the following keys:

```
RN:    Reference number
RA:    Reference authors
RL:    Reference data
RT:    Reference title
RX:    PubMed ID
```

For more information, see the TRANSFAC documentation.

multiple_value_keys = {'BF', 'BS', 'CC', 'DR', 'DT', 'HC', 'HP', 'OV'}

reference_keys = {'RA', 'RL', 'RT', 'RX'}

__getitem__(*key*)

Return a new Motif object for the positions included in key.

```
>>> from Bio import motifs
>>> motif = motifs.create(["AACGCCA", "ACCGCCC", "AACTCCG"])
>>> print(motif)
AACGCCA
ACCGCCC
AACTCCG
>>> print(motif[:-1])
AACGCC
ACCGCC
AACTCC
```

__annotations__ = {}

class Bio.motifs.transfac.**Record**

Bases: list

Store the information in a TRANSFAC matrix table.

The record inherits from a list containing the individual motifs.

Attributes:

- version - The version number, corresponding to the 'VV' field in the TRANSFAC file;

__init__()

Initialize the class.

__str__()

Turn the TRANSFAC matrix into a string.

Bio.motifs.transfac.**read**(*handle*, *strict=True*)

Parse a transfac format handle into a Record object.

Bio.motifs.transfac.**write**(*motifs*)

Write the representation of a motif in TRANSFAC format.

Bio.motifs.xms module

Parse XMS motif files.

class Bio.motifs.xms.**XMSScanner**(*doc*)

Bases: object

Class for scanning XMS XML file.

__init__(*doc*)

Generate motif Record from xms document, an XML-like motif pfm file.

handle_motif(*node*)

Read the motif's name and column from the node and add the motif record.

get_property_value(*node*, *key_name*)

Extract the value of the motif's property named key_name from node.

get_acgt(*node*)

Get and return the motif's weights of A, C, G, T.

get_text(*nodelist*)

Return a string representation of the motif's properties listed on nodelist .

class Bio.motifs.xms.**Record**(*iterable=()*, */*)

Bases: list

Class to store the information in a XMS matrix table.

The record inherits from a list containing the individual motifs.

__str__()

Return a string representation of the motifs in the Record object.

Bio.motifs.xms.**read**(*handle*)

Read motifs in XMS matrix format from a file handle.

XMS is an XML format for describing regulatory motifs and PSSMs. This format was defined by Thomas Down, and used in the NestedMICA and MotifExplorer programs.

Module contents

Tools for sequence motif analysis.

Bio.motifs contains the core Motif class containing various I/O methods as well as methods for motif comparisons and motif searching in sequences. It also includes functionality for parsing output from the AlignACE, MEME, and MAST programs, as well as files in the TRANSFAC format.

Bio.motifs.**create**(*instances*, *alphabet='ACGT'*)

Create a Motif object.

Bio.motifs.**parse**(*handle*, *fmt*, *strict=True*)

Parse an output file from a motif finding program.

Currently supported formats (case is ignored):

- AlignAce: AlignAce output file format
- ClusterBuster: Cluster Buster position frequency matrix format
- XMS: XMS matrix format
- MEME: MEME output file motif
- MINIMAL: MINIMAL MEME output file motif
- MAST: MAST output file motif
- TRANSFAC: TRANSFAC database file format
- pfm-four-columns: Generic position-frequency matrix format with four columns. (cisbp, homer, homeco, neph, tiffin)
- pfm-four-rows: Generic position-frequency matrix format with four row. (scertf, yetfasco, hdpi, idmmpmm, flyfactor survey)
- pfm: JASPAR-style position-frequency matrix
- jaspar: JASPAR-style multiple PFM format
- sites: JASPAR-style sites file

As files in the pfm and sites formats contain only a single motif, it is easier to use Bio.motifs.read() instead of Bio.motifs.parse() for those.

For example:

```
>>> from Bio import motifs
>>> with open("motifs/alignace.out") as handle:
...     for m in motifs.parse(handle, "AlignAce"):
...         print(m.consensus)
...
TCTACGATTGAG
CTGCACCTAGCTACGAGTGAG
GTGCCCTAAGCATACTAGGCG
```

(continues on next page)

(continued from previous page)

```
GCCACTAGCAGAGCAGGGGGC
CGACTCAGAGGTT
CCACGCTAAGAGAAGTGCCGGAG
GCACGTCCCTGAGCA
GTCCATCGCAAAGCGTGGGGC
GAGATCAGAGGGCCG
TGGACGCGGGG
GACCAGAGCCTCGCATGGGGG
AGCGCGCGTG
GCCGGTTGCTGTTCATTAGG
ACCGACGGCAGCTAAAAGGG
GACGCCGGGGAT
CGACTCGCGCTTACAAGG
```

If strict is True (default), the parser will raise a ValueError if the file contents does not strictly comply with the specified file format.

Bio.motifs.**read**(*handle*, *fmt*, *strict=True*)

Read a motif from a handle using the specified file-format.

This supports the same formats as Bio.motifs.parse(), but only for files containing exactly one motif. For example, reading a JASPAR-style pfm file:

```
>>> from Bio import motifs
>>> with open("motifs/SRF.pfm") as handle:
...     m = motifs.read(handle, "pfm")
>>> m.consensus
Seq('GCCCATATATGG')
```

Or a single-motif MEME file,

```
>>> from Bio import motifs
>>> with open("motifs/meme.psp_test.classic.zoops.xml") as handle:
...     m = motifs.read(handle, "meme")
>>> m.consensus
Seq('GCTTATGTAA')
```

If the handle contains no records, or more than one record, an exception is raised:

```
>>> from Bio import motifs
>>> with open("motifs/alignace.out") as handle:
...     motif = motifs.read(handle, "AlignAce")
Traceback (most recent call last):
    ...
ValueError: More than one motif found in handle
```

If however you want the first motif from a file containing multiple motifs this function would raise an exception (as shown in the example above). Instead use:

```
>>> from Bio import motifs
>>> with open("motifs/alignace.out") as handle:
...     record = motifs.parse(handle, "alignace")
>>> motif = record[0]
```

(continues on next page)

(continued from previous page)

```
>>> motif.consensus
Seq('TCTACGATTGAG')
```

Use the Bio.motifs.parse(handle, fmt) function if you want to read multiple records from the handle.

If strict is True (default), the parser will raise a ValueError if the file contents does not strictly comply with the specified file format.

class Bio.motifs.**Instances**(*instances=None*, *alphabet='ACGT'*)

Bases: `list`

Class containing a list of sequences that made the motifs.

__init__(*instances=None*, *alphabet='ACGT'*)

Initialize the class.

__str__()

Return a string containing the sequences of the motif.

count()

Count nucleotides in a position.

search(*sequence*)

Find positions of motifs in a given sequence.

This is a generator function, returning found positions of motif instances in a given sequence.

reverse_complement()

Compute reverse complement of sequences.

class Bio.motifs.**Motif**(*alphabet='ACGT'*, *alignment=None*, *counts=None*, *instances=None*)

Bases: `object`

A class representing sequence motifs.

__init__(*alphabet='ACGT'*, *alignment=None*, *counts=None*, *instances=None*)

Initialize the class.

property mask

property pseudocounts

property background

__getitem__(*key*)

Return a new Motif object for the positions included in key.

```
>>> from Bio import motifs
>>> motif = motifs.create(["AACGCCA", "ACCGCCC", "AACTCCG"])
>>> print(motif)
AACGCCA
ACCGCCC
AACTCCG
>>> print(motif[:-1])
AACGCC
ACCGCC
AACTCC
```

property pwm

Calculate and return the position weight matrix for this motif.

property pssm

Calculate and return the position specific scoring matrix for this motif.

property instances

Return the sequences from which the motif was built.

__str__(*masked=False*)

Return string representation of a motif.

__len__()

Return the length of a motif.

Please use this method (i.e. invoke len(m)) instead of referring to m.length directly.

reverse_complement()

Return the reverse complement of the motif as a new motif.

property consensus

Return the consensus sequence.

property anticonsensus

Return the least probable pattern to be generated from this motif.

property degenerate_consensus

Return the degenerate consensus sequence.

Following the rules adapted from D. R. Cavener: "Comparison of the consensus sequence flanking translational start sites in Drosophila and vertebrates." Nucleic Acids Research 15(4): 1353-1361. (1987).

The same rules are used by TRANSFAC.

property relative_entropy

Return an array with the relative entropy for each column of the motif.

weblogo(*fname*, *fmt='PNG'*, *version=None*, ***kwds*)

Download and save a weblogo using the Berkeley weblogo service.

Requires an internet connection.

The version parameter is deprecated and has no effect.

The parameters from **kwds are passed directly to the weblogo server.

Currently, this method uses WebLogo version 3.3. These are the arguments and their default values passed to WebLogo 3.3; see their website at http://weblogo.threeplusone.com for more information:

```
'stack_width' : 'medium',
'stacks_per_line' : '40',
'alphabet' : 'alphabet_dna',
'ignore_lower_case' : True,
'unit_name' : "bits",
'first_index' : '1',
'logo_start' : '1',
'logo_end': str(self.length),
'composition' : "comp_auto",
'percentCG' : '',
```

(continues on next page)

(continued from previous page)

```
'scale_width' : True,
'show_errorbars' : True,
'logo_title' : '',
'logo_label' : '',
'show_xaxis': True,
'xaxis_label': '',
'show_yaxis': True,
'yaxis_label': '',
'yaxis_scale': 'auto',
'yaxis_tic_interval' : '1.0',
'show_ends' : True,
'show_fineprint' : True,
'color_scheme': 'color_auto',
'symbols0': '',
'symbols1': '',
'symbols2': '',
'symbols3': '',
'symbols4': '',
'color0': '',
'color1': '',
'color2': '',
'color3': '',
'color4': '',
```

__format__(*format_spec*)

Return a string representation of the Motif in the given format.

Currently supported formats:

- clusterbuster: Cluster Buster position frequency matrix format
- pfm : JASPAR single Position Frequency Matrix
- jaspar : JASPAR multiple Position Frequency Matrix
- transfac : TRANSFAC like files

format(*format_spec*)

Return a string representation of the Motif in the given format.

Currently supported formats:

- clusterbuster: Cluster Buster position frequency matrix format
- pfm : JASPAR single Position Frequency Matrix
- jaspar : JASPAR multiple Position Frequency Matrix
- transfac : TRANSFAC like files

Bio.motifs.**write**(*motifs*, *fmt*)

Return a string representation of motifs in the given format.

Currently supported formats (case is ignored):

- clusterbuster: Cluster Buster position frequency matrix format
- pfm : JASPAR simple single Position Frequency Matrix
- jaspar : JASPAR multiple PFM format

- transfac : TRANSFAC like files

28.1.38 Bio.phenotype package

Submodules

Bio.phenotype.phen_micro module

Classes to work with Phenotype Microarray data.

More information on the single plates can be found here: http://www.biolog.com/

Classes:

- PlateRecord - Object that contain time course data on each well of the plate, as well as metadata (if any).
- WellRecord - Object that contains the time course data of a single well
- JsonWriter - Writer of PlateRecord objects in JSON format.

Functions:

- JsonIterator - Incremental PM JSON parser, this is an iterator that returns PlateRecord objects.
- CsvIterator - Incremental PM CSV parser, this is an iterator that returns PlateRecord objects.
- _toOPM - Used internally by JsonWriter, converts PlateRecord objects in dictionaries ready to be serialized in JSON format.

class Bio.phenotype.phen_micro.**PlateRecord**(*plateid*, *wells=None*)

Bases: object

PlateRecord object for storing Phenotype Microarray plates data.

A PlateRecord stores all the wells of a particular phenotype Microarray plate, along with metadata (if any). The single wells can be accessed calling their id as an index or iterating on the PlateRecord:

```
>>> from Bio import phenotype
>>> plate = phenotype.read("phenotype/Plate.json", "pm-json")
>>> well = plate['A05']
>>> for well in plate:
...     print(well.id)
...
A01
...
```

The plate rows and columns can be queried with an indexing system similar to NumPy and other matrices:

```
>>> print(plate[1])
Plate ID: PM01
Well: 12
Rows: 1
Columns: 12
PlateRecord('WellRecord['B01'], WellRecord['B02'], WellRecord['B03'], ...,␣
↪WellRecord['B12']')
```

```
>>> print(plate[:,1])
Plate ID: PM01
Well: 8
Rows: 8
Columns: 1
PlateRecord('WellRecord['A02'], WellRecord['B02'], WellRecord['C02'], ...,␣
↪WellRecord['H02']')
```

Single WellRecord objects can be accessed using this indexing system:

```
>>> print(plate[1,2])
Plate ID: PM01
Well ID: B03
Time points: 384
Minum signal 0.00 at time 11.00
Maximum signal 76.25 at time 18.00
WellRecord('(0.0, 11.0), (0.25, 11.0), (0.5, 11.0), (0.75, 11.0), (1.0, 11.0), ...,␣
↪(95.75, 11.0)')
```

The presence of a particular well can be inspected with the "in" keyword: >>> 'A01' in plate True

All the wells belonging to a "row" (identified by the first character of the well id) in the plate can be obtained:

```
>>> for well in plate.get_row('H'):
...     print(well.id)
...
H01
H02
H03
...
```

All the wells belonging to a "column" (identified by the number of the well) in the plate can be obtained:

```
>>> for well in plate.get_column(12):
...     print(well.id)
...
A12
B12
C12
...
```

Two PlateRecord objects can be compared: if all their wells are equal the two plates are considered equal:

```
>>> plate2 = phenotype.read("phenotype/Plate.json", "pm-json")
>>> plate == plate2
True
```

Two PlateRecord object can be summed up or subtracted from each other: the the signals of each well will be summed up or subtracted. The id of the left operand will be kept:

```
>>> plate3 = plate + plate2
>>> print(plate3.id)
PM01
```

Many Phenotype Microarray plate have a "negative control" well, which can be subtracted to all wells:

```
>>> subplate = plate.subtract_control()
```

__init__(*plateid*, *wells=None*)

Initialize the class.

__getitem__(*index*)

Access part of the plate.

Depending on the indices, you can get a WellRecord object (representing a single well of the plate), or another plate (representing some part or all of the original plate).

plate[wid] gives a WellRecord (if wid is a WellRecord id) plate[r,c] gives a WellRecord plate[r] gives a row as a PlateRecord plate[r,:] gives a row as a PlateRecord plate[:,c] gives a column as a PlateRecord

plate[:] and plate[:,:] give a copy of the plate

Anything else gives a subset of the original plate, e.g. plate[0:2] or plate[0:2,:] uses only row 0 and 1 plate[:,1:3] uses only columns 1 and 2 plate[0:2,1:3] uses only rows 0 & 1 and only cols 1 & 2

```
>>> from Bio import phenotype
>>> plate = phenotype.read("phenotype/Plate.json", "pm-json")
```

You can access a well of the plate, using its id.

```
>>> w = plate['A01']
```

You can access a row of the plate as a PlateRecord using an integer index:

```
>>> first_row = plate[0]
>>> print(first_row)
Plate ID: PM01
Well: 12
Rows: 1
Columns: 12
PlateRecord('WellRecord['A01'], WellRecord['A02'], WellRecord['A03'], ...,␣
↪WellRecord['A12']')
>>> last_row = plate[-1]
>>> print(last_row)
Plate ID: PM01
Well: 12
Rows: 1
Columns: 12
PlateRecord('WellRecord['H01'], WellRecord['H02'], WellRecord['H03'], ...,␣
↪WellRecord['H12']')
```

You can also access use python's slice notation to sub-plates containing only some of the plate rows:

```
>>> sub_plate = plate[2:5]
>>> print(sub_plate)
Plate ID: PM01
Well: 36
Rows: 3
Columns: 12
PlateRecord('WellRecord['C01'], WellRecord['C02'], WellRecord['C03'], ...,␣
↪WellRecord['E12']')
```

This includes support for a step, i.e. plate[start:end:step], which can be used to select every second row:

```
>>> sub_plate = plate[::2]
```

You can also use two indices to specify both rows and columns. Using simple integers gives you the single wells. e.g.

```
>>> w = plate[3, 4]
>>> print(w.id)
D05
```

To get a single column use this syntax:

```
>>> sub_plate = plate[:, 4]
>>> print(sub_plate)
Plate ID: PM01
Well: 8
Rows: 8
Columns: 1
PlateRecord('WellRecord['A05'], WellRecord['B05'], WellRecord['C05'], ...,␣
↪WellRecord['H05']')
```

Or, to get part of a column,

```
>>> sub_plate = plate[1:3, 4]
>>> print(sub_plate)
Plate ID: PM01
Well: 2
Rows: 2
Columns: 1
PlateRecord(WellRecord['B05'], WellRecord['C05'])
```

However, in general you get a sub-plate,

```
>>> print(plate[1:5, 3:6])
Plate ID: PM01
Well: 12
Rows: 4
Columns: 3
PlateRecord('WellRecord['B04'], WellRecord['B05'], WellRecord['B06'], ...,␣
↪WellRecord['E06']')
```

This should all seem familiar to anyone who has used the NumPy array or matrix objects.

__setitem__(*key*, *value*)

__delitem__(*key*)

__iter__()

__contains__(*wellid*)

__len__()

Return the number of wells in this plate.

__eq__(*other*)

Return self==value.

__add__(*plate*)

Add another PlateRecord object.

The wells in both plates must be the same

A new PlateRecord object is returned, having the same id as the left operand.

__sub__(*plate*)

Subtract another PlateRecord object.

The wells in both plates must be the same

A new PlateRecord object is returned, having the same id as the left operand.

get_row(*row*)

Get all the wells of a given row.

A row is identified with a letter (e.g. 'A')

get_column(*column*)

Get all the wells of a given column.

A column is identified with a number (e.g. '6')

subtract_control(*control='A01'*, *wells=None*)

Subtract a 'control' well from the other plates wells.

By default the control is subtracted to all wells, unless a list of well ID is provided

The control well should belong to the plate A new PlateRecord object is returned

__repr__()

Return a (truncated) representation of the plate for debugging.

__str__()

Return a human readable summary of the record (string).

The python built in function str works by calling the object's __str__ method. e.g.

```
>>> from Bio import phenotype
>>> record = next(phenotype.parse("phenotype/Plates.csv", "pm-csv"))
>>> print(record)
Plate ID: PM01
Well: 96
Rows: 8
Columns: 12
PlateRecord('WellRecord['A01'], WellRecord['A02'], WellRecord['A03'], ...,␣
↪WellRecord['H12']')
```

Note that long well lists are shown truncated.

__hash__ = None

class Bio.phenotype.phen_micro.**WellRecord**(*wellid*, *plate=None*, *signals=None*)

Bases: object

WellRecord stores all time course signals of a phenotype Microarray well.

The single time points and signals can be accessed iterating on the WellRecord or using lists indexes or slices:

```
>>> from Bio import phenotype
>>> plate = phenotype.read("phenotype/Plate.json", "pm-json")
>>> well = plate['A05']
>>> for time, signal in well:
...     print("Time: %f, Signal: %f" % (time, signal))
...
Time: 0.000000, Signal: 14.000000
Time: 0.250000, Signal: 13.000000
Time: 0.500000, Signal: 15.000000
Time: 0.750000, Signal: 15.000000
...
>>> well[1]
16.0
>>> well[1:5]
[16.0, 20.0, 18.0, 15.0]
>>> well[1:5:0.5]
[16.0, 19.0, 20.0, 18.0, 18.0, 18.0, 15.0, 18.0]
```

If a time point was not present in the input file but it's between the minimum and maximum time point, the interpolated signal is returned, otherwise a nan value:

```
>>> well[1.3]
19.0
>>> well[1250]
nan
```

Two WellRecord objects can be compared: if their input time/signal pairs are exactly the same, the two records are considered equal:

```
>>> well2 = plate['H12']
>>> well == well2
False
```

Two WellRecord objects can be summed up or subtracted from each other: a new WellRecord object is returned, having the left operand id.

```
>>> well1 = plate['A05']
>>> well2 = well + well1
>>> print(well2.id)
A05
```

If SciPy is installed, a sigmoid function can be fitted to the PM curve, in order to extract some parameters; three sigmoid functions are available: * gompertz * logistic * richards The functions are described in Zwietering et al., 1990 (PMID: 16348228)

For example:

```
well.fit()
print(well.slope, well.model)
(61.853516785566917, 'logistic')
```

If not sigmoid function is specified, the first one that is successfully fitted is used. The user can also specify a specific function.

To specify gompertz:

```
well.fit('gompertz')
print(well.slope, well.model)
(127.94630059171354, 'gompertz')
```

If no function can be fitted, the parameters are left as None, except for the max, min, average_height and area.

__init__(*wellid*, *plate=None*, *signals=None*)

Initialize the class.

__setitem__(*time*, *signal*)

Assign a signal at a certain time point.

__getitem__(*time*)

Return a subset of signals or a single signal.

__iter__()

__eq__(*other*)

Return self==value.

__add__(*well*)

Add another WellRecord object.

A new WellRecord object is returned, having the same id as the left operand

__sub__(*well*)

Subtract another WellRecord object.

A new WellRecord object is returned, having the same id as the left operand

__len__()

Return the number of time points sampled.

__repr__()

Return a (truncated) representation of the signals for debugging.

__str__()

Return a human readable summary of the record (string).

The python built-in function str works by calling the object's __str__ method. e.g.

```
>>> from Bio import phenotype
>>> plate = phenotype.read("phenotype/Plate.json", "pm-json")
>>> record = plate['A05']
>>> print(record)
Plate ID: PM01
Well ID: A05
Time points: 384
Minum signal 0.25 at time 13.00
Maximum signal 19.50 at time 23.00
WellRecord('(0.0, 14.0), (0.25, 13.0), (0.5, 15.0), (0.75, 15.0), (1.0, 16.0), .
↪.., (95.75, 16.0)')
```

Note that long time spans are shown truncated.

get_raw()

Get a list of time/signal pairs.

get_times()

Get a list of the recorded time points.

get_signals()

Get a list of the recorded signals (ordered by collection time).

fit(*function=('gompertz', 'logistic', 'richards')*)

Fit a sigmoid function to this well and extract curve parameters.

If function is None or an empty tuple/list, then no fitting is done. Only the object's `.min`, `.max` and `.average_height` are calculated.

By default the following fitting functions will be used in order:

- gompertz
- logistic
- richards

The first function that is successfully fitted to the signals will be used to extract the curve parameters and update `.area` and `.model`. If no function can be fitted an exception is raised.

The function argument should be a tuple or list of any of these three function names as strings.

There is no return value.

__hash__ = None

Bio.phenotype.phen_micro.**JsonIterator**(*handle*)

Iterate over PM json records as PlateRecord objects.

Arguments:

- handle - input file

Bio.phenotype.phen_micro.**CsvIterator**(*handle*)

Iterate over PM csv records as PlateRecord objects.

Arguments:

- handle - input file

class Bio.phenotype.phen_micro.**JsonWriter**(*plates*)

Bases: `object`

Class to write PM Json format files.

__init__(*plates*)

Initialize the class.

write(*handle*)

Write this instance's plates to a file handle.

Bio.phenotype.pm_fitting module

Growth curves fitting and parameters extraction for phenotype data.

This module provides functions to perform sigmoid functions fitting to Phenotype Microarray data. This module depends on scipy curve_fit function. If not available, a warning is raised.

Functions: logistic Logistic growth model. gompertz Gompertz growth model. richards Richards growth model. guess_plateau Guess the plateau point to improve sigmoid fitting. guess_lag Guess the lag point to improve sigmoid fitting. fit Sigmoid functions fit. get_area Calculate the area under the PM curve.

Bio.phenotype.pm_fitting.**logistic**(*x*, *A*, *u*, *d*, *v*, *y0*)

Logistic growth model.

Proposed in Zwietering et al., 1990 (PMID: 16348228)

Bio.phenotype.pm_fitting.**gompertz**(*x*, *A*, *u*, *d*, *v*, *y0*)

Gompertz growth model.

Proposed in Zwietering et al., 1990 (PMID: 16348228)

Bio.phenotype.pm_fitting.**richards**(*x*, *A*, *u*, *d*, *v*, *y0*)

Richards growth model (equivalent to Stannard).

Proposed in Zwietering et al., 1990 (PMID: 16348228)

Bio.phenotype.pm_fitting.**guess_lag**(*x*, *y*)

Given two axes returns a guess of the lag point.

The lag point is defined as the x point where the difference in y with the next point is higher then the mean differences between the points plus one standard deviation. If such point is not found or x and y have different lengths the function returns zero.

Bio.phenotype.pm_fitting.**guess_plateau**(*x*, *y*)

Given two axes returns a guess of the plateau point.

The plateau point is defined as the x point where the y point is near one standard deviation of the differences between the y points to the maximum y value. If such point is not found or x and y have different lengths the function returns zero.

Bio.phenotype.pm_fitting.**fit**(*function*, *x*, *y*)

Fit the provided function to the x and y values.

The function parameters and the parameters covariance.

Bio.phenotype.pm_fitting.**get_area**(*y*, *x*)

Get the area under the curve.

Module contents

phenotype data input/output.

Input

The main function is Bio.phenotype.parse(...) which takes an input file, and format string. This returns an iterator giving PlateRecord objects:

```
>>> from Bio import phenotype
>>> for record in phenotype.parse("phenotype/Plates.csv", "pm-csv"):
...     print("%s %i" % (record.id, len(record)))
...
PM01 96
PM09 96
```

Note that the parse() function will invoke the relevant parser for the format with its default settings. You may want more control, in which case you need to create a format specific sequence iterator directly.

Input - Single Records

If you expect your file to contain one-and-only-one record, then we provide the following 'helper' function which will return a single PlateRecord, or raise an exception if there are no records or more than one record:

```
>>> from Bio import phenotype
>>> record = phenotype.read("phenotype/Plate.json", "pm-json")
>>> print("%s %i" % (record.id, len(record)))
PM01 96
```

This style is useful when you expect a single record only (and would consider multiple records an error). For example, when dealing with PM JSON files saved by the opm library.

However, if you just want the first record from a file containing multiple record, use the next() function on the iterator:

```
>>> from Bio import phenotype
>>> record = next(phenotype.parse("phenotype/Plates.csv", "pm-csv"))
>>> print("%s %i" % (record.id, len(record)))
PM01 96
```

The above code will work as long as the file contains at least one record. Note that if there is more than one record, the remaining records will be silently ignored.

Output

Use the function Bio.phenotype.write(...), which takes a complete set of PlateRecord objects (either as a list, or an iterator), an output file handle (or in recent versions of Biopython an output filename as a string) and of course the file format:

```
from Bio import phenotype
records = ...
phenotype.write(records, "example.json", "pm-json")
```

Or, using a handle:

```
from Bio import phenotype
records = ...
```

(continues on next page)

(continued from previous page)

```
with open("example.json", "w") as handle:
   phenotype.write(records, handle, "pm-json")
```

You are expected to call this function once (with all your records) and if using a handle, make sure you close it to flush the data to the hard disk.

File Formats

When specifying the file format, use lowercase strings.

- pm-json - Phenotype Microarray plates in JSON format.
- **pm-csv - Phenotype Microarray plates in CSV format, which is the** machine vendor format

Note that while Bio.phenotype can read the above file formats, it can only write in JSON format.

`Bio.phenotype.write`(*plates*, *handle*, *format*)

Write complete set of PlateRecords to a file.

- plates - A list (or iterator) of PlateRecord objects.
- **handle - File handle object to write to, or filename as string** (note older versions of Biopython only took a handle).
- format - lower case string describing the file format to write.

You should close the handle after calling this function.

Returns the number of records written (as an integer).

`Bio.phenotype.parse`(*handle*, *format*)

Turn a phenotype file into an iterator returning PlateRecords.

- **handle - handle to the file, or the filename as a string** (note older versions of Biopython only took a handle).
- format - lower case string describing the file format.

Typical usage, opening a file to read in, and looping over the record(s):

```
>>> from Bio import phenotype
>>> filename = "phenotype/Plates.csv"
>>> for record in phenotype.parse(filename, "pm-csv"):
...     print("ID %s" % record.id)
...     print("Number of wells %i" % len(record))
...
ID PM01
Number of wells 96
ID PM09
Number of wells 96
```

Use the Bio.phenotype.read(...) function when you expect a single record only.

`Bio.phenotype.read`(*handle*, *format*)

Turn a phenotype file into a single PlateRecord.

- **handle - handle to the file, or the filename as a string** (note older versions of Biopython only took a handle).

- format - string describing the file format.

This function is for use parsing phenotype files containing exactly one record. For example, reading a PM JSON file:

```
>>> from Bio import phenotype
>>> record = phenotype.read("phenotype/Plate.json", "pm-json")
>>> print("ID %s" % record.id)
ID PM01
>>> print("Number of wells %i" % len(record))
Number of wells 96
```

If the handle contains no records, or more than one record, an exception is raised. For example:

```
from Bio import phenotype
record = phenotype.read("plates.csv", "pm-csv")
Traceback (most recent call last):
...
ValueError: More than one record found in handle
```

If however you want the first record from a file containing multiple records this function would raise an exception (as shown in the example above). Instead use:

```
>>> from Bio import phenotype
>>> record = next(phenotype.parse("phenotype/Plates.csv", "pm-csv"))
>>> print("First record's ID %s" % record.id)
First record's ID PM01
```

Use the Bio.phenotype.parse(handle, format) function if you want to read multiple records from the handle.

28.2 Submodules

28.2.1 Bio.File module

Code for more fancy file handles.

Bio.File defines private classes used in Bio.SeqIO and Bio.SearchIO for indexing files. These are not intended for direct use.

`Bio.File.as_handle`(*handleish*, *mode='r'*, ***kwargs*)

Context manager to ensure we are using a handle.

Context manager for arguments that can be passed to SeqIO and AlignIO read, write, and parse methods: either file objects or path-like objects (strings, pathlib.Path instances, or more generally, anything that can be handled by the builtin 'open' function).

When given a path-like object, returns an open file handle to that path, with provided mode, which will be closed when the manager exits.

All other inputs are returned, and are *not* closed.

Arguments:

- **handleish - Either a file handle or path-like object (anything which can be**
  passed to the builtin 'open' function, such as str, bytes, pathlib.Path, and os.DirEntry objects)
- mode - Mode to open handleish (used only if handleish is a string)

- kwargs - Further arguments to pass to open(...)

Examples

```
>>> from Bio import File
>>> import os
>>> with File.as_handle('seqs.fasta', 'w') as fp:
...     fp.write('>test\nACGT')
...
10
>>> fp.closed
True
```

```
>>> handle = open('seqs.fasta', 'w')
>>> with File.as_handle(handle) as fp:
...     fp.write('>test\nACGT')
...
10
>>> fp.closed
False
>>> fp.close()
>>> os.remove("seqs.fasta")  # tidy up
```

28.2.2 Bio.LogisticRegression module

Code for doing logistic regressions (DEPRECATED).

Classes:

- LogisticRegression Holds information for a LogisticRegression classifier.

Functions:

- train Train a new classifier.
- calculate Calculate the probabilities of each class, given an observation.
- classify Classify an observation into a class.

This module has been deprecated, please consider an alternative like scikit-learn instead.

class Bio.LogisticRegression.**LogisticRegression**

Bases: object

Holds information necessary to do logistic regression classification.

Attributes:

- beta - List of the weights for each dimension.

__init__()

Initialize the class.

Bio.LogisticRegression.**train**(*xs*, *ys*, *update_fn=None*, *typecode=None*)

Train a logistic regression classifier on a training set.

Argument xs is a list of observations and ys is a list of the class assignments, which should be 0 or 1. xs and ys should contain the same number of elements. update_fn is an optional callback function that takes as parameters that iteration number and log likelihood.

Bio.LogisticRegression.**calculate**(*lr*, *x*)

Calculate the probability for each class.

Arguments:

- lr is a LogisticRegression object.
- x is the observed data.

Returns a list of the probability that it fits each class.

Bio.LogisticRegression.**classify**(*lr*, *x*)

Classify an observation into a class.

28.2.3 Bio.MarkovModel module

A state-emitting MarkovModel.

Note terminology similar to Manning and Schutze is used.

Functions: train_bw Train a markov model using the Baum-Welch algorithm. train_visible Train a visible markov model using MLE. find_states Find the a state sequence that explains some observations.

load Load a MarkovModel. save Save a MarkovModel.

Classes: MarkovModel Holds the description of a markov model

Bio.MarkovModel.**itemindex**(*values*)

Return a dictionary of values with their sequence offset as keys.

class Bio.MarkovModel.**MarkovModel**(*states*, *alphabet*, *p_initial=None*, *p_transition=None*, *p_emission=None*)

Bases: object

Create a state-emitting MarkovModel object.

__init__(*states*, *alphabet*, *p_initial=None*, *p_transition=None*, *p_emission=None*)

Initialize the class.

__str__()

Create a string representation of the MarkovModel object.

Bio.MarkovModel.**load**(*handle*)

Parse a file handle into a MarkovModel object.

Bio.MarkovModel.**save**(*mm*, *handle*)

Save MarkovModel object into handle.

Bio.MarkovModel.**train_bw**(*states*, *alphabet*, *training_data*, *pseudo_initial=None*, *pseudo_transition=None*, *pseudo_emission=None*, *update_fn=None*)

Train a MarkovModel using the Baum-Welch algorithm.

Train a MarkovModel using the Baum-Welch algorithm. states is a list of strings that describe the names of each state. alphabet is a list of objects that indicate the allowed outputs. training_data is a list of observations. Each observation is a list of objects from the alphabet.

pseudo_initial, pseudo_transition, and pseudo_emission are optional parameters that you can use to assign pseudo-counts to different matrices. They should be matrices of the appropriate size that contain numbers to add to each parameter matrix, before normalization.

update_fn is an optional callback that takes parameters (iteration, log_likelihood). It is called once per iteration.

Bio.MarkovModel.**train_visible**(*states*, *alphabet*, *training_data*, *pseudo_initial=None*, *pseudo_transition=None*, *pseudo_emission=None*)

Train a visible MarkovModel using maximum likelihoood estimates for each of the parameters.

Train a visible MarkovModel using maximum likelihoood estimates for each of the parameters. states is a list of strings that describe the names of each state. alphabet is a list of objects that indicate the allowed outputs. training_data is a list of (outputs, observed states) where outputs is a list of the emission from the alphabet, and observed states is a list of states from states.

pseudo_initial, pseudo_transition, and pseudo_emission are optional parameters that you can use to assign pseudo-counts to different matrices. They should be matrices of the appropriate size that contain numbers to add to each parameter matrix.

Bio.MarkovModel.**find_states**(*markov_model*, *output*)

Find states in the given Markov model output.

Returns a list of (states, score) tuples.

28.2.4 Bio.MaxEntropy module

Maximum Entropy code.

Uses Improved Iterative Scaling.

class Bio.MaxEntropy.**MaxEntropy**

Bases: `object`

Hold information for a Maximum Entropy classifier.

Members: classes List of the possible classes of data. alphas List of the weights for each feature. feature_fns List of the feature functions.

Car data from example Naive Bayes Classifier example by Eric Meisner November 22, 2003 http://www.inf.u-szeged.hu/~ormandi/teaching

```
>>> from Bio.MaxEntropy import train, classify
>>> xcar = [
...     ['Red', 'Sports', 'Domestic'],
...     ['Red', 'Sports', 'Domestic'],
...     ['Red', 'Sports', 'Domestic'],
...     ['Yellow', 'Sports', 'Domestic'],
...     ['Yellow', 'Sports', 'Imported'],
...     ['Yellow', 'SUV', 'Imported'],
...     ['Yellow', 'SUV', 'Imported'],
...     ['Yellow', 'SUV', 'Domestic'],
...     ['Red', 'SUV', 'Imported'],
...     ['Red', 'Sports', 'Imported']]
>>> ycar = ['Yes','No','Yes','No','Yes','No','Yes','No','No','Yes']
```

Requires some rules or features

```
>>> def udf1(ts, cl):
...     return ts[0] != 'Red'
...
>>> def udf2(ts, cl):
...     return ts[1] != 'Sports'
...
>>> def udf3(ts, cl):
...     return ts[2] != 'Domestic'
...
>>> user_functions = [udf1, udf2, udf3]  # must be an iterable type
>>> xe = train(xcar, ycar, user_functions)
>>> for xv, yv in zip(xcar, ycar):
...     xc = classify(xe, xv)
...     print('Pred: %s gives %s y is %s' % (xv, xc, yv))
...
Pred: ['Red', 'Sports', 'Domestic'] gives No y is Yes
Pred: ['Red', 'Sports', 'Domestic'] gives No y is No
Pred: ['Red', 'Sports', 'Domestic'] gives No y is Yes
Pred: ['Yellow', 'Sports', 'Domestic'] gives No y is No
Pred: ['Yellow', 'Sports', 'Imported'] gives No y is Yes
Pred: ['Yellow', 'SUV', 'Imported'] gives No y is No
Pred: ['Yellow', 'SUV', 'Imported'] gives No y is Yes
Pred: ['Yellow', 'SUV', 'Domestic'] gives No y is No
Pred: ['Red', 'SUV', 'Imported'] gives No y is No
Pred: ['Red', 'Sports', 'Imported'] gives No y is Yes
```

__init__()

Initialize the class.

Bio.MaxEntropy.**calculate**(*me*, *observation*)

Calculate the log of the probability for each class.

me is a MaxEntropy object that has been trained. observation is a vector representing the observed data. The return value is a list of unnormalized log probabilities for each class.

Bio.MaxEntropy.**classify**(*me*, *observation*)

Classify an observation into a class.

Bio.MaxEntropy.**train**(*training_set*, *results*, *feature_fns*, *update_fn=None*, *max_iis_iterations=10000*, *iis_converge=1.0e-5*, *max_newton_iterations=100*, *newton_converge=1.0e-10*)

Train a maximum entropy classifier, returns MaxEntropy object.

Train a maximum entropy classifier on a training set. training_set is a list of observations. results is a list of the class assignments for each observation. feature_fns is a list of the features. These are callback functions that take an observation and class and return a 1 or 0. update_fn is a callback function that is called at each training iteration. It is passed a MaxEntropy object that encapsulates the current state of the training.

The maximum number of iterations and the convergence criterion for IIS are given by max_iis_iterations and iis_converge, respectively, while max_newton_iterations and newton_converge are the maximum number of iterations and the convergence criterion for Newton's method.

28.2.5 Bio.NaiveBayes module

General Naive Bayes learner (DEPRECATED).

Naive Bayes is a supervised classification algorithm that uses Bayes rule to compute the fit between a new observation and some previously observed data. The observations are discrete feature vectors, with the Bayes assumption that the features are independent. Although this is hardly ever true, the classifier works well enough in practice.

Glossary:

- observation - A feature vector of discrete data.
- class - A possible classification for an observation.

Classes:

- NaiveBayes - Holds information for a naive Bayes classifier.

Functions:

- train - Train a new naive Bayes classifier.
- calculate - Calculate the probabilities of each class, given an observation.
- classify - Classify an observation into a class.

`class Bio.NaiveBayes.NaiveBayes`

Bases: `object`

Hold information for a NaiveBayes classifier.

Attributes:

- classes - List of the possible classes of data.
- p_conditional - CLASS x DIM array of dicts of value -> `P(value|class,dim)`
- p_prior - List of the prior probabilities for every class.
- dimensionality - Dimensionality of the data.

`__init__()`

Initialize the class.

`Bio.NaiveBayes.calculate(nb, observation, scale=False)`

Calculate the logarithmic conditional probability for each class.

Arguments:

- nb - A NaiveBayes classifier that has been trained.
- observation - A list representing the observed data.
- scale - Boolean to indicate whether the probability should be scaled by `P(observation)`. By default, no scaling is done.

A dictionary is returned where the key is the class and the value is the log probability of the class.

`Bio.NaiveBayes.classify(nb, observation)`

Classify an observation into a class.

`Bio.NaiveBayes.train(training_set, results, priors=None, typecode=None)`

Train a NaiveBayes classifier on a training set.

Arguments:

- training_set - List of observations.

- results - List of the class assignments for each observation. Thus, training_set and results must be the same length.
- priors - Optional dictionary specifying the prior probabilities for each type of result. If not specified, the priors will be estimated from the training results.

28.2.6 Bio.Seq module

Provide objects to represent biological sequences.

See also the Seq wiki and the chapter in our tutorial:

- HTML Tutorial
- PDF Tutorial

class Bio.Seq.**SequenceDataAbstractBaseClass**

Bases: ABC

Abstract base class for sequence content providers.

Most users will not need to use this class. It is used internally as a base class for sequence content provider classes such as _UndefinedSequenceData defined in this module, and _TwoBitSequenceData in Bio.SeqIO.TwoBitIO. Instances of these classes can be used instead of a `bytes` object as the data argument when creating a Seq object, and provide the sequence content only when requested via `__getitem__`. This allows lazy parsers to load and parse sequence data from a file only for the requested sequence regions, and _UndefinedSequenceData instances to raise an exception when undefined sequence data are requested.

Future implementations of lazy parsers that similarly provide on-demand parsing of sequence data should use a subclass of this abstract class and implement the abstract methods `__len__` and `__getitem__`:

- `__len__` must return the sequence length;
- `__getitem__` must return
  - a `bytes` object for the requested region; or
  - a new instance of the subclass for the requested region; or
  - raise an `UndefinedSequenceError`.

  Calling `__getitem__` for a sequence region of size zero should always return an empty `bytes` object. Calling `__getitem__` for the full sequence (as in data[:]) should either return a `bytes` object with the full sequence, or raise an `UndefinedSequenceError`.

Subclasses of SequenceDataAbstractBaseClass must call `super().__init__()` as part of their `__init__` method.

__slots__ = ()

__init__()

Check if `__getitem__` returns a bytes-like object.

abstract __len__()

abstract __getitem__(*key*)

__bytes__()

__hash__()

Return hash(self).

__eq__(*other*)

Return self==value.

__lt__(*other*)

Return self<value.

__le__(*other*)

Return self<=value.

__gt__(*other*)

Return self>value.

__ge__(*other*)

Return self>=value.

__add__(*other*)

__radd__(*other*)

__mul__(*other*)

__contains__(*item*)

decode(*encoding='utf-8'*)

Decode the data as bytes using the codec registered for encoding.

encoding
The encoding with which to decode the bytes.

count(*sub*, *start=None*, *end=None*)

Return the number of non-overlapping occurrences of sub in data[start:end].

Optional arguments start and end are interpreted as in slice notation. This method behaves as the count method of Python strings.

find(*sub*, *start=None*, *end=None*)

Return the lowest index in data where subsection sub is found.

Return the lowest index in data where subsection sub is found, such that sub is contained within data[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rfind(*sub*, *start=None*, *end=None*)

Return the highest index in data where subsection sub is found.

Return the highest index in data where subsection sub is found, such that sub is contained within data[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

index(*sub*, *start=None*, *end=None*)

Return the lowest index in data where subsection sub is found.

Return the lowest index in data where subsection sub is found, such that sub is contained within data[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the subsection is not found.

rindex(*sub*, *start=None*, *end=None*)

Return the highest index in data where subsection sub is found.

Return the highest index in data where subsection sub is found, such that sub is contained within data[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raise ValueError when the subsection is not found.

startswith(*prefix*, *start=None*, *end=None*)

Return True if data starts with the specified prefix, False otherwise.

With optional start, test data beginning at that position. With optional end, stop comparing data at that position. prefix can also be a tuple of bytes to try.

endswith(*suffix*, *start=None*, *end=None*)

Return True if data ends with the specified suffix, False otherwise.

With optional start, test data beginning at that position. With optional end, stop comparing data at that position. suffix can also be a tuple of bytes to try.

split(*sep=None*, *maxsplit=-1*)

Return a list of the sections in the data, using sep as the delimiter.

sep
The delimiter according which to split the data. None (the default value) means split on ASCII whitespace characters (space, tab, return, newline, formfeed, vertical tab).

maxsplit
Maximum number of splits to do. -1 (the default value) means no limit.

rsplit(*sep=None*, *maxsplit=-1*)

Return a list of the sections in the data, using sep as the delimiter.

sep
The delimiter according which to split the data. None (the default value) means split on ASCII whitespace characters (space, tab, return, newline, formfeed, vertical tab).

maxsplit
Maximum number of splits to do. -1 (the default value) means no limit.

Splitting is done starting at the end of the data and working to the front.

strip(*chars=None*)

Strip leading and trailing characters contained in the argument.

If the argument is omitted or None, strip leading and trailing ASCII whitespace.

lstrip(*chars=None*)

Strip leading characters contained in the argument.

If the argument is omitted or None, strip leading ASCII whitespace.

rstrip(*chars=None*)

Strip trailing characters contained in the argument.

If the argument is omitted or None, strip trailing ASCII whitespace.

removeprefix(*prefix*)

Remove the prefix if present.

removesuffix(*suffix*)

Remove the suffix if present.

upper()

Return a copy of data with all ASCII characters converted to uppercase.

lower()

Return a copy of data with all ASCII characters converted to lowercase.

isupper()

Return True if all ASCII characters in data are uppercase.

If there are no cased characters, the method returns False.

islower()

Return True if all ASCII characters in data are lowercase.

If there are no cased characters, the method returns False.

replace(*old*, *new*)

Return a copy with all occurrences of substring old replaced by new.

translate(*table*, *delete=b''*)

Return a copy with each character mapped by the given translation table.

table
Translation table, which must be a bytes object of length 256.

All characters occurring in the optional argument delete are removed. The remaining characters are mapped through the given translation table.

property defined

Return True if the sequence is defined, False if undefined or partially defined.

Zero-length sequences are always considered to be defined.

property defined_ranges

Return a tuple of the ranges where the sequence contents is defined.

The return value has the format ((start1, end1), (start2, end2), ...).

__abstractmethods__ = frozenset({'__getitem__', '__len__'})

__annotations__ = {}

class Bio.Seq.**Seq**(*data: str | bytes | bytearray | _SeqAbstractBaseClass | SequenceDataAbstractBaseClass | dict | None*, *length: int | None = None*)

Bases: _SeqAbstractBaseClass

Read-only sequence object (essentially a string with biological methods).

Like normal python strings, our basic sequence object is immutable. This prevents you from doing my_seq[5] = "A" for example, but does allow Seq objects to be used as dictionary keys.

The Seq object provides a number of string like methods (such as count, find, split and strip).

The Seq object also provides some biological methods, such as complement, reverse_complement, transcribe, back_transcribe and translate (which are not applicable to protein sequences).

__init__(*data: str | bytes | bytearray | _SeqAbstractBaseClass | SequenceDataAbstractBaseClass | dict | None*, *length: int | None = None*)

Create a Seq object.

Arguments:

- data - Sequence, required (string)
- length - Sequence length, used only if data is None or a dictionary (integer)

You will typically use Bio.SeqIO to read in sequences from files as SeqRecord objects, whose sequence will be exposed as a Seq object via the seq property.

However, you can also create a Seq object directly:

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF")
>>> my_seq
Seq('MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF')
>>> print(my_seq)
MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF
```

To create a Seq object with for a sequence of known length but unknown sequence contents, use None for the data argument and pass the sequence length for the length argument. Trying to access the sequence contents of a Seq object created in this way will raise an UndefinedSequenceError:

```
>>> my_undefined_sequence = Seq(None, 20)
>>> my_undefined_sequence
Seq(None, length=20)
>>> len(my_undefined_sequence)
20
>>> print(my_undefined_sequence)
Traceback (most recent call last):
...
Bio.Seq.UndefinedSequenceError: Sequence content is undefined
```

If the sequence contents is known for parts of the sequence only, use a dictionary for the data argument to pass the known sequence segments:

```
>>> my_partially_defined_sequence = Seq({3: "ACGT"}, 10)
>>> my_partially_defined_sequence
Seq({3: 'ACGT'}, length=10)
>>> len(my_partially_defined_sequence)
10
>>> print(my_partially_defined_sequence)
Traceback (most recent call last):
...
Bio.Seq.UndefinedSequenceError: Sequence content is only partially defined
>>> my_partially_defined_sequence[3:7]
Seq('ACGT')
>>> print(my_partially_defined_sequence[3:7])
ACGT
```

__hash__()

Hash of the sequence as a string for comparison.

See Seq object comparison documentation (method __eq__ in particular) as this has changed in Biopython 1.65. Older versions would hash on object identity.

__abstractmethods__ = frozenset({})

__annotations__ = {'_data': typing.Union[bytes, Bio.Seq.SequenceDataAbstractBaseClass]}

class Bio.Seq.**MutableSeq**(*data*)

Bases: _SeqAbstractBaseClass

An editable sequence object.

Unlike normal python strings and our basic sequence object (the Seq class) which are immutable, the MutableSeq lets you edit the sequence in place. However, this means you cannot use a MutableSeq object as a dictionary key.

```
>>> from Bio.Seq import MutableSeq
>>> my_seq = MutableSeq("ACTCGTCGTCG")
>>> my_seq
MutableSeq('ACTCGTCGTCG')
>>> my_seq[5]
'T'
>>> my_seq[5] = "A"
>>> my_seq
MutableSeq('ACTCGACGTCG')
>>> my_seq[5]
'A'
>>> my_seq[5:8] = "NNN"
>>> my_seq
MutableSeq('ACTCGNNNTCG')
>>> len(my_seq)
11
```

Note that the MutableSeq object does not support as many string-like or biological methods as the Seq object.

__init__(*data*)

Create a MutableSeq object.

__setitem__(*index*, *value*)

Set a subsequence of single letter via value parameter.

```
>>> my_seq = MutableSeq('ACTCGACGTCG')
>>> my_seq[0] = 'T'
>>> my_seq
MutableSeq('TCTCGACGTCG')
```

__delitem__(*index*)

Delete a subsequence of single letter.

```
>>> my_seq = MutableSeq('ACTCGACGTCG')
>>> del my_seq[0]
>>> my_seq
MutableSeq('CTCGACGTCG')
```

append(*c*)

Add a subsequence to the mutable sequence object.

```
>>> my_seq = MutableSeq('ACTCGACGTCG')
>>> my_seq.append('A')
>>> my_seq
MutableSeq('ACTCGACGTCGA')
```

No return value.

insert(*i*, *c*)

Add a subsequence to the mutable sequence object at a given index.

```
>>> my_seq = MutableSeq('ACTCGACGTCG')
>>> my_seq.insert(0,'A')
>>> my_seq
MutableSeq('AACTCGACGTCG')
>>> my_seq.insert(8,'G')
>>> my_seq
MutableSeq('AACTCGACGGTCG')
```

No return value.

pop(*i=-1*)

Remove a subsequence of a single letter at given index.

```
>>> my_seq = MutableSeq('ACTCGACGTCG')
>>> my_seq.pop()
'G'
>>> my_seq
MutableSeq('ACTCGACGTC')
>>> my_seq.pop()
'C'
>>> my_seq
MutableSeq('ACTCGACGT')
```

Returns the last character of the sequence.

remove(*item*)

Remove a subsequence of a single letter from mutable sequence.

```
>>> my_seq = MutableSeq('ACTCGACGTCG')
>>> my_seq.remove('C')
>>> my_seq
MutableSeq('ATCGACGTCG')
>>> my_seq.remove('A')
>>> my_seq
MutableSeq('TCGACGTCG')
```

No return value.

reverse()

Modify the mutable sequence to reverse itself.

No return value.

extend(*other*)

Add a sequence to the original mutable sequence object.

```
>>> my_seq = MutableSeq('ACTCGACGTCG')
>>> my_seq.extend('A')
>>> my_seq
MutableSeq('ACTCGACGTCGA')
>>> my_seq.extend('TTT')
>>> my_seq
MutableSeq('ACTCGACGTCGATTT')
```

No return value.

__abstractmethods__ = frozenset({})

__annotations__ = {}

exception Bio.Seq.**UndefinedSequenceError**

Bases: ValueError

Sequence contents is undefined.

Bio.Seq.**transcribe**(*dna*)

Transcribe a DNA sequence into RNA.

Following the usual convention, the sequence is interpreted as the coding strand of the DNA double helix, not the template strand. This means we can get the RNA sequence just by switching T to U.

If given a string, returns a new string object.

Given a Seq or MutableSeq, returns a new Seq object.

e.g.

```
>>> transcribe("ACTGN")
'ACUGN'
```

Bio.Seq.**back_transcribe**(*rna*)

Return the RNA sequence back-transcribed into DNA.

If given a string, returns a new string object.

Given a Seq or MutableSeq, returns a new Seq object.

e.g.

```
>>> back_transcribe("ACUGN")
'ACTGN'
```

Bio.Seq.**translate**(*sequence*, *table='Standard'*, *stop_symbol='*'*, *to_stop=False*, *cds=False*, *gap=None*)

Translate a nucleotide sequence into amino acids.

If given a string, returns a new string object. Given a Seq or MutableSeq, returns a Seq object.

Arguments:

- table - Which codon table to use? This can be either a name (string), an NCBI identifier (integer), or a CodonTable object (useful for non-standard genetic codes). Defaults to the "Standard" table.
- stop_symbol - Single character string, what to use for any terminators, defaults to the asterisk, "*".
- to_stop - Boolean, defaults to False meaning do a full translation continuing on past any stop codons (translated as the specified stop_symbol). If True, translation is terminated at the first in frame stop codon (and the stop_symbol is not appended to the returned protein sequence).
- cds - Boolean, indicates this is a complete CDS. If True, this checks the sequence starts with a valid alternative start codon (which will be translated as methionine, M), that the sequence length is a multiple of three, and that there is a single in frame stop codon at the end (this will be excluded from the protein sequence, regardless of the to_stop option). If these tests fail, an exception is raised.
- gap - Single character string to denote symbol used for gaps. Defaults to None.

A simple string example using the default (standard) genetic code:

```
>>> coding_dna = "GTGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG"
>>> translate(coding_dna)
'VAIVMGR*KGAR*'
>>> translate(coding_dna, stop_symbol="@")
'VAIVMGR@KGAR@'
>>> translate(coding_dna, to_stop=True)
'VAIVMGR'
```

Now using NCBI table 2, where TGA is not a stop codon:

```
>>> translate(coding_dna, table=2)
'VAIVMGRWKGAR*'
>>> translate(coding_dna, table=2, to_stop=True)
'VAIVMGRWKGAR'
```

In fact this example uses an alternative start codon valid under NCBI table 2, GTG, which means this example is a complete valid CDS which when translated should really start with methionine (not valine):

```
>>> translate(coding_dna, table=2, cds=True)
'MAIVMGRWKGAR'
```

Note that if the sequence has no in-frame stop codon, then the to_stop argument has no effect:

```
>>> coding_dna2 = "GTGGCCATTGTAATGGGCCGC"
>>> translate(coding_dna2)
'VAIVMGR'
>>> translate(coding_dna2, to_stop=True)
'VAIVMGR'
```

NOTE - Ambiguous codons like "TAN" or "NNN" could be an amino acid or a stop codon. These are translated as "X". Any invalid codon (e.g. "TA?" or "T-A") will throw a TranslationError.

It will however translate either DNA or RNA.

NOTE - Since version 1.71 Biopython contains codon tables with 'ambiguous stop codons'. These are stop codons with unambiguous sequence but which have a context dependent coding as STOP or as amino acid. With these tables 'to_stop' must be False (otherwise a ValueError is raised). The dual coding codons will always be translated as amino acid, except for 'cds=True', where the last codon will be translated as STOP.

```
>>> coding_dna3 = "ATGGCACGGAAGTGA"
>>> translate(coding_dna3)
'MARK*'
```

```
>>> translate(coding_dna3, table=27)  # Table 27: TGA -> STOP or W
'MARKW'
```

It will however raise a BiopythonWarning (not shown).

```
>>> translate(coding_dna3, table=27, cds=True)
'MARK'
```

```
>>> translate(coding_dna3, table=27, to_stop=True)
Traceback (most recent call last):
  ...
ValueError: You cannot use 'to_stop=True' with this table ...
```

Bio.Seq.**reverse_complement**(*sequence*, *inplace=False*)

Return the reverse complement as a DNA sequence.

If given a string, returns a new string object. Given a Seq object, returns a new Seq object. Given a MutableSeq, returns a new MutableSeq object. Given a SeqRecord object, returns a new SeqRecord object.

```
>>> my_seq = "CGA"
>>> reverse_complement(my_seq)
'TCG'
>>> my_seq = Seq("CGA")
>>> reverse_complement(my_seq)
Seq('TCG')
>>> my_seq = MutableSeq("CGA")
>>> reverse_complement(my_seq)
MutableSeq('TCG')
>>> my_seq
MutableSeq('CGA')
```

Any U in the sequence is treated as a T:

```
>>> reverse_complement(Seq("CGAUT"))
Seq('AATCG')
```

In contrast, `reverse_complement_rna` returns an RNA sequence:

```
>>> reverse_complement_rna(Seq("CGAUT"))
Seq('AAUCG')
```

Supports and lower- and upper-case characters, and unambiguous and ambiguous nucleotides. All other characters are not converted:

```
>>> reverse_complement("ACGTUacgtuXYZxyz")
'zrxZRXaacgtAACGT'
```

The sequence is modified in-place and returned if inplace is True:

```
>>> my_seq = MutableSeq("CGA")
>>> reverse_complement(my_seq, inplace=True)
MutableSeq('TCG')
>>> my_seq
MutableSeq('TCG')
```

As strings and `Seq` objects are immutable, a `TypeError` is raised if `reverse_complement` is called on a `Seq` object with `inplace=True`.

Bio.Seq.**reverse_complement_rna**(*sequence*, *inplace=False*)

Return the reverse complement as an RNA sequence.

If given a string, returns a new string object. Given a Seq object, returns a new Seq object. Given a MutableSeq, returns a new MutableSeq object. Given a SeqRecord object, returns a new SeqRecord object.

```
>>> my_seq = "CGA"
>>> reverse_complement_rna(my_seq)
'UCG'
>>> my_seq = Seq("CGA")
```

(continues on next page)

(continued from previous page)

```
>>> reverse_complement_rna(my_seq)
Seq('UCG')
>>> my_seq = MutableSeq("CGA")
>>> reverse_complement_rna(my_seq)
MutableSeq('UCG')
>>> my_seq
MutableSeq('CGA')
```

Any T in the sequence is treated as a U:

```
>>> reverse_complement_rna(Seq("CGAUT"))
Seq('AAUCG')
```

In contrast, `reverse_complement` returns a DNA sequence:

```
>>> reverse_complement(Seq("CGAUT"), inplace=False)
Seq('AATCG')
```

Supports and lower- and upper-case characters, and unambiguous and ambiguous nucleotides. All other characters are not converted:

```
>>> reverse_complement_rna("ACGTUacgtuXYZxyz")
'zrxZRXaacguAACGU'
```

The sequence is modified in-place and returned if inplace is True:

```
>>> my_seq = MutableSeq("CGA")
>>> reverse_complement_rna(my_seq, inplace=True)
MutableSeq('UCG')
>>> my_seq
MutableSeq('UCG')
```

As strings and `Seq` objects are immutable, a `TypeError` is raised if `reverse_complement` is called on a `Seq` object with `inplace=True`.

`Bio.Seq.`**`complement`**(*sequence*, *inplace=False*)

Return the complement as a DNA sequence.

If given a string, returns a new string object. Given a Seq object, returns a new Seq object. Given a MutableSeq, returns a new MutableSeq object. Given a SeqRecord object, returns a new SeqRecord object.

```
>>> my_seq = "CGA"
>>> complement(my_seq)
'GCT'
>>> my_seq = Seq("CGA")
>>> complement(my_seq)
Seq('GCT')
>>> my_seq = MutableSeq("CGA")
>>> complement(my_seq)
MutableSeq('GCT')
>>> my_seq
MutableSeq('CGA')
```

Any U in the sequence is treated as a T:

```
>>> complement(Seq("CGAUT"))
Seq('GCTAA')
```

In contrast, `complement_rna` returns an RNA sequence:

```
>>> complement_rna(Seq("CGAUT"))
Seq('GCUAA')
```

Supports and lower- and upper-case characters, and unambiguous and ambiguous nucleotides. All other characters are not converted:

```
>>> complement("ACGTUacgtuXYZxyz")
'TGCAAtgcaaXRZxrz'
```

The sequence is modified in-place and returned if inplace is True:

```
>>> my_seq = MutableSeq("CGA")
>>> complement(my_seq, inplace=True)
MutableSeq('GCT')
>>> my_seq
MutableSeq('GCT')
```

As strings and `Seq` objects are immutable, a `TypeError` is raised if `reverse_complement` is called on a `Seq` object with `inplace=True`.

`Bio.Seq.`**`complement_rna`**(*sequence*, *inplace=False*)

Return the complement as an RNA sequence.

If given a string, returns a new string object. Given a Seq object, returns a new Seq object. Given a MutableSeq, returns a new MutableSeq object. Given a SeqRecord object, returns a new SeqRecord object.

```
>>> my_seq = "CGA"
>>> complement_rna(my_seq)
'GCU'
>>> my_seq = Seq("CGA")
>>> complement_rna(my_seq)
Seq('GCU')
>>> my_seq = MutableSeq("CGA")
>>> complement_rna(my_seq)
MutableSeq('GCU')
>>> my_seq
MutableSeq('CGA')
```

Any T in the sequence is treated as a U:

```
>>> complement_rna(Seq("CGAUT"))
Seq('GCUAA')
```

In contrast, `complement` returns a DNA sequence:

```
>>> complement(Seq("CGAUT"))
Seq('GCTAA')
```

Supports and lower- and upper-case characters, and unambiguous and ambiguous nucleotides. All other characters are not converted:

```
>>> complement_rna("ACGTUacgtuXYZxyz")
'UGCAAugcaaXRZxrz'
```

The sequence is modified in-place and returned if inplace is True:

```
>>> my_seq = MutableSeq("CGA")
>>> complement(my_seq, inplace=True)
MutableSeq('GCT')
>>> my_seq
MutableSeq('GCT')
```

As strings and `Seq` objects are immutable, a `TypeError` is raised if `reverse_complement` is called on a `Seq` object with `inplace=True`.

28.2.7 Bio.SeqFeature module

Represent a Sequence Feature holding info about a part of a sequence.

This is heavily modeled after the Biocorba SeqFeature objects, and may be pretty biased towards GenBank stuff since I'm writing it for the GenBank parser output...

What's here:

Base class to hold a Feature

Classes:

- SeqFeature

Hold information about a Reference

This is an attempt to create a General class to hold Reference type information.

Classes:

- Reference

Specify locations of a feature on a Sequence

This aims to handle, in Ewan Birney's words, 'the dreaded fuzziness issue'. This has the advantages of allowing us to handle fuzzy stuff in case anyone needs it, and also be compatible with BioPerl etc and BioSQL.

Classes:

- Location - abstract base class of SimpleLocation and CompoundLocation.
- SimpleLocation - Specify the start and end location of a feature.
- CompoundLocation - Collection of SimpleLocation objects (for joins etc).
- Position - abstract base class of ExactPosition, WithinPosition, BetweenPosition, AfterPosition, OneOfPosition, UncertainPosition, and UnknownPosition.
- ExactPosition - Specify the position as being exact.
- WithinPosition - Specify a position occurring within some range.

- BetweenPosition - Specify a position occurring between a range (OBSOLETE?).
- BeforePosition - Specify the position as being found before some base.
- AfterPosition - Specify the position as being found after some base.
- OneOfPosition - Specify a position consisting of multiple alternative positions.
- UncertainPosition - Specify a specific position which is uncertain.
- UnknownPosition - Represents missing information like '?' in UniProt.

Exceptions:

- LocationParserError - Exception indicating a failure to parse a location string.

exception Bio.SeqFeature.**LocationParserError**

Bases: ValueError

Could not parse a feature location string.

class Bio.SeqFeature.**SeqFeature**(*location=None*, *type=''*, *id='<unknown id>'*, *qualifiers=None*, *sub_features=None*)

Bases: object

Represent a Sequence Feature on an object.

Attributes:

- location - the location of the feature on the sequence (SimpleLocation)
- type - the specified type of the feature (ie. CDS, exon, repeat...)
- id - A string identifier for the feature.
- qualifiers - A dictionary of qualifiers on the feature. These are analogous to the qualifiers from a GenBank feature table. The keys of the dictionary are qualifier names, the values are the qualifier values.

__init__(*location=None*, *type=''*, *id='<unknown id>'*, *qualifiers=None*, *sub_features=None*)

Initialize a SeqFeature on a sequence.

location can either be a SimpleLocation (with strand argument also given if required), or None.

e.g. With no strand, on the forward strand, and on the reverse strand:

```
>>> from Bio.SeqFeature import SeqFeature, SimpleLocation
>>> f1 = SeqFeature(SimpleLocation(5, 10), type="domain")
>>> f1.location.strand == None
True
>>> f2 = SeqFeature(SimpleLocation(7, 110, strand=1), type="CDS")
>>> f2.location.strand == +1
True
>>> f3 = SeqFeature(SimpleLocation(9, 108, strand=-1), type="CDS")
>>> f3.location.strand == -1
True
```

For exact start/end positions, an integer can be used (as shown above) as shorthand for the ExactPosition object. For non-exact locations, the SimpleLocation must be specified via the appropriate position objects.

property strand

Alias for the location's strand (DEPRECATED).

property ref

Alias for the location's ref (DEPRECATED).

property ref_db

Alias for the location's ref_db (DEPRECATED).

__eq__(*other*)

Check if two SeqFeature objects should be considered equal.

__repr__()

Represent the feature as a string for debugging.

__str__()

Return the full feature as a python string.

extract(*parent_sequence*, *references=None*)

Extract the feature's sequence from supplied parent sequence.

The parent_sequence can be a Seq like object or a string, and will generally return an object of the same type. The exception to this is a MutableSeq as the parent sequence will return a Seq object.

This should cope with complex locations including complements, joins and fuzzy positions. Even mixed strand features should work! This also covers features on protein sequences (e.g. domains), although here reverse strand features are not permitted. If the location refers to other records, they must be supplied in the optional dictionary references.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqFeature import SeqFeature, SimpleLocation
>>> seq = Seq("MKQHKAMIVALIVICITAVVAAL")
>>> f = SeqFeature(SimpleLocation(8, 15), type="domain")
>>> f.extract(seq)
Seq('VALIVIC')
```

If the SimpleLocation is None, e.g. when parsing invalid locus locations in the GenBank parser, extract() will raise a ValueError.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqFeature import SeqFeature
>>> seq = Seq("MKQHKAMIVALIVICITAVVAAL")
>>> f = SeqFeature(None, type="domain")
>>> f.extract(seq)
Traceback (most recent call last):
  ...
ValueError: The feature's .location is None. Check the sequence file for a␣
↪valid location.
```

Note - currently only compound features of type "join" are supported.

translate(*parent_sequence*, *table='Standard'*, *start_offset=None*, *stop_symbol='*'*, *to_stop=False*, *cds=None*, *gap=None*)

Get a translation of the feature's sequence.

This method is intended for CDS or other features that code proteins and is a shortcut that will both extract the feature and translate it, taking into account the codon_start and transl_table qualifiers, if they are present. If they are not present the value of the arguments "table" and "start_offset" are used.

The "cds" parameter is set to "True" if the feature is of type "CDS" but can be overridden by giving an explicit argument.

The arguments stop_symbol, to_stop and gap have the same meaning as Seq.translate, refer to that documentation for further information.

Arguments:

- parent_sequence - A DNA or RNA sequence.
- table - Which codon table to use if there is no transl_table qualifier for this feature. This can be either a name (string), an NCBI identifier (integer), or a CodonTable object (useful for non-standard genetic codes). This defaults to the "Standard" table.
- start_offset - offset at which the first complete codon of a coding feature can be found, relative to the first base of that feature. Has a valid value of 0, 1 or 2. NOTE: this uses python's 0-based numbering whereas the codon_start qualifier in files from NCBI use 1-based numbering. Will override a codon_start qualifier

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqFeature import SeqFeature, SimpleLocation
>>> seq = Seq("GGTTACACTTACCGATAATGTCTCTGATGA")
>>> f = SeqFeature(SimpleLocation(0, 30), type="CDS")
>>> f.qualifiers['transl_table'] = [11]
```

Note that features of type CDS are subject to the usual checks at translation. But you can override this behavior by giving explicit arguments:

```
>>> f.translate(seq, cds=False)
Seq('GYTYR*CL**')
```

Now use the start_offset argument to change the frame. Note this uses python 0-based numbering.

```
>>> f.translate(seq, start_offset=1, cds=False)
Seq('VTLTDNVSD')
```

Alternatively use the codon_start qualifier to do the same thing. Note: this uses 1-based numbering, which is found in files from NCBI.

```
>>> f.qualifiers['codon_start'] = [2]
>>> f.translate(seq, cds=False)
Seq('VTLTDNVSD')
```

__bool__()

Boolean value of an instance of this class (True).

This behavior is for backwards compatibility, since until the __len__ method was added, a SeqFeature always evaluated as True.

Note that in comparison, Seq objects, strings, lists, etc, will all evaluate to False if they have length zero.

WARNING: The SeqFeature may in future evaluate to False when its length is zero (in order to better match normal python behavior)!

__len__()

Return the length of the region where the feature is located.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqFeature import SeqFeature, SimpleLocation
>>> seq = Seq("MKQHKAMIVALIVICITAVVAAL")
>>> f = SeqFeature(SimpleLocation(8, 15), type="domain")
```

(continues on next page)

(continued from previous page)

```
>>> len(f)
7
>>> f.extract(seq)
Seq('VALIVIC')
>>> len(f.extract(seq))
7
```

This is a proxy for taking the length of the feature's location:

```
>>> len(f.location)
7
```

For simple features this is the same as the region spanned (end position minus start position using Pythonic counting). However, for a compound location (e.g. a CDS as the join of several exons) the gaps are not counted (e.g. introns). This ensures that len(f) matches len(f.extract(parent_seq)), and also makes sure things work properly with features wrapping the origin etc.

__iter__()

Iterate over the parent positions within the feature.

The iteration order is strand aware, and can be thought of as moving along the feature using the parent sequence coordinates:

```
>>> from Bio.SeqFeature import SeqFeature, SimpleLocation
>>> f = SeqFeature(SimpleLocation(5, 10, strand=-1), type="domain")
>>> len(f)
5
>>> for i in f: print(i)
9
8
7
6
5
>>> list(f)
[9, 8, 7, 6, 5]
```

This is a proxy for iterating over the location,

```
>>> list(f.location)
[9, 8, 7, 6, 5]
```

__contains__(*value*)

Check if an integer position is within the feature.

```
>>> from Bio.SeqFeature import SeqFeature, SimpleLocation
>>> f = SeqFeature(SimpleLocation(5, 10, strand=-1), type="domain")
>>> len(f)
5
>>> [i for i in range(15) if i in f]
[5, 6, 7, 8, 9]
```

For example, to see which features include a SNP position, you could use this:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("GenBank/NC_000932.gb", "gb")
>>> for f in record.features:
...     if 1750 in f:
...         print("%s %s" % (f.type, f.location))
source [0:154478](+)
gene [1716:4347](-)
tRNA join{[4310:4347](-), [1716:1751](-)}
```

Note that for a feature defined as a join of several subfeatures (e.g. the union of several exons) the gaps are not checked (e.g. introns). In this example, the tRNA location is defined in the GenBank file as complement(join(1717..1751,4311..4347)), so that position 1760 falls in the gap:

```
>>> for f in record.features:
...     if 1760 in f:
...         print("%s %s" % (f.type, f.location))
source [0:154478](+)
gene [1716:4347](-)
```

Note that additional care may be required with fuzzy locations, for example just before a BeforePosition:

```
>>> from Bio.SeqFeature import SeqFeature, SimpleLocation
>>> from Bio.SeqFeature import BeforePosition
>>> f = SeqFeature(SimpleLocation(BeforePosition(3), 8), type="domain")
>>> len(f)
5
>>> [i for i in range(10) if i in f]
[3, 4, 5, 6, 7]
```

Note that is is a proxy for testing membership on the location.

```
>>> [i for i in range(10) if i in f.location]
[3, 4, 5, 6, 7]
```

__hash__ = None

class Bio.SeqFeature.**Reference**

Bases: object

Represent a Generic Reference object.

Attributes:

- location - A list of Location objects specifying regions of the sequence that the references correspond to. If no locations are specified, the entire sequence is assumed.
- authors - A big old string, or a list split by author, of authors for the reference.
- title - The title of the reference.
- journal - Journal the reference was published in.
- medline_id - A medline reference for the article.
- pubmed_id - A pubmed reference for the article.
- comment - A place to stick any comments about the reference.

__init__()

Initialize the class.

__str__()

Return the full Reference object as a python string.

__repr__()

Represent the Reference object as a string for debugging.

__eq__(*other*)

Check if two Reference objects should be considered equal.

Note prior to Biopython 1.70 the location was not compared, as until then __eq__ for the SimpleLocation class was not defined.

__hash__ = None

class Bio.SeqFeature.**Location**

Bases: ABC

Abstract base class representing a location.

abstract __repr__()

Represent the Location object as a string for debugging.

fromstring(*length=None*, *circular=False*, *stranded=True*)

Create a Location object from a string.

This should accept any valid location string in the INSDC Feature Table format (https://www.insdc.org/submitting-standards/feature-table/) as used in GenBank, DDBJ and EMBL files.

Simple examples:

```
>>> Location.fromstring("123..456", 1000)
SimpleLocation(ExactPosition(122), ExactPosition(456), strand=1)
>>> Location.fromstring("complement(<123..>456)", 1000)
SimpleLocation(BeforePosition(122), AfterPosition(456), strand=-1)
```

A more complex location using within positions,

```
>>> Location.fromstring("(9.10)..(20.25)", 1000)
SimpleLocation(WithinPosition(8, left=8, right=9), WithinPosition(25, left=20,␣
↪right=25), strand=1)
```

Notice how that will act as though it has overall start 8 and end 25.

Zero length between feature,

```
>>> Location.fromstring("123^124", 1000)
SimpleLocation(ExactPosition(123), ExactPosition(123), strand=1)
```

The expected sequence length is needed for a special case, a between position at the start/end of a circular genome:

```
>>> Location.fromstring("1000^1", 1000)
SimpleLocation(ExactPosition(1000), ExactPosition(1000), strand=1)
```

Apart from this special case, between positions P^Q must have P+1==Q,

```
>>> Location.fromstring("123^456", 1000)
Traceback (most recent call last):
   ...
Bio.SeqFeature.LocationParserError: invalid feature location '123^456'
```

You can optionally provide a reference name:

```
>>> Location.fromstring("AL391218.9:105173..108462", 2000000)
SimpleLocation(ExactPosition(105172), ExactPosition(108462), strand=1, ref=
↪'AL391218.9')
```

```
>>> Location.fromstring("<2644..159", 2868, "circular")
CompoundLocation([SimpleLocation(BeforePosition(2643), ExactPosition(2868),␣
↪strand=1), SimpleLocation(ExactPosition(0), ExactPosition(159), strand=1)],
↪'join')
```

__abstractmethods__ = frozenset({'__repr__'})

__annotations__ = {}

class Bio.SeqFeature.**SimpleLocation**(*start*, *end*, *strand=None*, *ref=None*, *ref_db=None*)

Bases: *Location*

Specify the location of a feature along a sequence.

The SimpleLocation is used for simple continuous features, which can be described as running from a start position to and end position (optionally with a strand and reference information). More complex locations made up from several non-continuous parts (e.g. a coding sequence made up of several exons) are described using a SeqFeature with a CompoundLocation.

Note that the start and end location numbering follow Python's scheme, thus a GenBank entry of 123..150 (one based counting) becomes a location of [122:150] (zero based counting).

```
>>> from Bio.SeqFeature import SimpleLocation
>>> f = SimpleLocation(122, 150)
>>> print(f)
[122:150]
>>> print(f.start)
122
>>> print(f.end)
150
>>> print(f.strand)
None
```

Note the strand defaults to None. If you are working with nucleotide sequences you'd want to be explicit if it is the forward strand:

```
>>> from Bio.SeqFeature import SimpleLocation
>>> f = SimpleLocation(122, 150, strand=+1)
>>> print(f)
[122:150](+)
>>> print(f.strand)
1
```

Note that for a parent sequence of length n, the SimpleLocation start and end must satisfy the inequality 0 <= start <= end <= n. This means even for features on the reverse strand of a nucleotide sequence, we expect the 'start' coordinate to be less than the 'end'.

```
>>> from Bio.SeqFeature import SimpleLocation
>>> r = SimpleLocation(122, 150, strand=-1)
>>> print(r)
[122:150](-)
>>> print(r.start)
122
>>> print(r.end)
150
>>> print(r.strand)
-1
```

i.e. Rather than thinking of the 'start' and 'end' biologically in a strand aware manner, think of them as the 'left most' or 'minimum' boundary, and the 'right most' or 'maximum' boundary of the region being described. This is particularly important with compound locations describing non-continuous regions.

In the example above we have used standard exact positions, but there are also specialised position objects used to represent fuzzy positions as well, for example a GenBank location like complement(<123..150) would use a BeforePosition object for the start.

__init__(*start*, *end*, *strand=None*, *ref=None*, *ref_db=None*)

Initialize the class.

start and end arguments specify the values where the feature begins and ends. These can either by any of the *Position objects that inherit from Position, or can just be integers specifying the position. In the case of integers, the values are assumed to be exact and are converted in ExactPosition arguments. This is meant to make it easy to deal with non-fuzzy ends.

i.e. Short form:

```
>>> from Bio.SeqFeature import SimpleLocation
>>> loc = SimpleLocation(5, 10, strand=-1)
>>> print(loc)
[5:10](-)
```

Explicit form:

```
>>> from Bio.SeqFeature import SimpleLocation, ExactPosition
>>> loc = SimpleLocation(ExactPosition(5), ExactPosition(10), strand=-1)
>>> print(loc)
[5:10](-)
```

Other fuzzy positions are used similarly,

```
>>> from Bio.SeqFeature import SimpleLocation
>>> from Bio.SeqFeature import BeforePosition, AfterPosition
>>> loc2 = SimpleLocation(BeforePosition(5), AfterPosition(10), strand=-1)
>>> print(loc2)
[<5:>10](-)
```

For nucleotide features you will also want to specify the strand, use 1 for the forward (plus) strand, -1 for the reverse (negative) strand, 0 for stranded but strand unknown (? in GFF3), or None for when the strand does not apply (dot in GFF3), e.g. features on proteins.

```
>>> loc = SimpleLocation(5, 10, strand=+1)
>>> print(loc)
[5:10](+)
>>> print(loc.strand)
1
```

Normally feature locations are given relative to the parent sequence you are working with, but an explicit accession can be given with the optional ref and db_ref strings:

```
>>> loc = SimpleLocation(105172, 108462, ref="AL391218.9", strand=1)
>>> print(loc)
AL391218.9[105172:108462](+)
>>> print(loc.ref)
AL391218.9
```

static fromstring(*text*, *length=None*, *circular=False*)

Create a SimpleLocation object from a string.

property strand

Strand of the location (+1, -1, 0 or None).

__str__()

Return a representation of the SimpleLocation object (with python counting).

For the simple case this uses the python splicing syntax, [122:150] (zero based counting) which GenBank would call 123..150 (one based counting).

__repr__()

Represent the SimpleLocation object as a string for debugging.

__add__(*other*)

Combine location with another SimpleLocation object, or shift it.

You can add two feature locations to make a join CompoundLocation:

```
>>> from Bio.SeqFeature import SimpleLocation
>>> f1 = SimpleLocation(5, 10)
>>> f2 = SimpleLocation(20, 30)
>>> combined = f1 + f2
>>> print(combined)
join{[5:10], [20:30]}
```

This is thus equivalent to:

```
>>> from Bio.SeqFeature import CompoundLocation
>>> join = CompoundLocation([f1, f2])
>>> print(join)
join{[5:10], [20:30]}
```

You can also use sum(...) in this way:

```
>>> join = sum([f1, f2])
>>> print(join)
join{[5:10], [20:30]}
```

Furthermore, you can combine a SimpleLocation with a CompoundLocation in this way.

Separately, adding an integer will give a new SimpleLocation with its start and end offset by that amount. For example:

```
>>> print(f1)
[5:10]
>>> print(f1 + 100)
[105:110]
>>> print(200 + f1)
[205:210]
```

This can be useful when editing annotation.

__radd__(*other*)

Return a SimpleLocation object by shifting the location by an integer amount.

__sub__(*other*)

Subtracting an integer will shift the start and end by that amount.

```
>>> from Bio.SeqFeature import SimpleLocation
>>> f1 = SimpleLocation(105, 150)
>>> print(f1)
[105:150]
>>> print(f1 - 100)
[5:50]
```

This can be useful when editing annotation. You can also add an integer to a feature location (which shifts in the opposite direction).

__nonzero__()

Return True regardless of the length of the feature.

This behavior is for backwards compatibility, since until the __len__ method was added, a SimpleLocation always evaluated as True.

Note that in comparison, Seq objects, strings, lists, etc, will all evaluate to False if they have length zero.

WARNING: The SimpleLocation may in future evaluate to False when its length is zero (in order to better match normal python behavior)!

__len__()

Return the length of the region described by the SimpleLocation object.

Note that extra care may be needed for fuzzy locations, e.g.

```
>>> from Bio.SeqFeature import SimpleLocation
>>> from Bio.SeqFeature import BeforePosition, AfterPosition
>>> loc = SimpleLocation(BeforePosition(5), AfterPosition(10))
>>> len(loc)
5
```

__contains__(*value*)

Check if an integer position is within the SimpleLocation object.

Note that extra care may be needed for fuzzy locations, e.g.

```
>>> from Bio.SeqFeature import SimpleLocation
>>> from Bio.SeqFeature import BeforePosition, AfterPosition
>>> loc = SimpleLocation(BeforePosition(5), AfterPosition(10))
>>> len(loc)
5
>>> [i for i in range(15) if i in loc]
[5, 6, 7, 8, 9]
```

__iter__()

Iterate over the parent positions within the SimpleLocation object.

```
>>> from Bio.SeqFeature import SimpleLocation
>>> from Bio.SeqFeature import BeforePosition, AfterPosition
>>> loc = SimpleLocation(BeforePosition(5), AfterPosition(10))
>>> len(loc)
5
>>> for i in loc: print(i)
5
6
7
8
9
>>> list(loc)
[5, 6, 7, 8, 9]
>>> [i for i in range(15) if i in loc]
[5, 6, 7, 8, 9]
```

Note this is strand aware:

```
>>> loc = SimpleLocation(BeforePosition(5), AfterPosition(10), strand = -1)
>>> list(loc)
[9, 8, 7, 6, 5]
```

__eq__(*other*)

Implement equality by comparing all the location attributes.

property parts

Read only list of sections (always one, the SimpleLocation object).

This is a convenience property allowing you to write code handling both SimpleLocation objects (with one part) and more complex CompoundLocation objects (with multiple parts) interchangeably.

property start

Start location - left most (minimum) value, regardless of strand.

Read only, returns an integer like position object, possibly a fuzzy position.

property end

End location - right most (maximum) value, regardless of strand.

Read only, returns an integer like position object, possibly a fuzzy position.

extract(*parent_sequence*, *references=None*)

Extract the sequence from supplied parent sequence using the SimpleLocation object.

The parent_sequence can be a Seq like object or a string, and will generally return an object of the same type. The exception to this is a MutableSeq as the parent sequence will return a Seq object. If the location refers to other records, they must be supplied in the optional dictionary references.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqFeature import SimpleLocation
>>> seq = Seq("MKQHKAMIVALIVICITAVVAAL")
>>> feature_loc = SimpleLocation(8, 15)
>>> feature_loc.extract(seq)
Seq('VALIVIC')
```

__abstractmethods__ = frozenset({})

__annotations__ = {}

__hash__ = None

Bio.SeqFeature.**FeatureLocation**

alias of *SimpleLocation*

class Bio.SeqFeature.**CompoundLocation**(*parts*, *operator='join'*)

Bases: *Location*

For handling joins etc where a feature location has several parts.

__init__(*parts*, *operator='join'*)

Initialize the class.

```
>>> from Bio.SeqFeature import SimpleLocation, CompoundLocation
>>> f1 = SimpleLocation(10, 40, strand=+1)
>>> f2 = SimpleLocation(50, 59, strand=+1)
>>> f = CompoundLocation([f1, f2])
>>> len(f) == len(f1) + len(f2) == 39 == len(list(f))
True
>>> print(f.operator)
join
>>> 5 in f
False
>>> 15 in f
True
>>> f.strand
1
```

Notice that the strand of the compound location is computed automatically - in the case of mixed strands on the sub-locations the overall strand is set to None.

```
>>> f = CompoundLocation([SimpleLocation(3, 6, strand=+1),
...                       SimpleLocation(10, 13, strand=-1)])
>>> print(f.strand)
None
>>> len(f)
6
>>> list(f)
[3, 4, 5, 12, 11, 10]
```

The example above doing list(f) iterates over the coordinates within the feature. This allows you to use max and min on the location, to find the range covered:

```
>>> min(f)
3
>>> max(f)
12
```

More generally, you can use the compound location's start and end which give the full span covered, 0 <= start <= end <= full sequence length.

```
>>> f.start == min(f)
True
>>> f.end == max(f) + 1
True
```

This is consistent with the behavior of the SimpleLocation for a single region, where again the 'start' and 'end' do not necessarily give the biological start and end, but rather the 'minimal' and 'maximal' coordinate boundaries.

Note that adding locations provides a more intuitive method of construction:

```
>>> f = SimpleLocation(3, 6, strand=+1) + SimpleLocation(10, 13, strand=-1)
>>> len(f)
6
>>> list(f)
[3, 4, 5, 12, 11, 10]
```

__str__()

Return a representation of the CompoundLocation object (with python counting).

__repr__()

Represent the CompoundLocation object as string for debugging.

property strand

Overall strand of the compound location.

If all the parts have the same strand, that is returned. Otherwise for mixed strands, this returns None.

```
>>> from Bio.SeqFeature import SimpleLocation, CompoundLocation
>>> f1 = SimpleLocation(15, 17, strand=1)
>>> f2 = SimpleLocation(20, 30, strand=-1)
>>> f = f1 + f2
>>> f1.strand
1
>>> f2.strand
-1
>>> f.strand
>>> f.strand is None
True
```

If you set the strand of a CompoundLocation, this is applied to all the parts - use with caution:

```
>>> f.strand = 1
>>> f1.strand
1
>>> f2.strand
1
```

(continues on next page)

(continued from previous page)

```
>>> f.strand
1
```

__add__(*other*)

Combine locations, or shift the location by an integer offset.

```
>>> from Bio.SeqFeature import SimpleLocation
>>> f1 = SimpleLocation(15, 17) + SimpleLocation(20, 30)
>>> print(f1)
join{[15:17], [20:30]}
```

You can add another SimpleLocation:

```
>>> print(f1 + SimpleLocation(40, 50))
join{[15:17], [20:30], [40:50]}
>>> print(SimpleLocation(5, 10) + f1)
join{[5:10], [15:17], [20:30]}
```

You can also add another CompoundLocation:

```
>>> f2 = SimpleLocation(40, 50) + SimpleLocation(60, 70)
>>> print(f2)
join{[40:50], [60:70]}
>>> print(f1 + f2)
join{[15:17], [20:30], [40:50], [60:70]}
```

Also, as with the SimpleLocation, adding an integer shifts the location's coordinates by that offset:

```
>>> print(f1 + 100)
join{[115:117], [120:130]}
>>> print(200 + f1)
join{[215:217], [220:230]}
>>> print(f1 + (-5))
join{[10:12], [15:25]}
```

__radd__(*other*)

Add a feature to the left.

__contains__(*value*)

Check if an integer position is within the CompoundLocation object.

__nonzero__()

Return True regardless of the length of the feature.

This behavior is for backwards compatibility, since until the __len__ method was added, a SimpleLocation always evaluated as True.

Note that in comparison, Seq objects, strings, lists, etc, will all evaluate to False if they have length zero.

WARNING: The SimpleLocation may in future evaluate to False when its length is zero (in order to better match normal python behavior)!

__len__()

Return the length of the CompoundLocation object.

__iter__()

Iterate over the parent positions within the CompoundLocation object.

__eq__(*other*)

Check if all parts of CompoundLocation are equal to all parts of other CompoundLocation.

property start

Start location - left most (minimum) value, regardless of strand.

Read only, returns an integer like position object, possibly a fuzzy position.

For the special case of a CompoundLocation wrapping the origin of a circular genome, this will return zero.

property end

End location - right most (maximum) value, regardless of strand.

Read only, returns an integer like position object, possibly a fuzzy position.

For the special case of a CompoundLocation wrapping the origin of a circular genome this will match the genome length.

property ref

Not present in CompoundLocation, dummy method for API compatibility.

property ref_db

Not present in CompoundLocation, dummy method for API compatibility.

extract(*parent_sequence*, *references=None*)

Extract the sequence from supplied parent sequence using the CompoundLocation object.

The parent_sequence can be a Seq like object or a string, and will generally return an object of the same type. The exception to this is a MutableSeq as the parent sequence will return a Seq object. If the location refers to other records, they must be supplied in the optional dictionary references.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqFeature import SimpleLocation, CompoundLocation
>>> seq = Seq("MKQHKAMIVALIVICITAVVAAL")
>>> fl1 = SimpleLocation(2, 8)
>>> fl2 = SimpleLocation(10, 15)
>>> fl3 = CompoundLocation([fl1,fl2])
>>> fl3.extract(seq)
Seq('QHKAMILIVIC')
```

__abstractmethods__ = frozenset({})

__annotations__ = {}

__hash__ = None

class Bio.SeqFeature.**Position**

Bases: ABC

Abstract base class representing a position.

abstract __repr__()

Represent the Position object as a string for debugging.

static fromstring(*text*, *offset=0*)

Build a Position object from the text string.

For an end position, leave offset as zero (default):

```
>>> Position.fromstring("5")
ExactPosition(5)
```

For a start position, set offset to minus one (for Python counting):

```
>>> Position.fromstring("5", -1)
ExactPosition(4)
```

This also covers fuzzy positions:

```
>>> p = Position.fromstring("<5")
>>> p
BeforePosition(5)
>>> print(p)
<5
>>> int(p)
5
```

```
>>> Position.fromstring(">5")
AfterPosition(5)
```

By default assumes an end position, so note the integer behavior:

```
>>> p = Position.fromstring("one-of(5,8,11)")
>>> p
OneOfPosition(11, choices=[ExactPosition(5), ExactPosition(8),␣
↪ExactPosition(11)])
>>> print(p)
one-of(5,8,11)
>>> int(p)
11
```

```
>>> Position.fromstring("(8.10)")
WithinPosition(10, left=8, right=10)
```

Fuzzy start positions:

```
>>> p = Position.fromstring("<5", -1)
>>> p
BeforePosition(4)
>>> print(p)
<4
>>> int(p)
4
```

Notice how the integer behavior changes too!

```
>>> p = Position.fromstring("one-of(5,8,11)", -1)
>>> p
```

(continues on next page)

(continued from previous page)

```
OneOfPosition(4, choices=[ExactPosition(4), ExactPosition(7),␣
↪ExactPosition(10)])
>>> print(p)
one-of(4,7,10)
>>> int(p)
4
```

__abstractmethods__ = frozenset({'__repr__'})

__annotations__ = {}

class Bio.SeqFeature.**ExactPosition**(*position*, *extension=0*)

Bases: int, *Position*

Specify the specific position of a boundary.

Arguments:

- position - The position of the boundary.
- extension - An optional argument which must be zero since we don't have an extension. The argument is provided so that the same number of arguments can be passed to all position types.

In this case, there is no fuzziness associated with the position.

```
>>> p = ExactPosition(5)
>>> p
ExactPosition(5)
>>> print(p)
5
```

```
>>> isinstance(p, Position)
True
>>> isinstance(p, int)
True
```

Integer comparisons and operations should work as expected:

```
>>> p == 5
True
>>> p < 6
True
>>> p <= 5
True
>>> p + 10
ExactPosition(15)
```

static __new__(*cls*, *position*, *extension=0*)

Create an ExactPosition object.

__str__()

Return a representation of the ExactPosition object (with python counting).

__repr__()

Represent the ExactPosition object as a string for debugging.

__add__(*offset*)

Return a copy of the position object with its location shifted (PRIVATE).

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.SeqFeature.**UncertainPosition**(*position*, *extension=0*)

Bases: *ExactPosition*

Specify a specific position which is uncertain.

This is used in UniProt, e.g. ?222 for uncertain position 222, or in the XML format explicitly marked as uncertain. Does not apply to GenBank/EMBL.

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.SeqFeature.**UnknownPosition**

Bases: *Position*

Specify a specific position which is unknown (has no position).

This is used in UniProt, e.g. ? or in the XML as unknown.

__repr__()

Represent the UnknownPosition object as a string for debugging.

__hash__()

Return the hash value of the UnknownPosition object.

__add__(*offset*)

Return a copy of the position object with its location shifted (PRIVATE).

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.SeqFeature.**WithinPosition**(*position*, *left*, *right*)

Bases: int, *Position*

Specify the position of a boundary within some coordinates.

Arguments: - position - The default integer position - left - The start (left) position of the boundary - right - The end (right) position of the boundary

This allows dealing with a location like ((11.14)..100). This indicates that the start of the sequence is somewhere between 11 and 14. Since this is a start coordinate, it should act like it is at position 11 (or in Python counting, 10).

```
>>> p = WithinPosition(10, 10, 13)
>>> p
WithinPosition(10, left=10, right=13)
>>> print(p)
(10.13)
>>> int(p)
10
```

Basic integer comparisons and operations should work as though this were a plain integer:

```
>>> p == 10
True
>>> p in [9, 10, 11]
True
>>> p < 11
True
>>> p + 10
WithinPosition(20, left=20, right=23)
```

```
>>> isinstance(p, WithinPosition)
True
>>> isinstance(p, Position)
True
>>> isinstance(p, int)
True
```

Note this also applies for comparison to other position objects, where again the integer behavior is used:

```
>>> p == 10
True
>>> p == ExactPosition(10)
True
>>> p == BeforePosition(10)
True
>>> p == AfterPosition(10)
True
```

If this were an end point, you would want the position to be 13 (the right/larger value, not the left/smaller value as above):

```
>>> p2 = WithinPosition(13, 10, 13)
>>> p2
WithinPosition(13, left=10, right=13)
>>> print(p2)
(10.13)
>>> int(p2)
13
>>> p2 == 13
True
>>> p2 == ExactPosition(13)
True
```

static __new__(*cls*, *position*, *left*, *right*)

Create a WithinPosition object.

__getnewargs__()

Return the arguments accepted by __new__.

Necessary to allow pickling and unpickling of class instances.

__repr__()

Represent the WithinPosition object as a string for debugging.

__str__()

Return a representation of the WithinPosition object (with python counting).

__add__(*offset*)

Return a copy of the position object with its location shifted.

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.SeqFeature.**BetweenPosition**(*position*, *left*, *right*)

Bases: int, *Position*

Specify the position of a boundary between two coordinates (OBSOLETE?).

Arguments:

- position - The default integer position
- left - The start (left) position of the boundary
- right - The end (right) position of the boundary

This allows dealing with a position like 123^456. This indicates that the start of the sequence is somewhere between 123 and 456. It is up to the parser to set the position argument to either boundary point (depending on if this is being used as a start or end of the feature). For example as a feature end:

```
>>> p = BetweenPosition(456, 123, 456)
>>> p
BetweenPosition(456, left=123, right=456)
>>> print(p)
(123^456)
>>> int(p)
456
```

Integer equality and comparison use the given position,

```
>>> p == 456
True
>>> p in [455, 456, 457]
True
>>> p > 300
True
```

The old legacy properties of position and extension give the starting/lower/left position as an integer, and the distance to the ending/higher/right position as an integer. Note that the position object will act like either the left or the right end-point depending on how it was created:

```
>>> p2 = BetweenPosition(123, left=123, right=456)
>>> int(p) == int(p2)
False
>>> p == 456
True
>>> p2 == 123
True
```

Note this potentially surprising behavior:

```
>>> BetweenPosition(123, left=123, right=456) == ExactPosition(123)
True
>>> BetweenPosition(123, left=123, right=456) == BeforePosition(123)
```

(continues on next page)

(continued from previous page)

```
True
>>> BetweenPosition(123, left=123, right=456) == AfterPosition(123)
True
```

i.e. For equality (and sorting) the position objects behave like integers.

static __new__(*cls*, *position*, *left*, *right*)

Create a new instance in BetweenPosition object.

__getnewargs__()

Return the arguments accepted by __new__.

Necessary to allow pickling and unpickling of class instances.

__repr__()

Represent the BetweenPosition object as a string for debugging.

__str__()

Return a representation of the BetweenPosition object (with python counting).

__add__(*offset*)

Return a copy of the position object with its location shifted (PRIVATE).

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.SeqFeature.**BeforePosition**(*position*, *extension=0*)

Bases: `int`, *Position*

Specify a position where the actual location occurs before it.

Arguments:

- position - The upper boundary of where the location can occur.
- extension - An optional argument which must be zero since we don't have an extension. The argument is provided so that the same number of arguments can be passed to all position types.

This is used to specify positions like (<10..100) where the location occurs somewhere before position 10.

```
>>> p = BeforePosition(5)
>>> p
BeforePosition(5)
>>> print(p)
<5
>>> int(p)
5
>>> p + 10
BeforePosition(15)
```

Note this potentially surprising behavior:

```
>>> p == ExactPosition(5)
True
>>> p == AfterPosition(5)
True
```

Just remember that for equality and sorting the position objects act like integers.

static __new__(*cls*, *position*, *extension=0*)

Create a new instance in BeforePosition object.

__repr__()

Represent the location as a string for debugging.

__str__()

Return a representation of the BeforePosition object (with python counting).

__add__(*offset*)

Return a copy of the position object with its location shifted (PRIVATE).

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.SeqFeature.**AfterPosition**(*position*, *extension=0*)

Bases: int, *Position*

Specify a position where the actual location is found after it.

Arguments:

- position - The lower boundary of where the location can occur.
- extension - An optional argument which must be zero since we don't have an extension. The argument is provided so that the same number of arguments can be passed to all position types.

This is used to specify positions like (>10..100) where the location occurs somewhere after position 10.

```
>>> p = AfterPosition(7)
>>> p
AfterPosition(7)
>>> print(p)
>7
>>> int(p)
7
>>> p + 10
AfterPosition(17)
```

```
>>> isinstance(p, AfterPosition)
True
>>> isinstance(p, Position)
True
>>> isinstance(p, int)
True
```

Note this potentially surprising behavior:

```
>>> p == ExactPosition(7)
True
>>> p == BeforePosition(7)
True
```

Just remember that for equality and sorting the position objects act like integers.

static __new__(*cls*, *position*, *extension=0*)

Create a new instance of the AfterPosition object.

__repr__()

Represent the location as a string for debugging.

__str__()

Return a representation of the AfterPosition object (with python counting).

__add__(*offset*)

Return a copy of the position object with its location shifted (PRIVATE).

__abstractmethods__ = frozenset({})

__annotations__ = {}

class Bio.SeqFeature.**OneOfPosition**(*position*, *choices*)

Bases: int, *Position*

Specify a position where the location can be multiple positions.

This models the GenBank 'one-of(1888,1901)' function, and tries to make this fit within the Biopython Position models. If this was a start position it should act like 1888, but as an end position 1901.

```
>>> p = OneOfPosition(1888, [ExactPosition(1888), ExactPosition(1901)])
>>> p
OneOfPosition(1888, choices=[ExactPosition(1888), ExactPosition(1901)])
>>> int(p)
1888
```

Integer comparisons and operators act like using int(p),

```
>>> p == 1888
True
>>> p <= 1888
True
>>> p > 1888
False
>>> p + 100
OneOfPosition(1988, choices=[ExactPosition(1988), ExactPosition(2001)])
```

```
>>> isinstance(p, OneOfPosition)
True
>>> isinstance(p, Position)
True
>>> isinstance(p, int)
True
```

static __new__(*cls*, *position*, *choices*)

Initialize with a set of possible positions.

choices is a list of Position derived objects, specifying possible locations.

position is an integer specifying the default behavior.

__abstractmethods__ = frozenset({})

__annotations__ = {}

__getnewargs__()

Return the arguments accepted by __new__.

Necessary to allow pickling and unpickling of class instances.

__repr__()

Represent the OneOfPosition object as a string for debugging.

__str__()

Return a representation of the OneOfPosition object (with python counting).

__add__(*offset*)

Return a copy of the position object with its location shifted (PRIVATE).

28.2.8 Bio.SeqRecord module

Represent a Sequence Record, a sequence with annotation.

class Bio.SeqRecord.**SeqRecord**(*seq: Seq | MutableSeq | None*, *id: str | None = '<unknown id>'*, *name: str = '<unknown name>'*, *description: str = '<unknown description>'*, *dbxrefs: list[str] | None = None*, *features: list[SeqFeature] | None = None*, *annotations: dict[str, str | int] | None = None*, *letter_annotations: dict[str, Sequence[Any]] | None = None*)

Bases: object

A SeqRecord object holds a sequence and information about it.

Main attributes:

- id - Identifier such as a locus tag (string)
- seq - The sequence itself (Seq object or similar)

Additional attributes:

- name - Sequence name, e.g. gene name (string)
- description - Additional text (string)
- dbxrefs - List of database cross references (list of strings)
- features - Any (sub)features defined (list of SeqFeature objects)
- annotations - Further information about the whole sequence (dictionary). Most entries are strings, or lists of strings.
- letter_annotations - Per letter/symbol annotation (restricted dictionary). This holds Python sequences (lists, strings or tuples) whose length matches that of the sequence. A typical use would be to hold a list of integers representing sequencing quality scores, or a string representing the secondary structure.

You will typically use Bio.SeqIO to read in sequences from files as SeqRecord objects. However, you may want to create your own SeqRecord objects directly (see the __init__ method for further details):

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> record = SeqRecord(Seq("MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF"),
...                    id="YP_025292.1", name="HokC",
...                    description="toxic membrane protein")
```

(continues on next page)

(continued from previous page)

```
>>> print(record)
ID: YP_025292.1
Name: HokC
Description: toxic membrane protein
Number of features: 0
Seq('MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF')
```

If you want to save SeqRecord objects to a sequence file, use Bio.SeqIO for this. For the special case where you want the SeqRecord turned into a string in a particular file format there is a format method which uses Bio.SeqIO internally:

```
>>> print(record.format("fasta"))
>YP_025292.1 toxic membrane protein
MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF
```

You can also do things like slicing a SeqRecord, checking its length, etc

```
>>> len(record)
44
>>> edited = record[:10] + record[11:]
>>> print(edited.seq)
MKQHKAMIVAIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF
>>> print(record.seq)
MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF
```

__init__(*seq: Seq | MutableSeq | None*, *id: str | None = '<unknown id>'*, *name: str = '<unknown name>'*, *description: str = '<unknown description>'*, *dbxrefs: list[str] | None = None*, *features: list[SeqFeature] | None = None*, *annotations: dict[str, str | int] | None = None*, *letter_annotations: dict[str, Sequence[Any]] | None = None*) → None

Create a SeqRecord.

Arguments:

- seq - Sequence, required (Seq or MutableSeq)
- id - Sequence identifier, recommended (string)
- name - Sequence name, optional (string)
- description - Sequence description, optional (string)
- dbxrefs - Database cross references, optional (list of strings)
- features - Any (sub)features, optional (list of SeqFeature objects)
- annotations - Dictionary of annotations for the whole sequence
- letter_annotations - Dictionary of per-letter-annotations, values should be strings, list or tuples of the same length as the full sequence.

You will typically use Bio.SeqIO to read in sequences from files as SeqRecord objects. However, you may want to create your own SeqRecord objects directly.

Note that while an id is optional, we strongly recommend you supply a unique id string for each record. This is especially important if you wish to write your sequences to a file.

You can create a 'blank' SeqRecord object, and then populate the attributes later.

dbxrefs: list[str]

annotations: dict[str, str | int]

property letter_annotations

Dictionary of per-letter-annotation for the sequence.

For example, this can hold quality scores used in FASTQ or QUAL files. Consider this example using Bio.SeqIO to read in an example Solexa variant FASTQ file as a SeqRecord:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("Quality/solexa_faked.fastq", "fastq-solexa")
>>> print("%s %s" % (record.id, record.seq))
slxa_0001_1_0001_01 ACGTACGTACGTACGTACGTACGTACGTACGTACGTNNNNNN
>>> print(list(record.letter_annotations))
['solexa_quality']
>>> print(record.letter_annotations["solexa_quality"])
[40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21,
↪ 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1,
↪ -2, -3, -4, -5]
```

The letter_annotations get sliced automatically if you slice the parent SeqRecord, for example taking the last ten bases:

```
>>> sub_record = record[-10:]
>>> print("%s %s" % (sub_record.id, sub_record.seq))
slxa_0001_1_0001_01 ACGTNNNNNN
>>> print(sub_record.letter_annotations["solexa_quality"])
[4, 3, 2, 1, 0, -1, -2, -3, -4, -5]
```

Any python sequence (i.e. list, tuple or string) can be recorded in the SeqRecord's letter_annotations dictionary as long as the length matches that of the SeqRecord's sequence. e.g.

```
>>> len(sub_record.letter_annotations)
1
>>> sub_record.letter_annotations["dummy"] = "abcdefghij"
>>> len(sub_record.letter_annotations)
2
```

You can delete entries from the letter_annotations dictionary as usual:

```
>>> del sub_record.letter_annotations["solexa_quality"]
>>> sub_record.letter_annotations
{'dummy': 'abcdefghij'}
```

You can completely clear the dictionary easily as follows:

```
>>> sub_record.letter_annotations = {}
>>> sub_record.letter_annotations
{}
```

Note that if replacing the record's sequence with a sequence of a different length you must first clear the letter_annotations dict.

property seq

The sequence itself, as a Seq or MutableSeq object.

__getitem__(*index: int*) → str

__getitem__(*index: slice*) → *SeqRecord*

Return a sub-sequence or an individual letter.

Slicing, e.g. my_record[5:10], returns a new SeqRecord for that sub-sequence with some annotation preserved as follows:

- The name, id and description are kept as-is.
- Any per-letter-annotations are sliced to match the requested sub-sequence.
- Unless a stride is used, all those features which fall fully within the subsequence are included (with their locations adjusted accordingly). If you want to preserve any truncated features (e.g. GenBank/EMBL source features), you must explicitly add them to the new SeqRecord yourself.
- With the exception of any molecule type, the annotations dictionary and the dbxrefs list are not used for the new SeqRecord, as in general they may not apply to the subsequence. If you want to preserve them, you must explicitly copy them to the new SeqRecord yourself.

Using an integer index, e.g. my_record[5] is shorthand for extracting that letter from the sequence, my_record.seq[5].

For example, consider this short protein and its secondary structure as encoded by the PDB (e.g. H for alpha helices), plus a simple feature for its histidine self phosphorylation site:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.SeqFeature import SeqFeature, SimpleLocation
>>> rec = SeqRecord(Seq("MAAGVKQLADDRTLLMAGVSHDLRTPLTRIRLAT"
...                     "EMMSEQDGYLAESINKDIEECNAIIEQFIDYLR"),
...                 id="1JOY", name="EnvZ",
...                 description="Homodimeric domain of EnvZ from E. coli")
>>> rec.letter_annotations["secondary_structure"] = "  S ␣
↪SSSSSSHHHHHTTTHHHHHHHHHHHHHHHHHHHHHHHTHHHHHHHHHHHHHHHHHHHHTT  "
>>> rec.features.append(SeqFeature(SimpleLocation(20, 21),
...                     type = "Site"))
```

Now let's have a quick look at the full record,

```
>>> print(rec)
ID: 1JOY
Name: EnvZ
Description: Homodimeric domain of EnvZ from E. coli
Number of features: 1
Per letter annotation for: secondary_structure
Seq('MAAGVKQLADDRTLLMAGVSHDLRTPLTRIRLATEMMSEQDGYLAESINKDIEE...YLR')
>>> rec.letter_annotations["secondary_structure"]
'  S  SSSSSSHHHHHTTTHHHHHHHHHHHHHHHHHHHHHHHTHHHHHHHHHHHHHHHHHHHHTT  '
>>> print(rec.features[0].location)
[20:21]
```

Now let's take a sub sequence, here chosen as the first (fractured) alpha helix which includes the histidine phosphorylation site:

```
>>> sub = rec[11:41]
>>> print(sub)
ID: 1JOY
```

(continues on next page)

(continued from previous page)

```
Name: EnvZ
Description: Homodimeric domain of EnvZ from E. coli
Number of features: 1
Per letter annotation for: secondary_structure
Seq('RTLLMAGVSHDLRTPLTRIRLATEMMSEQD')
>>> sub.letter_annotations["secondary_structure"]
'HHHHHTTTHHHHHHHHHHHHHHHHHHHHHH'
>>> print(sub.features[0].location)
[9:10]
```

You can also of course omit the start or end values, for example to get the first ten letters only:

```
>>> print(rec[:10])
ID: 1JOY
Name: EnvZ
Description: Homodimeric domain of EnvZ from E. coli
Number of features: 0
Per letter annotation for: secondary_structure
Seq('MAAGVKQLAD')
```

Or for the last ten letters:

```
>>> print(rec[-10:])
ID: 1JOY
Name: EnvZ
Description: Homodimeric domain of EnvZ from E. coli
Number of features: 0
Per letter annotation for: secondary_structure
Seq('IIEQFIDYLR')
```

If you omit both, then you get a copy of the original record (although lacking the annotations and dbxrefs):

```
>>> print(rec[:])
ID: 1JOY
Name: EnvZ
Description: Homodimeric domain of EnvZ from E. coli
Number of features: 1
Per letter annotation for: secondary_structure
Seq('MAAGVKQLADDRTLLMAGVSHDLRTPLTRIRLATEMMSEQDGYLAESINKDIEE...YLR')
```

Finally, indexing with a simple integer is shorthand for pulling out that letter from the sequence directly:

```
>>> rec[5]
'K'
>>> rec.seq[5]
'K'
```

__iter__() → Iterable[*Seq* | *MutableSeq*]

Iterate over the letters in the sequence.

For example, using Bio.SeqIO to read in a protein FASTA file:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("Fasta/loveliesbleeding.pro", "fasta")
>>> for amino in record:
...     print(amino)
...     if amino == "L": break
X
A
G
L
>>> print(record.seq[3])
L
```

This is just a shortcut for iterating over the sequence directly:

```
>>> for amino in record.seq:
...     print(amino)
...     if amino == "L": break
X
A
G
L
>>> print(record.seq[3])
L
```

Note that this does not facilitate iteration together with any per-letter-annotation. However, you can achieve that using the python zip function on the record (or its sequence) and the relevant per-letter-annotation:

```
>>> from Bio import SeqIO
>>> rec = SeqIO.read("Quality/solexa_faked.fastq", "fastq-solexa")
>>> print("%s %s" % (rec.id, rec.seq))
slxa_0001_1_0001_01 ACGTACGTACGTACGTACGTACGTACGTACGTACGTNNNNNN
>>> print(list(rec.letter_annotations))
['solexa_quality']
>>> for nuc, qual in zip(rec, rec.letter_annotations["solexa_quality"]):
...     if qual > 35:
...         print("%s %i" % (nuc, qual))
A 40
C 39
G 38
T 37
A 36
```

You may agree that using zip(rec.seq, ...) is more explicit than using zip(rec, ...) as shown above.

__contains__(*char: str*) → bool

Implement the 'in' keyword, searches the sequence.

e.g.

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("Fasta/sweetpea.nu", "fasta")
>>> "GAATTC" in record
False
```

(continues on next page)

(continued from previous page)

```
>>> "AAA" in record
True
```

This essentially acts as a proxy for using "in" on the sequence:

```
>>> "GAATTC" in record.seq
False
>>> "AAA" in record.seq
True
```

Note that you can also use Seq objects as the query,

```
>>> from Bio.Seq import Seq
>>> Seq("AAA") in record
True
```

See also the Seq object's __contains__ method.

__bytes__() → bytes

__str__() → str

Return a human readable summary of the record and its annotation (string).

The python built in function str works by calling the object's __str__ method. e.g.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> record = SeqRecord(Seq("MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF"),
...                    id="YP_025292.1", name="HokC",
...                    description="toxic membrane protein, small")
>>> print(str(record))
ID: YP_025292.1
Name: HokC
Description: toxic membrane protein, small
Number of features: 0
Seq('MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF')
```

In this example you don't actually need to call str explicitly, as the print command does this automatically:

```
>>> print(record)
ID: YP_025292.1
Name: HokC
Description: toxic membrane protein, small
Number of features: 0
Seq('MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF')
```

Note that long sequences are shown truncated.

__repr__() → str

Return a concise summary of the record for debugging (string).

The python built in function repr works by calling the object's __repr__ method. e.g.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
```

(continues on next page)

(continued from previous page)

```
>>> rec = SeqRecord(Seq("MASRGVNKVILVGNLGQDPEVRYMPNGGAVANITLATSESWRDKAT"
...                      "GEMKEQTEWHRVVLFGKLAEVASEYLRKGSQVYIEGQLRTRKWTDQ"
...                      "SGQDRYTTEVVVNVGGTMQMLGGRQGGGAPAGGNIGGGQPQGGWGQ"
...                      "PQQPQGGNQFSGGAQSRPQQSAPAAPSNEPPMDFDDDIPF"),
...                  id="NP_418483.1", name="b4059",
...                  description="ssDNA-binding protein",
...                  dbxrefs=["ASAP:13298", "GI:16131885", "GeneID:948570"])
>>> print(repr(rec))
SeqRecord(seq=Seq('MASRGVNKVILVGNLGQDPEVRYMPNGGAVANITLATSESWRDKATGEMKEQTE...IPF
↪'), id='NP_418483.1', name='b4059', description='ssDNA-binding protein',␣
↪dbxrefs=['ASAP:13298', 'GI:16131885', 'GeneID:948570'])
```

At the python prompt you can also use this shorthand:

```
>>> rec
SeqRecord(seq=Seq('MASRGVNKVILVGNLGQDPEVRYMPNGGAVANITLATSESWRDKATGEMKEQTE...IPF
↪'), id='NP_418483.1', name='b4059', description='ssDNA-binding protein',␣
↪dbxrefs=['ASAP:13298', 'GI:16131885', 'GeneID:948570'])
```

Note that long sequences are shown truncated. Also note that any annotations, letter_annotations and features are not shown (as they would lead to a very long string).

format(*format: str*) → str

Return the record as a string in the specified file format.

The format should be a lower case string supported as an output format by Bio.SeqIO, which is used to turn the SeqRecord into a string. e.g.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> record = SeqRecord(Seq("MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF"),
...                    id="YP_025292.1", name="HokC",
...                    description="toxic membrane protein")
>>> record.format("fasta")
'>YP_025292.1 toxic membrane protein\
↪nMKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF\n'
>>> print(record.format("fasta"))
>YP_025292.1 toxic membrane protein
MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF
```

The Python print function automatically appends a new line, meaning in this example a blank line is shown. If you look at the string representation you can see there is a trailing new line (shown as slash n) which is important when writing to a file or if concatenating multiple sequence strings together.

Note that this method will NOT work on every possible file format supported by Bio.SeqIO (e.g. some are for multiple sequences only, and binary formats are not supported).

__format__(*format_spec: str*) → str

Return the record as a string in the specified file format.

This method supports the Python format() function and f-strings. The format_spec should be a lower case string supported by Bio.SeqIO as a text output file format. Requesting a binary file format raises a ValueError. e.g.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> record = SeqRecord(Seq("MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF"),
...                    id="YP_025292.1", name="HokC",
...                    description="toxic membrane protein")
...
>>> format(record, "fasta")
'>YP_025292.1 toxic membrane protein\
↪nMKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF\n'
>>> print(f"Here is {record.id} in FASTA format:\n{record:fasta}")
Here is YP_025292.1 in FASTA format:
>YP_025292.1 toxic membrane protein
MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF
```

See also the SeqRecord's format() method.

__len__() → int

Return the length of the sequence.

For example, using Bio.SeqIO to read in a FASTA nucleotide file:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("Fasta/sweetpea.nu", "fasta")
>>> len(record)
309
>>> len(record.seq)
309
```

__lt__(*other: Any*) → NoReturn

Define the less-than operand (not implemented).

__le__(*other: Any*) → NoReturn

Define the less-than-or-equal-to operand (not implemented).

__eq__(*other: object*) → NoReturn

Define the equal-to operand (not implemented).

__ne__(*other: object*) → NoReturn

Define the not-equal-to operand (not implemented).

__gt__(*other: Any*) → NoReturn

Define the greater-than operand (not implemented).

__ge__(*other: Any*) → NoReturn

Define the greater-than-or-equal-to operand (not implemented).

__bool__() → bool

Boolean value of an instance of this class (True).

This behaviour is for backwards compatibility, since until the __len__ method was added, a SeqRecord always evaluated as True.

Note that in comparison, a Seq object will evaluate to False if it has a zero length sequence.

WARNING: The SeqRecord may in future evaluate to False when its sequence is of zero length (in order to better match the Seq object behaviour)!

__add__(*other:* SeqRecord | Seq | MutableSeq | *str*) → *SeqRecord*

Add another sequence or string to this sequence.

The other sequence can be a SeqRecord object, a Seq object (or similar, e.g. a MutableSeq) or a plain Python string. If you add a plain string or a Seq (like) object, the new SeqRecord will simply have this appended to the existing data. However, any per letter annotation will be lost:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("Quality/solexa_faked.fastq", "fastq-solexa")
>>> print("%s %s" % (record.id, record.seq))
slxa_0001_1_0001_01 ACGTACGTACGTACGTACGTACGTACGTACGTACGTNNNNNN
>>> print(list(record.letter_annotations))
['solexa_quality']
```

```
>>> new = record + "ACT"
>>> print("%s %s" % (new.id, new.seq))
slxa_0001_1_0001_01 ACGTACGTACGTACGTACGTACGTACGTACGTACGTNNNNNNACT
>>> print(list(new.letter_annotations))
[]
```

The new record will attempt to combine the annotation, but for any ambiguities (e.g. different names) it defaults to omitting that annotation.

```
>>> from Bio import SeqIO
>>> with open("GenBank/pBAD30.gb") as handle:
...     plasmid = SeqIO.read(handle, "gb")
>>> print("%s %i" % (plasmid.id, len(plasmid)))
pBAD30 4923
```

Now let's cut the plasmid into two pieces, and join them back up the other way round (i.e. shift the starting point on this plasmid, have a look at the annotated features in the original file to see why this particular split point might make sense):

```
>>> left = plasmid[:3765]
>>> right = plasmid[3765:]
>>> new = right + left
>>> print("%s %i" % (new.id, len(new)))
pBAD30 4923
>>> str(new.seq) == str(right.seq + left.seq)
True
>>> len(new.features) == len(left.features) + len(right.features)
True
```

When we add the left and right SeqRecord objects, their annotation is all consistent, so it is all conserved in the new SeqRecord:

```
>>> new.id == left.id == right.id == plasmid.id
True
>>> new.name == left.name == right.name == plasmid.name
True
>>> new.description == plasmid.description
True
>>> new.annotations == left.annotations == right.annotations
True
```

(continues on next page)

(continued from previous page)

```
>>> new.letter_annotations == plasmid.letter_annotations
True
>>> new.dbxrefs == left.dbxrefs == right.dbxrefs
True
```

However, we should point out that when we sliced the SeqRecord, any annotations dictionary or dbxrefs list entries were lost. You can explicitly copy them like this:

```
>>> new.annotations = plasmid.annotations.copy()
>>> new.dbxrefs = plasmid.dbxrefs[:]
```

__radd__(*other:* Seq | MutableSeq | *str*) → *SeqRecord*

Add another sequence or string to this sequence (from the left).

This method handles adding a Seq object (or similar, e.g. MutableSeq) or a plain Python string (on the left) to a SeqRecord (on the right). See the __add__ method for more details, but for example:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("Quality/solexa_faked.fastq", "fastq-solexa")
>>> print("%s %s" % (record.id, record.seq))
slxa_0001_1_0001_01 ACGTACGTACGTACGTACGTACGTACGTACGTACGTNNNNNN
>>> print(list(record.letter_annotations))
['solexa_quality']
```

```
>>> new = "ACT" + record
>>> print("%s %s" % (new.id, new.seq))
slxa_0001_1_0001_01 ACTACGTACGTACGTACGTACGTACGTACGTACGTACGTNNNNNN
>>> print(list(new.letter_annotations))
[]
```

count(*sub*, *start=None*, *end=None*)

Return the number of non-overlapping occurrences of sub in seq[start:end].

Optional arguments start and end are interpreted as in slice notation. This method behaves as the count method of Python strings.

upper() → *SeqRecord*

Return a copy of the record with an upper case sequence.

All the annotation is preserved unchanged. e.g.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> record = SeqRecord(Seq("acgtACGT"), id="Test",
...                    description = "Made up for this example")
>>> record.letter_annotations["phred_quality"] = [1, 2, 3, 4, 5, 6, 7, 8]
>>> print(record.upper().format("fastq"))
@Test Made up for this example
ACGTACGT
+
"#$%&'()
```

Naturally, there is a matching lower method:

```
>>> print(record.lower().format("fastq"))
@Test Made up for this example
acgtacgt
+
"#$%&'()
```

__annotations__ = {'_per_letter_annotations': '_RestrictedDict', 'annotations': dict[str, typing.Union[str, int]], 'dbxrefs': list[str]}

__hash__ = None

lower() → *SeqRecord*

Return a copy of the record with a lower case sequence.

All the annotation is preserved unchanged. e.g.

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("Fasta/aster.pro", "fasta")
>>> print(record.format("fasta"))
>gi|3298468|dbj|BAA31520.1| SAMIPF
GGHVNPAVTFGAFVGGNITLLRGIVYIIAQLLGSTVACLLLKFVTNDMAVGVFSLSAGVG
VTNALVFEIVMTFGLVYTVYATAIDPKKGSLGTIAPIAIGFIVGANI

>>> print(record.lower().format("fasta"))
>gi|3298468|dbj|BAA31520.1| SAMIPF
gghvnpavtfgafvggnitllrgivyiiaqllgstvaclllkfvtndmavgvfslsagvg
vtnalvfeivmtfglvytvyataidpkkgslgtiapiaigfivgani
```

To take a more annotation rich example,

```
>>> from Bio import SeqIO
>>> old = SeqIO.read("EMBL/TRBG361.embl", "embl")
>>> len(old.features)
3
>>> new = old.lower()
>>> len(old.features) == len(new.features)
True
>>> old.annotations["organism"] == new.annotations["organism"]
True
>>> old.dbxrefs == new.dbxrefs
True
```

isupper()

Return True if all ASCII characters in the record's sequence are uppercase.

If there are no cased characters, the method returns False.

islower()

Return True if all ASCII characters in the record's sequence are lowercase.

If there are no cased characters, the method returns False.

reverse_complement(*id: bool = False*, *name: bool = False*, *description: bool = False*, *features: bool = True*, *annotations: bool = False*, *letter_annotations: bool = True*, *dbxrefs: bool = False*) → *SeqRecord*

Return new SeqRecord with reverse complement sequence.

By default the new record does NOT preserve the sequence identifier, name, description, general annotation or database cross-references - these are unlikely to apply to the reversed sequence.

You can specify the returned record's id, name and description as strings, or True to keep that of the parent, or False for a default.

You can specify the returned record's features with a list of SeqFeature objects, or True to keep that of the parent, or False to omit them. The default is to keep the original features (with the strand and locations adjusted).

You can also specify both the returned record's annotations and letter_annotations as dictionaries, True to keep that of the parent, or False to omit them. The default is to keep the original annotations (with the letter annotations reversed).

To show what happens to the pre-letter annotations, consider an example Solexa variant FASTQ file with a single entry, which we'll read in as a SeqRecord:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("Quality/solexa_faked.fastq", "fastq-solexa")
>>> print("%s %s" % (record.id, record.seq))
slxa_0001_1_0001_01 ACGTACGTACGTACGTACGTACGTACGTACGTACGTNNNNNN
>>> print(list(record.letter_annotations))
['solexa_quality']
>>> print(record.letter_annotations["solexa_quality"])
[40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21,
↪ 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1,
↪ -2, -3, -4, -5]
```

Now take the reverse complement, here we explicitly give a new identifier (the old identifier with a suffix):

```
>>> rc_record = record.reverse_complement(id=record.id + "_rc")
>>> print("%s %s" % (rc_record.id, rc_record.seq))
slxa_0001_1_0001_01_rc NNNNNNACGTACGTACGTACGTACGTACGTACGTACGTACGT
```

Notice that the per-letter-annotations have also been reversed, although this may not be appropriate for all cases.

```
>>> print(rc_record.letter_annotations["solexa_quality"])
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,␣
↪17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,␣
↪36, 37, 38, 39, 40]
```

Now for the features, we need a different example. Parsing a GenBank file is probably the easiest way to get an nice example with features in it...

```
>>> from Bio import SeqIO
>>> with open("GenBank/pBAD30.gb") as handle:
...     plasmid = SeqIO.read(handle, "gb")
>>> print("%s %i" % (plasmid.id, len(plasmid)))
pBAD30 4923
>>> plasmid.seq
Seq('GCTAGCGGAGTGTATACTGGCTTACTATGTTGGCACTGATGAGGGTGTCAGTGA...ATG')
>>> len(plasmid.features)
13
```

Now, let's take the reverse complement of this whole plasmid:

```
>>> rc_plasmid = plasmid.reverse_complement(id=plasmid.id+"_rc")
>>> print("%s %i" % (rc_plasmid.id, len(rc_plasmid)))
pBAD30_rc 4923
>>> rc_plasmid.seq
Seq('CATGGGCAAATATTATACGCAAGGCGACAAGGTGCTGATGCCGCTGGCGATTCA...AGC')
>>> len(rc_plasmid.features)
13
```

Let's compare the first CDS feature - it has gone from being the second feature (index 1) to the second last feature (index -2), its strand has changed, and the location switched round.

```
>>> print(plasmid.features[1])
type: CDS
location: [1081:1960](-)
qualifiers:
    Key: label, Value: ['araC']
    Key: note, Value: ['araC regulator of the arabinose BAD promoter']
    Key: vntifkey, Value: ['4']

>>> print(rc_plasmid.features[-2])
type: CDS
location: [2963:3842](+)
qualifiers:
    Key: label, Value: ['araC']
    Key: note, Value: ['araC regulator of the arabinose BAD promoter']
    Key: vntifkey, Value: ['4']
```

You can check this new location, based on the length of the plasmid:

```
>>> len(plasmid) - 1081
3842
>>> len(plasmid) - 1960
2963
```

Note that if the SeqFeature annotation includes any strand specific information (e.g. base changes for a SNP), this information is not amended, and would need correction after the reverse complement.

Note trying to reverse complement a protein SeqRecord raises an exception:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> protein_rec = SeqRecord(Seq("MAIVMGR"), id="Test",
...                         annotations={"molecule_type": "protein"})
>>> protein_rec.reverse_complement()
Traceback (most recent call last):
  ...
ValueError: Proteins do not have complements!
```

If you have RNA without any U bases, it must be annotated as RNA otherwise it will be treated as DNA by default with A mapped to T:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> rna1 = SeqRecord(Seq("ACG"), id="Test")
```

(continues on next page)

(continued from previous page)

```
>>> rna2 = SeqRecord(Seq("ACG"), id="Test", annotations={"molecule_type": "RNA"}
↪)
>>> print(rna1.reverse_complement(id="RC", description="unk").format("fasta"))
>RC unk
CGT

>>> print(rna2.reverse_complement(id="RC", description="RNA").format("fasta"))
>RC RNA
CGU
```

Also note you can reverse complement a SeqRecord using a MutableSeq:

```
>>> from Bio.Seq import MutableSeq
>>> from Bio.SeqRecord import SeqRecord
>>> rec = SeqRecord(MutableSeq("ACGT"), id="Test")
>>> rec.seq[0] = "T"
>>> print("%s %s" % (rec.id, rec.seq))
Test TCGT
>>> rc = rec.reverse_complement(id=True)
>>> print("%s %s" % (rc.id, rc.seq))
Test ACGA
```

translate(*table: str = 'Standard'*, *stop_symbol: str = '*'*, *to_stop: bool = False*, *cds: bool = False*, *gap: str | None = None*, *id: bool = False*, *name: bool = False*, *description: bool = False*, *features: bool = False*, *annotations: bool = False*, *letter_annotations: bool = False*, *dbxrefs: bool = False*) → *SeqRecord*

Return new SeqRecord with translated sequence.

This calls the record's .seq.translate() method (which describes the translation related arguments, like table for the genetic code),

By default the new record does NOT preserve the sequence identifier, name, description, general annotation or database cross-references - these are unlikely to apply to the translated sequence.

You can specify the returned record's id, name and description as strings, or True to keep that of the parent, or False for a default.

You can specify the returned record's features with a list of SeqFeature objects, or False (default) to omit them.

You can also specify both the returned record's annotations and letter_annotations as dictionaries, True to keep that of the parent (annotations only), or False (default) to omit them.

e.g. Loading a FASTA gene and translating it,

```
>>> from Bio import SeqIO
>>> gene_record = SeqIO.read("Fasta/sweetpea.nu", "fasta")
>>> print(gene_record.format("fasta"))
>gi|3176602|gb|U78617.1|LOU78617 Lathyrus odoratus phytochrome A (PHYA) gene,␣
↪partial cds
CAGGCTGCGCGGTTTCTATTTATGAAGAACAAGGTCCGTATGATAGTTGATTGTCATGCA
AAACATGTGAAGGTTCTTCAAGACGAAAAACTCCCATTTGATTTGACTCTGTGCGGTTCG
ACCTTAAGAGCTCCACATAGTTGCCATTTGCAGTACATGGCTAACATGGATTCAATTGCT
TCATTGGTTATGGCAGTGGTCGTCAATGACAGCGATGAAGATGGAGATAGCCGTGACGCA
```

(continues on next page)

(continued from previous page)

```
GTTCTACCACAAAAGAAAAAGAGACTTTGGGGTTTGGTAGTTTGTCATAACACTACTCCG
AGGTTTGTT
```

And now translating the record, specifying the new ID and description:

```
>>> protein_record = gene_record.translate(table=11,
...                                        id="phya",
...                                        description="translation")
>>> print(protein_record.format("fasta"))
>phya translation
QAARFLFMKNKVRMIVDCHAKHVKVLQDEKLPFDLTLCGSTLRAPHSCHLQYMANMDSIA
SLVMAVVVNDSDEDGDSRDAVLPQKKKRLWGLVVCHNTTPRFV
```

28.2.9 Bio.bgzf module

Read and write BGZF compressed files (the GZIP variant used in BAM).

The SAM/BAM file format (Sequence Alignment/Map) comes in a plain text format (SAM), and a compressed binary format (BAM). The latter uses a modified form of gzip compression called BGZF (Blocked GNU Zip Format), which can be applied to any file format to provide compression with efficient random access. BGZF is described together with the SAM/BAM file format at https://samtools.sourceforge.net/SAM1.pdf

Please read the text below about 'virtual offsets' before using BGZF files for random access.

Aim of this module

The Python gzip library can be used to read BGZF files, since for decompression they are just (specialised) gzip files. What this module aims to facilitate is random access to BGZF files (using the 'virtual offset' idea), and writing BGZF files (which means using suitably sized gzip blocks and writing the extra 'BC' field in the gzip headers). As in the gzip library, the zlib library is used internally.

In addition to being required for random access to and writing of BAM files, the BGZF format can also be used on other sequential data (in the sense of one record after another), such as most of the sequence data formats supported in Bio.SeqIO (like FASTA, FASTQ, GenBank, etc) or large MAF alignments.

The Bio.SeqIO indexing functions use this module to support BGZF files.

Technical Introduction to BGZF

The gzip file format allows multiple compressed blocks, each of which could be a stand alone gzip file. As an interesting bonus, this means you can use Unix `cat` to combine two or more gzip files into one by concatenating them. Also, each block can have one of several compression levels (including uncompressed, which actually takes up a little bit more space due to the gzip header).

What the BAM designers realised was that while random access to data stored in traditional gzip files was slow, breaking the file into gzip blocks would allow fast random access to each block. To access a particular piece of the decompressed data, you just need to know which block it starts in (the offset of the gzip block start), and how far into the (decompressed) contents of the block you need to read.

One problem with this is finding the gzip block sizes efficiently. You can do it with a standard gzip file, but it requires every block to be decompressed – and that would be rather slow. Additionally typical gzip files may use very large blocks.

All that differs in BGZF is that compressed size of each gzip block is limited to 2^16 bytes, and an extra 'BC' field in the gzip header records this size. Traditional decompression tools can ignore this, and unzip the file just like any other gzip file.

The point of this is you can look at the first BGZF block, find out how big it is from this 'BC' header, and thus seek immediately to the second block, and so on.

The BAM indexing scheme records read positions using a 64 bit 'virtual offset', comprising `coffset << 16 | uoffset`, where `coffset` is the file offset of the BGZF block containing the start of the read (unsigned integer using up to 64-16 = 48 bits), and `uoffset` is the offset within the (decompressed) block (unsigned 16 bit integer).

This limits you to BAM files where the last block starts by 2^48 bytes, or 256 petabytes, and the decompressed size of each block is at most 2^16 bytes, or 64kb. Note that this matches the BGZF 'BC' field size which limits the compressed size of each block to 2^16 bytes, allowing for BAM files to use BGZF with no gzip compression (useful for intermediate files in memory to reduce CPU load).

Warning about namespaces

It is considered a bad idea to use "from XXX import *" in Python, because it pollutes the namespace. This is a real issue with Bio.bgzf (and the standard Python library gzip) because they contain a function called open i.e. Suppose you do this:

```
>>> from Bio.bgzf import *
>>> print(open.__module__)
Bio.bgzf
```

Or,

```
>>> from gzip import *
>>> print(open.__module__)
gzip
```

Notice that the open function has been replaced. You can "fix" this if you need to by importing the built-in open function:

```
>>> from builtins import open
```

However, what we recommend instead is to use the explicit namespace, e.g.

```
>>> from Bio import bgzf
>>> print(bgzf.open.__module__)
Bio.bgzf
```

Examples

This is an ordinary GenBank file compressed using BGZF, so it can be decompressed using gzip,

```
>>> import gzip
>>> handle = gzip.open("GenBank/NC_000932.gb.bgz", "r")
>>> assert 0 == handle.tell()
>>> line = handle.readline()
>>> assert 80 == handle.tell()
>>> line = handle.readline()
>>> assert 143 == handle.tell()
```

(continues on next page)

(continued from previous page)

```
>>> data = handle.read(70000)
>>> assert 70143 == handle.tell()
>>> handle.close()
```

We can also access the file using the BGZF reader - but pay attention to the file offsets which will be explained below:

```
>>> handle = BgzfReader("GenBank/NC_000932.gb.bgz", "r")
>>> assert 0 == handle.tell()
>>> print(handle.readline().rstrip())
LOCUS       NC_000932             154478 bp    DNA     circular PLN 15-APR-2009
>>> assert 80 == handle.tell()
>>> print(handle.readline().rstrip())
DEFINITION  Arabidopsis thaliana chloroplast, complete genome.
>>> assert 143 == handle.tell()
>>> data = handle.read(70000)
>>> assert 987828735 == handle.tell()
>>> print(handle.readline().rstrip())
f="GeneID:844718"
>>> print(handle.readline().rstrip())
     CDS             complement(join(84337..84771,85454..85843))
>>> offset = handle.seek(make_virtual_offset(55074, 126))
>>> print(handle.readline().rstrip())
    68521 tatgtcattc gaaattgtat aaagacaact cctatttaat agagctattt gtgcaagtat
>>> handle.close()
```

Notice the handle's offset looks different as a BGZF file. This brings us to the key point about BGZF, which is the block structure:

```
>>> handle = open("GenBank/NC_000932.gb.bgz", "rb")
>>> for values in BgzfBlocks(handle):
...     print("Raw start %i, raw length %i; data start %i, data length %i" % values)
Raw start 0, raw length 15073; data start 0, data length 65536
Raw start 15073, raw length 17857; data start 65536, data length 65536
Raw start 32930, raw length 22144; data start 131072, data length 65536
Raw start 55074, raw length 22230; data start 196608, data length 65536
Raw start 77304, raw length 14939; data start 262144, data length 43478
Raw start 92243, raw length 28; data start 305622, data length 0
>>> handle.close()
```

In this example the first three blocks are 'full' and hold 65536 bytes of uncompressed data. The fourth block isn't full and holds 43478 bytes. Finally there is a special empty fifth block which takes 28 bytes on disk and serves as an 'end of file' (EOF) marker. If this is missing, it is possible your BGZF file is incomplete.

By reading ahead 70,000 bytes we moved into the second BGZF block, and at that point the BGZF virtual offsets start to look different to a simple offset into the decompressed data as exposed by the gzip library.

As an example, consider seeking to the decompressed position 196734. Since 196734 = 65536 + 65536 + 65536 + 126 = 65536*3 + 126, this is equivalent to jumping the first three blocks (which in this specific example are all size 65536 after decompression - which does not always hold) and starting at byte 126 of the fourth block (after decompression). For BGZF, we need to know the fourth block's offset of 55074 and the offset within the block of 126 to get the BGZF virtual offset.

```
>>> print(55074 << 16 | 126)
3609329790
```

(continues on next page)

(continued from previous page)

```
>>> print(bgzf.make_virtual_offset(55074, 126))
3609329790
```

Thus for this BGZF file, decompressed position 196734 corresponds to the virtual offset 3609329790. However, another BGZF file with different contents would have compressed more or less efficiently, so the compressed blocks would be different sizes. What this means is the mapping between the uncompressed offset and the compressed virtual offset depends on the BGZF file you are using.

If you are accessing a BGZF file via this module, just use the handle.tell() method to note the virtual offset of a position you may later want to return to using handle.seek().

The catch with BGZF virtual offsets is while they can be compared (which offset comes first in the file), you cannot safely subtract them to get the size of the data between them, nor add/subtract a relative offset.

Of course you can parse this file with Bio.SeqIO using BgzfReader, although there isn't any benefit over using gzip.open(...), unless you want to index BGZF compressed sequence files:

```
>>> from Bio import SeqIO
>>> handle = BgzfReader("GenBank/NC_000932.gb.bgz")
>>> record = SeqIO.read(handle, "genbank")
>>> handle.close()
>>> print(record.id)
NC_000932.1
```

Text Mode

Like the standard library gzip.open(...), the BGZF code defaults to opening files in binary mode.

You can request the file be opened in text mode, but beware that this is hard coded to the simple "latin1" (aka "iso-8859-1") encoding (which includes all the ASCII characters), which works well with most Western European languages. However, it is not fully compatible with the more widely used UTF-8 encoding.

In variable width encodings like UTF-8, some single characters in the unicode text output are represented by multiple bytes in the raw binary form. This is problematic with BGZF, as we cannot always decode each block in isolation - a single unicode character could be split over two blocks. This can even happen with fixed width unicode encodings, as the BGZF block size is not fixed.

Therefore, this module is currently restricted to only support single byte unicode encodings, such as ASCII, "latin1" (which is a superset of ASCII), or potentially other character maps (not implemented).

Furthermore, unlike the default text mode on Python 3, we do not attempt to implement universal new line mode. This transforms the various operating system new line conventions like Windows (CR LF or "rn"), Unix (just LF, "n"), or old Macs (just CR, "r"), into just LF ("n"). Here we have the same problem - is "r" at the end of a block an incomplete Windows style new line?

Instead, you will get the CR ("r") and LF ("n") characters as is.

If your data is in UTF-8 or any other incompatible encoding, you must use binary mode, and decode the appropriate fragments yourself.

Bio.bgzf.**open**(*filename*, *mode='rb'*)

Open a BGZF file for reading, writing or appending.

If text mode is requested, in order to avoid multi-byte characters, this is hard coded to use the "latin1" encoding, and "r" and "n" are passed as is (without implementing universal new line mode).

If your data is in UTF-8 or any other incompatible encoding, you must use binary mode, and decode the appropriate fragments yourself.

Bio.bgzf.**make_virtual_offset**(*block_start_offset*, *within_block_offset*)

Compute a BGZF virtual offset from block start and within block offsets.

The BAM indexing scheme records read positions using a 64 bit 'virtual offset', comprising in C terms:

block_start_offset << 16 | within_block_offset

Here block_start_offset is the file offset of the BGZF block start (unsigned integer using up to 64-16 = 48 bits), and within_block_offset within the (decompressed) block (unsigned 16 bit integer).

```
>>> make_virtual_offset(0, 0)
0
>>> make_virtual_offset(0, 1)
1
>>> make_virtual_offset(0, 2**16 - 1)
65535
>>> make_virtual_offset(0, 2**16)
Traceback (most recent call last):
...
ValueError: Require 0 <= within_block_offset < 2**16, got 65536
```

```
>>> 65536 == make_virtual_offset(1, 0)
True
>>> 65537 == make_virtual_offset(1, 1)
True
>>> 131071 == make_virtual_offset(1, 2**16 - 1)
True
```

```
>>> 6553600000 == make_virtual_offset(100000, 0)
True
>>> 6553600001 == make_virtual_offset(100000, 1)
True
>>> 6553600010 == make_virtual_offset(100000, 10)
True
```

```
>>> make_virtual_offset(2**48, 0)
Traceback (most recent call last):
...
ValueError: Require 0 <= block_start_offset < 2**48, got 281474976710656
```

Bio.bgzf.**split_virtual_offset**(*virtual_offset*)

Divides a 64-bit BGZF virtual offset into block start & within block offsets.

```
>>> (100000, 0) == split_virtual_offset(6553600000)
True
>>> (100000, 10) == split_virtual_offset(6553600010)
True
```

Bio.bgzf.**BgzfBlocks**(*handle*)

Low level debugging function to inspect BGZF blocks.

Expects a BGZF compressed file opened in binary read mode using the builtin open function. Do not use a handle from this bgzf module or the gzip module's open function which will decompress the file.

Returns the block start offset (see virtual offsets), the block length (add these for the start of the next block), and

the decompressed length of the blocks contents (limited to 65536 in BGZF), as an iterator - one tuple per BGZF block.

```
>>> from builtins import open
>>> handle = open("SamBam/ex1.bam", "rb")
>>> for values in BgzfBlocks(handle):
...     print("Raw start %i, raw length %i; data start %i, data length %i" % values)
Raw start 0, raw length 18239; data start 0, data length 65536
Raw start 18239, raw length 18223; data start 65536, data length 65536
Raw start 36462, raw length 18017; data start 131072, data length 65536
Raw start 54479, raw length 17342; data start 196608, data length 65536
Raw start 71821, raw length 17715; data start 262144, data length 65536
Raw start 89536, raw length 17728; data start 327680, data length 65536
Raw start 107264, raw length 17292; data start 393216, data length 63398
Raw start 124556, raw length 28; data start 456614, data length 0
>>> handle.close()
```

Indirectly we can tell this file came from an old version of samtools because all the blocks (except the final one and the dummy empty EOF marker block) are 65536 bytes. Later versions avoid splitting a read between two blocks, and give the header its own block (useful to speed up replacing the header). You can see this in ex1_refresh.bam created using samtools 0.1.18:

samtools view -b ex1.bam > ex1_refresh.bam

```
>>> handle = open("SamBam/ex1_refresh.bam", "rb")
>>> for values in BgzfBlocks(handle):
...     print("Raw start %i, raw length %i; data start %i, data length %i" % values)
Raw start 0, raw length 53; data start 0, data length 38
Raw start 53, raw length 18195; data start 38, data length 65434
Raw start 18248, raw length 18190; data start 65472, data length 65409
Raw start 36438, raw length 18004; data start 130881, data length 65483
Raw start 54442, raw length 17353; data start 196364, data length 65519
Raw start 71795, raw length 17708; data start 261883, data length 65411
Raw start 89503, raw length 17709; data start 327294, data length 65466
Raw start 107212, raw length 17390; data start 392760, data length 63854
Raw start 124602, raw length 28; data start 456614, data length 0
>>> handle.close()
```

The above example has no embedded SAM header (thus the first block is very small at just 38 bytes of decompressed data), while the next example does (a larger block of 103 bytes). Notice that the rest of the blocks show the same sizes (they contain the same read data):

```
>>> handle = open("SamBam/ex1_header.bam", "rb")
>>> for values in BgzfBlocks(handle):
...     print("Raw start %i, raw length %i; data start %i, data length %i" % values)
Raw start 0, raw length 104; data start 0, data length 103
Raw start 104, raw length 18195; data start 103, data length 65434
Raw start 18299, raw length 18190; data start 65537, data length 65409
Raw start 36489, raw length 18004; data start 130946, data length 65483
Raw start 54493, raw length 17353; data start 196429, data length 65519
Raw start 71846, raw length 17708; data start 261948, data length 65411
Raw start 89554, raw length 17709; data start 327359, data length 65466
Raw start 107263, raw length 17390; data start 392825, data length 63854
Raw start 124653, raw length 28; data start 456679, data length 0
>>> handle.close()
```

class Bio.bgzf.**BgzfReader**(*filename=None*, *mode='r'*, *fileobj=None*, *max_cache=100*)

Bases: object

BGZF reader, acts like a read only handle but seek/tell differ.

Let's use the BgzfBlocks function to have a peek at the BGZF blocks in an example BAM file,

```
>>> from builtins import open
>>> handle = open("SamBam/ex1.bam", "rb")
>>> for values in BgzfBlocks(handle):
...     print("Raw start %i, raw length %i; data start %i, data length %i" % values)
Raw start 0, raw length 18239; data start 0, data length 65536
Raw start 18239, raw length 18223; data start 65536, data length 65536
Raw start 36462, raw length 18017; data start 131072, data length 65536
Raw start 54479, raw length 17342; data start 196608, data length 65536
Raw start 71821, raw length 17715; data start 262144, data length 65536
Raw start 89536, raw length 17728; data start 327680, data length 65536
Raw start 107264, raw length 17292; data start 393216, data length 63398
Raw start 124556, raw length 28; data start 456614, data length 0
>>> handle.close()
```

Now let's see how to use this block information to jump to specific parts of the decompressed BAM file:

```
>>> handle = BgzfReader("SamBam/ex1.bam", "rb")
>>> assert 0 == handle.tell()
>>> magic = handle.read(4)
>>> assert 4 == handle.tell()
```

So far nothing so strange, we got the magic marker used at the start of a decompressed BAM file, and the handle position makes sense. Now however, let's jump to the end of this block and 4 bytes into the next block by reading 65536 bytes,

```
>>> data = handle.read(65536)
>>> len(data)
65536
>>> assert 1195311108 == handle.tell()
```

Expecting 4 + 65536 = 65540 were you? Well this is a BGZF 64-bit virtual offset, which means:

```
>>> split_virtual_offset(1195311108)
(18239, 4)
```

You should spot 18239 as the start of the second BGZF block, while the 4 is the offset into this block. See also make_virtual_offset,

```
>>> make_virtual_offset(18239, 4)
1195311108
```

Let's jump back to almost the start of the file,

```
>>> make_virtual_offset(0, 2)
2
>>> handle.seek(2)
2
>>> handle.close()
```

Note that you can use the max_cache argument to limit the number of BGZF blocks cached in memory. The default is 100, and since each block can be up to 64kb, the default cache could take up to 6MB of RAM. The cache is not important for reading through the file in one pass, but is important for improving performance of random access.

__init__(*filename=None*, *mode='r'*, *fileobj=None*, *max_cache=100*)

Initialize the class for reading a BGZF file.

You would typically use the top level `bgzf.open(...)` function which will call this class internally. Direct use is discouraged.

Either the `filename` (string) or `fileobj` (input file object in binary mode) arguments must be supplied, but not both.

Argument `mode` controls if the data will be returned as strings in text mode ("rt", "tr", or default "r"), or bytes binary mode ("rb" or "br"). The argument name matches the built-in `open(...)` and standard library `gzip.open(...)` function.

If text mode is requested, in order to avoid multi-byte characters, this is hard coded to use the "latin1" encoding, and "r" and "n" are passed as is (without implementing universal new line mode). There is no `encoding` argument.

If your data is in UTF-8 or any other incompatible encoding, you must use binary mode, and decode the appropriate fragments yourself.

Argument `max_cache` controls the maximum number of BGZF blocks to cache in memory. Each can be up to 64kb thus the default of 100 blocks could take up to 6MB of RAM. This is important for efficient random access, a small value is fine for reading the file in one pass.

tell()

Return a 64-bit unsigned BGZF virtual offset.

seek(*virtual_offset*)

Seek to a 64-bit unsigned BGZF virtual offset.

read(*size=-1*)

Read method for the BGZF module.

readline()

Read a single line for the BGZF file.

__next__()

Return the next line.

__iter__()

Iterate over the lines in the BGZF file.

close()

Close BGZF file.

seekable()

Return True indicating the BGZF supports random access.

isatty()

Return True if connected to a TTY device.

fileno()

Return integer file descriptor.

__enter__()

Open a file operable with WITH statement.

__exit__(*type*, *value*, *traceback*)

Close a file with WITH statement.

class Bio.bgzf.**BgzfWriter**(*filename=None*, *mode='w'*, *fileobj=None*, *compresslevel=6*)

Bases: object

Define a BGZFWriter object.

__init__(*filename=None*, *mode='w'*, *fileobj=None*, *compresslevel=6*)

Initialize the class.

write(*data*)

Write method for the class.

flush()

Flush data explicitally.

close()

Flush data, write 28 bytes BGZF EOF marker, and close BGZF file.

samtools will look for a magic EOF marker, just a 28 byte empty BGZF block, and if it is missing warns the BAM file may be truncated. In addition to samtools writing this block, so too does bgzip - so this implementation does too.

tell()

Return a BGZF 64-bit virtual offset.

seekable()

Return True indicating the BGZF supports random access.

isatty()

Return True if connected to a TTY device.

fileno()

Return integer file descriptor.

__enter__()

Open a file operable with WITH statement.

__exit__(*type*, *value*, *traceback*)

Close a file with WITH statement.

28.2.10 Bio.cpairwise2 module

Optimized C routines that complement pairwise2.py. These are called from within pairwise2.py.

Bio.cpairwise2.**rint**()

28.2.11 Bio.kNN module

Code for doing k-nearest-neighbors classification (DEPRECATED).

k Nearest Neighbors is a supervised learning algorithm that classifies a new observation based the classes in its surrounding neighborhood.

Glossary:

- distance The distance between two points in the feature space.
- weight The importance given to each point for classification.

Classes:

- kNN Holds information for a nearest neighbors classifier.

Functions:

- train Train a new kNN classifier.
- calculate Calculate the probabilities of each class, given an observation.
- classify Classify an observation into a class.

Weighting Functions:

- equal_weight Every example is given a weight of 1.

This module has been deprecated, please consider an alternative like scikit-learn instead.

`class Bio.kNN.kNN`

Bases: `object`

Holds information necessary to do nearest neighbors classification.

Attributes:

- classes Set of the possible classes.
- xs List of the neighbors.
- ys List of the classes that the neighbors belong to.
- k Number of neighbors to look at.

`__init__()`

Initialize the class.

`Bio.kNN.equal_weight(x, y)`

Return integer one (dummy method for equally weighting).

`Bio.kNN.train(xs, ys, k, typecode=None)`

Train a k nearest neighbors classifier on a training set.

xs is a list of observations and ys is a list of the class assignments. Thus, xs and ys should contain the same number of elements. k is the number of neighbors that should be examined when doing the classification.

`Bio.kNN.calculate(knn, x, weight_fn=None, distance_fn=None)`

Calculate the probability for each class.

Arguments:

- x is the observed data.
- weight_fn is an optional function that takes x and a training example, and returns a weight.

- distance_fn is an optional function that takes two points and returns the distance between them. If distance_fn is None (the default), the Euclidean distance is used.

Returns a dictionary of the class to the weight given to the class.

Bio.kNN.**classify**(*knn*, *x*, *weight_fn=None*, *distance_fn=None*)

Classify an observation into a class.

If not specified, weight_fn will give all neighbors equal weight. distance_fn is an optional function that takes two points and returns the distance between them. If distance_fn is None (the default), the Euclidean distance is used.

28.2.12 Bio.pairwise2 module

Pairwise sequence alignment using a dynamic programming algorithm.

This provides functions to get global and local alignments between two sequences. A global alignment finds the best concordance between all characters in two sequences. A local alignment finds just the subsequences that align the best. Local alignments must have a positive score to be reported and they will not be extended for 'zero counting' matches. This means a local alignment will always start and end with a positive counting match.

When doing alignments, you can specify the match score and gap penalties. The match score indicates the compatibility between an alignment of two characters in the sequences. Highly compatible characters should be given positive scores, and incompatible ones should be given negative scores or 0. The gap penalties should be negative.

The names of the alignment functions in this module follow the convention <alignment type>XX where <alignment type> is either "global" or "local" and XX is a 2 character code indicating the parameters it takes. The first character indicates the parameters for matches (and mismatches), and the second indicates the parameters for gap penalties.

The match parameters are:

```
CODE  DESCRIPTION & OPTIONAL KEYWORDS
x     No parameters. Identical characters have score of 1, otherwise 0.
m     A match score is the score of identical chars, otherwise mismatch
      score. Keywords ``match``, ``mismatch``.
d     A dictionary returns the score of any pair of characters.
      Keyword ``match_dict``.
c     A callback function returns scores. Keyword ``match_fn``.
```

The gap penalty parameters are:

```
CODE  DESCRIPTION & OPTIONAL KEYWORDS
x     No gap penalties.
s     Same open and extend gap penalties for both sequences.
      Keywords ``open``, ``extend``.
d     The sequences have different open and extend gap penalties.
      Keywords ``openA``, ``extendA``, ``openB``, ``extendB``.
c     A callback function returns the gap penalties.
      Keywords ``gap_A_fn``, ``gap_B_fn``.
```

All the different alignment functions are contained in an object `align`. For example:

```
>>> from Bio import pairwise2
>>> alignments = pairwise2.align.globalxx("ACCGT", "ACG")
```

For better readability, the required arguments can be used with optional keywords:

```
>>> alignments = pairwise2.align.globalxx(sequenceA="ACCGT", sequenceB="ACG")
```

The result is a list of the alignments between the two strings. Each alignment is a named tuple consisting of the two aligned sequences, the score and the start and end positions of the alignment:

```
>>> print(alignments)
[Alignment(seqA='ACCGT', seqB='A-CG-', score=3.0, start=0, end=5), ...
```

You can access each element of an alignment by index or name:

```
>>> alignments[0][2]
3.0
>>> alignments[0].score
3.0
```

For a nice printout of an alignment, use the `format_alignment` method of the module:

```
>>> from Bio.pairwise2 import format_alignment
>>> print(format_alignment(*alignments[0]))
ACCGT
| ||
A-CG-
  Score=3
```

All alignment functions have the following arguments:

- Two sequences: strings, Biopython sequence objects or lists. Lists are useful for supplying sequences which contain residues that are encoded by more than one letter.
- `penalize_extend_when_opening`: boolean (default: False). Whether to count an extension penalty when opening a gap. If false, a gap of 1 is only penalized an "open" penalty, otherwise it is penalized "open+extend".
- `penalize_end_gaps`: boolean. Whether to count the gaps at the ends of an alignment. By default, they are counted for global alignments but not for local ones. Setting `penalize_end_gaps` to (boolean, boolean) allows you to specify for the two sequences separately whether gaps at the end of the alignment should be counted.
- `gap_char`: string (default: `'-'`). Which character to use as a gap character in the alignment returned. If your input sequences are lists, you must change this to `['-']`.
- `force_generic`: boolean (default: False). Always use the generic, non-cached, dynamic programming function (slow!). For debugging.
- `score_only`: boolean (default: False). Only get the best score, don't recover any alignments. The return value of the function is the score. Faster and uses less memory.
- `one_alignment_only`: boolean (default: False). Only recover one alignment.

The other parameters of the alignment function depend on the function called. Some examples:

- Find the best global alignment between the two sequences. Identical characters are given 1 point. No points are deducted for mismatches or gaps.

```
>>> for a in pairwise2.align.globalxx("ACCGT", "ACG"):
...     print(format_alignment(*a))
ACCGT
| ||
A-CG-
  Score=3
```

(continues on next page)

(continued from previous page)

```
ACCGT
|| |
AC-G-
  Score=3
```

- Same thing as before, but with a local alignment. Note that `format_alignment` will only show the aligned parts of the sequences, together with the starting positions.

```
>>> for a in pairwise2.align.localxx("ACCGT", "ACG"):
...     print(format_alignment(*a))
1 ACCG
  | ||
1 A-CG
  Score=3

1 ACCG
  || |
1 AC-G
  Score=3
```

To restore the 'historic' behaviour of `format_alignemt`, i.e., showing also the un-aligned parts of both sequences, use the new keyword parameter `full_sequences`:

```
>>> for a in pairwise2.align.localxx("ACCGT", "ACG"):
...     print(format_alignment(*a, full_sequences=True))
ACCGT
| ||
A-CG-
  Score=3

ACCGT
|| |
AC-G-
  Score=3
```

- Do a global alignment. Identical characters are given 2 points, 1 point is deducted for each non-identical character. Don't penalize gaps.

```
>>> for a in pairwise2.align.globalmx("ACCGT", "ACG", 2, -1):
...     print(format_alignment(*a))
ACCGT
| ||
A-CG-
  Score=6

ACCGT
|| |
AC-G-
  Score=6
```

- Same as above, except now 0.5 points are deducted when opening a gap, and 0.1 points are deducted when extending it.

```
>>> for a in pairwise2.align.globalms("ACCGT", "ACG", 2, -1, -.5, -.1):
...     print(format_alignment(*a))
ACCGT
| ||
A-CG-
  Score=5

ACCGT
|| |
AC-G-
  Score=5
```

- Note that you can use keywords to increase the readability, e.g.:

```
>>> a = pairwise2.align.globalms("ACGT", "ACG", match=2, mismatch=-1, open=-.5,
...                              extend=-.1)
```

- Depending on the penalties, a gap in one sequence may be followed by a gap in the other sequence.If you don't like this behaviour, increase the gap-open penalty:

```
>>> for a in pairwise2.align.globalms("A", "T", 5, -4, -1, -.1):
...     print(format_alignment(*a))
A-

-T
  Score=-2

>>> for a in pairwise2.align.globalms("A", "T", 5, -4, -3, -.1):
...     print(format_alignment(*a))
A
.
T
  Score=-4
```

- The alignment function can also use known matrices already included in Biopython (in `Bio.Align.substitution_matrices`):

```
>>> from Bio.Align import substitution_matrices
>>> matrix = substitution_matrices.load("BLOSUM62")
>>> for a in pairwise2.align.globaldx("KEVLA", "EVL", matrix):
...     print(format_alignment(*a))
KEVLA
 |||
-EVL-
  Score=13
```

- With the parameter `c` you can define your own match- and gap functions. E.g. to define an affine logarithmic gap function and using it:

```
>>> from math import log
>>> def gap_function(x, y):  # x is gap position in seq, y is gap length
...     if y == 0:  # No gap
...         return 0
```

(continues on next page)

(continued from previous page)

```
...     elif y == 1:  # Gap open penalty
...         return -2
...     return - (2 + y/4.0 + log(y)/2.0)
...
>>> alignment = pairwise2.align.globalmc("ACCCCCGT", "ACG", 5, -4,
...                                      gap_function, gap_function)
```

You can define different gap functions for each sequence. Self-defined match functions must take the two residues to be compared and return a score.

To see a description of the parameters for a function, please look at the docstring for the function via the help function, e.g. type `help(pairwise2.align.localds)` at the Python prompt.

class Bio.pairwise2.**Alignment**(*seqA*, *seqB*, *score*, *start*, *end*)

Bases: `tuple`

__getnewargs__()

Return self as a plain tuple. Used by copy and pickle.

__match_args__ = ('seqA', 'seqB', 'score', 'start', 'end')

static __new__(*_cls*, *seqA*, *seqB*, *score*, *start*, *end*)

Create new instance of Alignment(seqA, seqB, score, start, end)

__repr__()

Return a nicely formatted representation string

__slots__ = ()

end

Alias for field number 4

score

Alias for field number 2

seqA

Alias for field number 0

seqB

Alias for field number 1

start

Alias for field number 3

class Bio.pairwise2.**identity_match**(*match=1*, *mismatch=0*)

Bases: `object`

Create a match function for use in an alignment.

match and mismatch are the scores to give when two residues are equal or unequal. By default, match is 1 and mismatch is 0.

__init__(*match=1*, *mismatch=0*)

Initialize the class.

__call__(*charA*, *charB*)

Call a match function instance already created.

class Bio.pairwise2.**dictionary_match**(*score_dict*, *symmetric=1*)

Bases: object

Create a match function for use in an alignment.

Attributes:

- score_dict - A dictionary where the keys are tuples (residue 1, residue 2) and the values are the match scores between those residues.
- symmetric - A flag that indicates whether the scores are symmetric.

__init__(*score_dict*, *symmetric=1*)

Initialize the class.

__call__(*charA*, *charB*)

Call a dictionary match instance already created.

class Bio.pairwise2.**affine_penalty**(*open*, *extend*, *penalize_extend_when_opening=0*)

Bases: object

Create a gap function for use in an alignment.

__init__(*open*, *extend*, *penalize_extend_when_opening=0*)

Initialize the class.

__call__(*index*, *length*)

Call a gap function instance already created.

Bio.pairwise2.**calc_affine_penalty**(*length*, *open*, *extend*, *penalize_extend_when_opening*)

Calculate a penalty score for the gap function.

Bio.pairwise2.**print_matrix**(*matrix*)

Print out a matrix for debugging purposes.

Bio.pairwise2.**format_alignment**(*align1*, *align2*, *score*, *begin*, *end*, *full_sequences=False*)

Format the alignment prettily into a string.

IMPORTANT: Gap symbol must be "-" (or ['-'] for lists)!

Since Biopython 1.71 identical matches are shown with a pipe character, mismatches as a dot, and gaps as a space.

Prior releases just used the pipe character to indicate the aligned region (matches, mismatches and gaps).

Also, in local alignments, if the alignment does not include the whole sequences, now only the aligned part is shown, together with the start positions of the aligned subsequences. The start positions are 1-based; so start position n is the n-th base/amino acid in the *un-aligned* sequence.

NOTE: This is different to the alignment's begin/end values, which give the Python indices (0-based) of the bases/amino acids in the *aligned* sequences.

If you want to restore the 'historic' behaviour, that means displaying the whole sequences (including the non-aligned parts), use `full_sequences=True`. In this case, the non-aligned leading and trailing parts are also indicated by spaces in the match-line.

28.3 Module contents

Collection of modules for dealing with biological data in Python.

The Biopython Project is an international association of developers of freely available Python tools for computational molecular biology.

https://biopython.org

exception Bio.**MissingExternalDependencyError**

Bases: Exception

Missing an external dependency.

Used for things like missing command line tools. Important for our unit tests to allow skipping tests with missing external dependencies.

exception Bio.**MissingPythonDependencyError**

Bases: *MissingExternalDependencyError*, ImportError

Missing an external python dependency (subclass of ImportError).

Used for missing Python modules (rather than just a typical ImportError). Important for our unit tests to allow skipping tests with missing external python dependencies, while also allowing the exception to be caught as an ImportError.

__annotations__ = {}

exception Bio.**StreamModeError**

Bases: ValueError

Incorrect stream mode (text vs binary).

This error should be raised when a stream (file or file-like object) argument is in text mode while the receiving function expects binary mode, or vice versa.

exception Bio.**BiopythonWarning**

Bases: Warning

Biopython warning.

Biopython should use this warning (or subclasses of it), making it easy to silence all our warning messages should you wish to:

```
>>> import warnings
>>> from Bio import BiopythonWarning
>>> warnings.simplefilter('ignore', BiopythonWarning)
```

Consult the warnings module documentation for more details.

exception Bio.**BiopythonParserWarning**

Bases: *BiopythonWarning*

Biopython parser warning.

Some in-valid data files cannot be parsed and will trigger an exception. Where a reasonable interpretation is possible, Biopython will issue this warning to indicate a potential problem. To silence these warnings, use:

```
>>> import warnings
>>> from Bio import BiopythonParserWarning
>>> warnings.simplefilter('ignore', BiopythonParserWarning)
```

Consult the warnings module documentation for more details.

__annotations__ = {}

exception Bio.BiopythonDeprecationWarning

Bases: *BiopythonWarning*

Biopython deprecation warning.

Biopython uses this warning instead of the built in DeprecationWarning since those are ignored by default since Python 2.7.

To silence all our deprecation warning messages, use:

```
>>> import warnings
>>> from Bio import BiopythonDeprecationWarning
>>> warnings.simplefilter('ignore', BiopythonDeprecationWarning)
```

Code marked as deprecated is likely to be removed in a future version of Biopython. To avoid removal of this code, please contact the Biopython developers via the mailing list or GitHub.

__annotations__ = {}

exception Bio.BiopythonExperimentalWarning

Bases: *BiopythonWarning*

Biopython experimental code warning.

Biopython uses this warning for experimental code ('alpha' or 'beta' level code) which is released as part of the standard releases to mark sub-modules or functions for early adopters to test & give feedback.

Code issuing this warning is likely to change (or even be removed) in a subsequent release of Biopython. Such code should NOT be used for production/stable code. It should only be used if:

- You are running the latest release of Biopython, or ideally the latest code from our repository.
- You are subscribed to the biopython-dev mailing list to provide feedback on this code, and to be alerted of changes to it.

If all goes well, experimental code would be promoted to stable in a subsequent release, and this warning removed from it.

__annotations__ = {}

CHAPTER

TWENTYNINE

BIOSQL PACKAGE

29.1 Submodules

29.1.1 BioSQL.BioSeq module

Implementations of Biopython-like Seq objects on top of BioSQL.

This allows retrieval of items stored in a BioSQL database using a biopython-like SeqRecord and Seq interface.

Note: Currently we do not support recording per-letter-annotations (like quality scores) in BioSQL.

class BioSQL.BioSeq.**DBSeqRecord**(*adaptor*, *primary_id*)

Bases: *SeqRecord*

BioSQL equivalent of the Biopython SeqRecord object.

__init__(*adaptor*, *primary_id*)

Create a DBSeqRecord object.

Arguments:

- adaptor - A BioSQL.BioSeqDatabase.Adaptor object
- primary_id - An internal integer ID used by BioSQL

You wouldn't normally create a DBSeqRecord object yourself, this is done for you when using a BioSeqDatabase object

property seq

Seq object

property dbxrefs: list[str]

Database cross references.

__annotations__ = {}

property features

Features

property annotations: dict[str, str | int]

Annotations.

29.1.2 BioSQL.BioSeqDatabase module

Connect with a BioSQL database and load Biopython like objects from it.

This provides interfaces for loading biological objects from a relational database, and is compatible with the BioSQL standards.

`BioSQL.BioSeqDatabase.open_database(driver='MySQLdb', **kwargs)`

Load an existing BioSQL-style database.

This function is the easiest way to retrieve a connection to a database, doing something like:

```
from BioSQL import BioSeqDatabase
server = BioSeqDatabase.open_database(user="root", db="minidb")
```

Arguments:

- driver - The name of the database driver to use for connecting. The driver should implement the python DB API. By default, the MySQLdb driver is used.
- user -the username to connect to the database with.
- password, passwd - the password to connect with
- host - the hostname of the database
- database or db - the name of the database

`class BioSQL.BioSeqDatabase.DBServer(conn, module, module_name=None)`

Bases: `object`

Represents a BioSQL database containing namespaces (sub-databases).

This acts like a Python dictionary, giving access to each namespace (defined by a row in the biodatabase table) as a BioSeqDatabase object.

`__init__(conn, module, module_name=None)`

Create a DBServer object.

Arguments:

- conn - A database connection object
- module - The module used to create the database connection
- module_name - Optionally, the name of the module. Default: module.__name__

Normally you would not want to create a DBServer object yourself. Instead use the open_database function, which returns an instance of DBServer.

`__repr__()`

Return a short description of the class name and database connection.

`__getitem__(name)`

Return a BioSeqDatabase object.

Arguments:

- name - The name of the BioSeqDatabase

`__len__()`

Return number of namespaces (sub-databases) in this database.

__contains__(*value*)

Check if a namespace (sub-database) in this database.

__iter__()

Iterate over namespaces (sub-databases) in the database.

keys()

Iterate over namespaces (sub-databases) in the database.

values()

Iterate over BioSeqDatabase objects in the database.

items()

Iterate over (namespace, BioSeqDatabase) in the database.

__delitem__(*name*)

Remove a namespace and all its entries.

new_database(*db_name*, *authority=None*, *description=None*)

Add a new database to the server and return it.

load_database_sql(*sql_file*)

Load a database schema into the given database.

This is used to create tables, etc when a database is first created. sql_file should specify the complete path to a file containing SQL entries for building the tables.

commit()

Commit the current transaction to the database.

rollback()

Roll-back the current transaction.

close()

Close the connection. No further activity possible.

class BioSQL.BioSeqDatabase.**Adaptor**(*conn*, *dbutils*, *wrap_cursor=False*)

Bases: object

High level wrapper for a database connection and cursor.

Most database calls in BioSQL are done indirectly though this adaptor class. This provides helper methods for fetching data and executing sql.

__init__(*conn*, *dbutils*, *wrap_cursor=False*)

Create an Adaptor object.

Arguments:

- conn - A database connection
- dbutils - A BioSQL.DBUtils object
- wrap_cursor - Optional, whether to wrap the cursor object

last_id(*table*)

Return the last row id for the selected table.

autocommit(*y=True*)

Set the autocommit mode. True values enable; False value disable.

commit()

Commit the current transaction.

rollback()

Roll-back the current transaction.

close()

Close the connection. No further activity possible.

fetch_dbid_by_dbname(*dbname*)

Return the internal id for the sub-database using its name.

fetch_seqid_by_display_id(*dbid*, *name*)

Return the internal id for a sequence using its display id.

Arguments:

- dbid - the internal id for the sub-database
- name - the name of the sequence. Corresponds to the name column of the bioentry table of the SQL schema

fetch_seqid_by_accession(*dbid*, *name*)

Return the internal id for a sequence using its accession.

Arguments:

- dbid - the internal id for the sub-database
- name - the accession of the sequence. Corresponds to the accession column of the bioentry table of the SQL schema

fetch_seqids_by_accession(*dbid*, *name*)

Return a list internal ids using an accession.

Arguments:

- dbid - the internal id for the sub-database
- name - the accession of the sequence. Corresponds to the accession column of the bioentry table of the SQL schema

fetch_seqid_by_version(*dbid*, *name*)

Return the internal id for a sequence using its accession and version.

Arguments:

- dbid - the internal id for the sub-database
- name - the accession of the sequence containing a version number. Must correspond to <accession>.<version>

fetch_seqid_by_identifier(*dbid*, *identifier*)

Return the internal id for a sequence using its identifier.

Arguments:

- dbid - the internal id for the sub-database
- identifier - the identifier of the sequence. Corresponds to the identifier column of the bioentry table in the SQL schema.

list_biodatabase_names()

Return a list of all of the sub-databases.

list_bioentry_ids(*dbid*)

Return a list of internal ids for all of the sequences in a sub-databae.

Arguments:

- dbid - The internal id for a sub-database

list_bioentry_display_ids(*dbid*)

Return a list of all sequence names in a sub-databae.

Arguments:

- dbid - The internal id for a sub-database

list_any_ids(*sql*, *args*)

Return ids given a SQL statement to select for them.

This assumes that the given SQL does a SELECT statement that returns a list of items. This parses them out of the 2D list they come as and just returns them in a list.

execute_one(*sql*, *args=None*)

Execute sql that returns 1 record, and return the record.

execute(*sql*, *args=None*)

Just execute an sql command.

executemany(*sql*, *args*)

Execute many sql commands.

get_subseq_as_string(*seqid*, *start*, *end*)

Return a substring of a sequence.

Arguments:

- seqid - The internal id for the sequence
- start - The start position of the sequence; 0-indexed
- end - The end position of the sequence

execute_and_fetch_col0(*sql*, *args=None*)

Return a list of values from the first column in the row.

execute_and_fetchall(*sql*, *args=None*)

Return a list of tuples of all rows.

class BioSQL.BioSeqDatabase.**MysqlConnectorAdaptor**(*conn*, *dbutils*, *wrap_cursor=False*)

Bases: *Adaptor*

A BioSQL Adaptor class with fixes for the MySQL interface.

BioSQL was failing due to returns of bytearray objects from the mysql-connector-python database connector. This adaptor class scrubs returns of bytearrays and of byte strings converting them to string objects instead. This adaptor class was made in response to backwards incompatible changes added to mysql-connector-python in release 2.0.0 of the package.

execute_one(*sql*, *args=None*)

Execute sql that returns 1 record, and return the record.

execute_and_fetch_col0(*sql*, *args=None*)

Return a list of values from the first column in the row.

execute_and_fetchall(*sql*, *args=None*)

Return a list of tuples of all rows.

__annotations__ = {}

class BioSQL.BioSeqDatabase.BioSeqDatabase(*adaptor*, *name*)

Bases: object

Represents a namespace (sub-database) within the BioSQL database.

i.e. One row in the biodatabase table, and all all rows in the bioentry table associated with it.

__init__(*adaptor*, *name*)

Create a BioDatabase object.

Arguments:

- adaptor - A BioSQL.Adaptor object
- name - The name of the sub-database (namespace)

__repr__()

Return a short summary of the BioSeqDatabase.

get_Seq_by_id(*name*)

Get a DBSeqRecord object by its name.

Example: seq_rec = db.get_Seq_by_id('ROA1_HUMAN')

The name of this method is misleading since it returns a DBSeqRecord rather than a Seq object, and presumably was to mirror BioPerl.

get_Seq_by_acc(*name*)

Get a DBSeqRecord object by accession number.

Example: seq_rec = db.get_Seq_by_acc('X77802')

The name of this method is misleading since it returns a DBSeqRecord rather than a Seq object, and presumably was to mirror BioPerl.

get_Seq_by_ver(*name*)

Get a DBSeqRecord object by version number.

Example: seq_rec = db.get_Seq_by_ver('X77802.1')

The name of this method is misleading since it returns a DBSeqRecord rather than a Seq object, and presumably was to mirror BioPerl.

get_Seqs_by_acc(*name*)

Get a list of DBSeqRecord objects by accession number.

Example: seq_recs = db.get_Seq_by_acc('X77802')

The name of this method is misleading since it returns a list of DBSeqRecord objects rather than a list of Seq objects, and presumably was to mirror BioPerl.

__getitem__(*key*)

Return a DBSeqRecord for one of the sequences in the sub-database.

Arguments:

- key - The internal id for the sequence

__delitem__(*key*)

Remove an entry and all its annotation.

__len__()

Return number of records in this namespace (sub database).

__contains__(*value*)

Check if a primary (internal) id is this namespace (sub database).

__iter__()

Iterate over ids (which may not be meaningful outside this database).

keys()

Iterate over ids (which may not be meaningful outside this database).

values()

Iterate over DBSeqRecord objects in the namespace (sub database).

items()

Iterate over (id, DBSeqRecord) for the namespace (sub database).

lookup(***kwargs*)

Return a DBSeqRecord using an acceptable identifier.

Arguments:

- kwargs - A single key-value pair where the key is one of primary_id, gi, display_id, name, accession, version

load(*record_iterator*, *fetch_NCBI_taxonomy=False*)

Load a set of SeqRecords into the BioSQL database.

record_iterator is either a list of SeqRecord objects, or an Iterator object that returns SeqRecord objects (such as the output from the Bio.SeqIO.parse() function), which will be used to populate the database.

fetch_NCBI_taxonomy is boolean flag allowing or preventing connection to the taxonomic database on the NCBI server (via Bio.Entrez) to fetch a detailed taxonomy for each SeqRecord.

Example:

```
from Bio import SeqIO
count = db.load(SeqIO.parse(open(filename), format))
```

Returns the number of records loaded.

29.1.3 BioSQL.DBUtils module

Helper code for Biopython's BioSQL code (for internal use).

class BioSQL.DBUtils.**Generic_dbutils**

Bases: object

Default database utilities.

__init__()

Create a Generic_dbutils object.

tname(*table*)
Return the name of the table.

last_id(*cursor*, *table*)
Return the last used id for a table.

execute(*cursor*, *sql*, *args=None*)
Just execute an sql command.

executemany(*cursor*, *sql*, *seq*)
Execute many sql commands.

autocommit(*conn*, *y=1*)
Set autocommit on the database connection.

class BioSQL.DBUtils.**Sqlite_dbutils**
Bases: *Generic_dbutils*

Custom database utilities for SQLite.

execute(*cursor*, *sql*, *args=None*)
Execute SQL command.

Replaces %s with ? for variable substitution in sqlite3.

executemany(*cursor*, *sql*, *seq*)
Execute many sql statements.

__annotations__ = {}

class BioSQL.DBUtils.**Mysql_dbutils**
Bases: *Generic_dbutils*

Custom database utilities for MySQL.

last_id(*cursor*, *table*)
Return the last used id for a table.

__annotations__ = {}

class BioSQL.DBUtils.**Psycopg2_dbutils**
Bases: _PostgreSQL_dbutils

Custom database utilities for Psycopg2 (PostgreSQL).

autocommit(*conn*, *y=True*)
Set autocommit on the database connection.

__annotations__ = {}

class BioSQL.DBUtils.**Pgdb_dbutils**
Bases: _PostgreSQL_dbutils

Custom database utilities for Pgdb (aka PyGreSQL, for PostgreSQL).

autocommit(*conn*, *y=True*)
Set autocommit on the database connection. Currently not implemented.

__annotations__ = {}

BioSQL.DBUtils.**get_dbutils**(*module_name*)
Return the correct dbutils object for the database driver.

29.1.4 BioSQL.Loader module

Load biopython objects into a BioSQL database for persistent storage.

This code makes it possible to store biopython objects in a relational database and then retrieve them back. You shouldn't use any of the classes in this module directly. Rather, call the load() method on a database object.

class BioSQL.Loader.**DatabaseLoader**(*adaptor*, *dbid*, *fetch_NCBI_taxonomy=False*)

Bases: object

Object used to load SeqRecord objects into a BioSQL database.

__init__(*adaptor*, *dbid*, *fetch_NCBI_taxonomy=False*)

Initialize with connection information for the database.

Creating a DatabaseLoader object is normally handled via the BioSeqDatabase DBServer object, for example:

```
from BioSQL import BioSeqDatabase
server = BioSeqDatabase.open_database(driver="MySQLdb",
                                      user="gbrowse",
                                      passwd="biosql",
                                      host="localhost",
                                      db="test_biosql")
try:
    db = server["test"]
except KeyError:
    db = server.new_database("test",
    description="For testing GBrowse")
```

load_seqrecord(*record*)

Load a Biopython SeqRecord into the database.

class BioSQL.Loader.**DatabaseRemover**(*adaptor*, *dbid*)

Bases: object

Complement the Loader functionality by fully removing a database.

This probably isn't really useful for normal purposes, since you can just do a:

```
DROP DATABASE db_name
```

and then recreate the database. But, it's really useful for testing purposes.

__init__(*adaptor*, *dbid*)

Initialize with a database id and adaptor connection.

remove()

Remove everything related to the given database id.

29.2 Module contents

Storing and retrieve biological sequences in a BioSQL relational database.

See:

- http://biopython.org/wiki/BioSQL
- http://www.biosql.org/

BIBLIOGRAPHY

[Altschul1990] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, David J. Lipman: Basic Local Alignment Search Tool. *Journal of Molecular Biology* **215** (3): 403–410 (1990). https://doi.org/10.1016/S0022-2836(05)80360-2

[Bailey1994] Timothy L. Bailey and Charles Elkan: Fitting a mixture model by expectation maximization to discover motifs in biopolymers, *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology* 28–36. AAAI Press, Menlo Park, California (1994).

[Cavener1987] Douglas R. Cavener: Comparison of the consensus sequence flanking translational start sites in Drosophila and vertebrates. *Nucleic Acids Research* **15** (4): 1353–1361 (1987). https://doi.org/10.1093/nar/15.4.1353

[Chapman2000] Brad Chapman and Jeff Chang: Biopython: Python tools for computational biology. *ACM SIGBIO Newsletter* **20** (2): 15–19 (August 2000).

[Cock2009] Peter J. A. Cock, Tiago Antao, Jeffrey T. Chang, Brad A. Chapman, Cymon J. Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczyński, Michiel J. L. de Hoon: Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics* **25** (11), 1422–1423 (2009). https://doi.org/10.1093/bioinformatics/btp163

[Cock2010] Peter J. A. Cock, Christopher J. Fields, Naohisa Goto, Michael L. Heuer, Peter M. Rice: The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research* **38** (6): 1767–1771 (2010). https://doi.org/10.1093/nar/gkp1137

[Cornish1985] Athel Cornish-Bowden: Nomenclature for incompletely specified bases in nucleic acid sequences: Recommendations 1984. *Nucleic Acids Research* **13** (9): 3021–3030 (1985). https://doi.org/10.1093/nar/13.9.3021

[Darling2004] Aaron E. Darling, Bob Mau, Frederick R. Blattner, Nicole T. Perna: Mauve: Multiple alignment of conserved genomic sequence with rearrangements. *Genome Research* **14** (7): 1394–1403 (2004). https://doi.org/10.1101/gr.2289704

[Dayhoff1978] M.O. Dayhoff, R.M. Schwartz, and B.C. Orcutt: A Model of Evolutionary Change in Proteins. *Atlas of Protein Sequence and Structure*, Volume 5, Supplement 3, 1978: 345–352. The National Biomedical Research Foundation, 1979.

[DeHoon2004] Michiel J. L. de Hoon, Seiya Imoto, John Nolan, Satoru Miyano: Open source clustering software. *Bioinformatics* **20** (9): 1453–1454 (2004). https://doi.org/10.1093/bioinformatics/bth078

[Durbin1998] Richard Durbin, Sean R. Eddy, Anders Krogh, Graeme Mitchison: Biological sequence analysis: Probabilistic models of proteins and nucleic acids. Cambridge University Press, Cambridge, UK (1998).

[Eisen1998] Michiel B. Eisen, Paul T. Spellman, Patrick O. Brown, David Botstein: Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences USA* **95** (25): 14863–14868 (1998). https://doi.org/10.1073/pnas.95.25.14863

[Goldman1994] Nick Goldman and Ziheng Yang: A codon-based model of nucleotide substitution for protein-coding DNA sequences. *Molecular Biology and Evolution* **11** (5) 725–736 (1994). https://doi.org/10.1093/oxfordjournals.molbev.a040153

[Golub1971] Gene H. Golub, Christian Reinsch: Singular value decomposition and least squares solutions. In *Handbook for Automatic Computation*, **2**, (Linear Algebra) (J. H. Wilkinson and C. Reinsch, eds), 134–151. New York: Springer-Verlag (1971).

[Golub1989] Gene H. Golub, Charles F. Van Loan: *Matrix computations*, 2nd edition (1989).

[Hamelryck2003A] Thomas Hamelryck and Bernard Manderick: PDB parser and structure class implemented in Python. *Bioinformatics* **19** (17): 2308–2310 (2003) https://doi.org/10.1093/bioinformatics/btg299

[Hamelryck2003B] Thomas Hamelryck: Efficient identification of side-chain patterns using a multidimensional index tree. *Proteins* **51** (1): 96–108 (2003). https://doi.org/10.1002/prot.10338

[Hamelryck2005] Thomas Hamelryck: An amino acid has two sides; A new 2D measure provides a different view of solvent exposure. *Proteins* **59** (1): 29–48 (2005). https://doi.org/10.1002/prot.20379

[Henikoff1992] Steven Henikoff, Jorja G. Henikoff: Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences USA* **89** (2): 10915–10919 (1992). https://doi.org/10.1073/pnas.89.22.10915

[Hihara2001] Yukako Hihara, Ayako Kamei, Minoru Kanehisa, Aaron Kaplan and Masahiko Ikeuchi: DNA microarray analysis of cyanobacterial gene expression during acclimation to high light. *Plant Cell* **13** (4): 793–806 (2001). https://doi.org/10.1105/tpc.13.4.793

[Hughey1996] Richard Hughey, Anders Krogh: Hidden Markov models for sequence analysis: extension and analysis of the basic method. *Computer Applications in the Biosciences: CABIOS* **12** (2): 95–107 (1996). https://doi.org/10.1093/bioinformatics/12.2.95

[Jupe2012] Florian Jupe, Leighton Pritchard, Graham J. Etherington, Katrin MacKenzie, Peter JA Cock, Frank Wright, Sanjeev Kumar Sharma, Dan Bolser, Glenn J Bryan, Jonathan DG Jones, Ingo Hein: Identification and localisation of the NB-LRR gene family within the potato genome. *BMC Genomics* **13**: 75 (2012). https://doi.org/10.1186/1471-2164-13-75

[Kachitvichyanukul1988] Voratas Kachitvichyanukul, Bruce W. Schmeiser: Binomial Random Variate Generation. *Communications of the ACM* **31** (2): 216–222 (1988). https://doi.org/10.1145/42372.42381

[Kent2002] W. James Kent: BLAT – The BLAST-Like Alignment Tool. *Genome Research* **12**: 656–664 (2002). https://doi.org/10.1101/gr.229202

[Kohonen1997] Teuvo Kohonen: Self-organizing maps, 2nd Edition. Berlin; New York: Springer-Verlag (1997).

[Krogh1994] Anders Krogh, Michael Brown, I. Saira Mian, Kimmen Sjölander, David Haussler: Hidden Markov Models in computational biology: Applications to protein modeling. *Journal of Molecular Biology* **235** (5): 1501–1531 (1994). https://doi.org/10.1006/jmbi.1994.1104

[Lecuyer1988] Pierre L'Ecuyer: Efficient and Portable Combined Random Number Generators. *Communications of the ACM* **31** (6): 742–749,774 (1988). https://doi.org/10.1145/62959.62969

[Li1985] Wen-Hsiung Li, Chung-I Wu, Chi-Cheng Luo: A new method for estimating synonymous and nonsynonymous rates of nucleotide substitution considering the relative likelihood of nucleotide and codon changes. *Molecular Biology and Evolution* **2** (2): 150–174 (1985). https://doi.org/10.1093/oxfordjournals.molbev.a040343

[Li2009] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin: The Sequence Alignment/Map format and SAMtools. *Bioinformatics* **25** (16): 2078–2079 (2009). https://doi.org/10.1093/bioinformatics/btp352

[Maddison1997] David R. Maddison, David L. Swofford, Wayne P. Maddison: Nexus: An Extensible File Format for Systematic Information. *Systematic Biology* **46** (4): 590–621 (1997). https://doi.org/10.1093/sysbio/46.4.590

[Majumdar2005] Indraneel Majumdar, S. Sri Krishna, Nick V. Grishin: PALSSE: A program to delineate linear secondary structural elements from protein structures. *BMC Bioinformatics* **6**: 202 (2005). https://doi.org/10.1186/1471-2105-6-202.

[Matys2003] V. Matys, E. Fricke, R. Geffers, E. Gößling, M. Haubrock, R. Hehl, K. Hornischer, D. Karas, A.E. Kel, O.V. Kel-Margoulis, D.U. Kloos, S. Land, B. Lewicki-Potapov, H. Michael, R. Münch, I. Reuter, S. Rotert, H. Saxel, M. Scheer, S. Thiele, E. Wingender E: TRANSFAC: transcriptional regulation, from patterns to profiles. *Nucleic Acids Research* **31** (1): 374–378 (2003). https://doi.org/10.1093/nar/gkg108

[Nei1986] Masatoshi Nei and Takashi Gojobori: Simple methods for estimating the numbers of synonymous and nonsynonymous nucleotide substitutions. *Molecular Biology and Evolution* **3** (5): 418–426 (1986). https://doi.org/10.1093/oxfordjournals.molbev.a040410

[Pearson1988] William R. Pearson, David J. Lipman: Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences USA* **85** (8): 2444–2448 (1988). https://doi.org/10.1073/pnas.85.8.2444

[Pritchard2006] Leighton Pritchard, Jennifer A. White, Paul R.J. Birch, Ian K. Toth: GenomeDiagram: a python package for the visualization of large-scale genomic data. *Bioinformatics* **22** (5): 616–617 (2006). https://doi.org/10.1093/bioinformatics/btk021

[Proux2002] Caroline Proux, Douwe van Sinderen, Juan Suarez, Pilar Garcia, Victor Ladero, Gerald F. Fitzgerald, Frank Desiere, Harald Brüssow: The dilemma of phage taxonomy illustrated by comparative genomics of Sfi21-Like Siphoviridae in lactic acid bacteria. *Journal of Bacteriology* **184** (21): 6026–6036 (2002). https://doi.org/10.1128/JB.184.21.6026-6036.2002

[Rice2000] Peter Rice, Ian Longden, Alan Bleasby: EMBOSS: The European Molecular Biology Open Software Suite. *Trends in Genetics* **16** (6): 276–277 (2000). https://doi.org/10.1016/S0168-9525(00)02024-2

[Saldanha2004] Alok Saldanha: Java Treeview—extensible visualization of microarray data. *Bioinformatics* **20** (17): 3246–3248 (2004). https://doi.org/10.1093/bioinformatics/bth349

[Schneider1986] Thomas D. Schneider, Gary D. Stormo, Larry Gold: Information content of binding sites on nucleotide sequences. *Journal of Molecular Biology* **188** (3): 415–431 (1986). https://doi.org/10.1016/0022-2836(86)90165-8

[Schneider2005] Adrian Schneider, Gina M. Cannarozzi, and Gaston H. Gonnet: Empirical codon substitution matrix. *BMC Bioinformatics* **6**: 134 (2005). https://doi.org/10.1186/1471-2105-6-134

[Sibson1973] Robin Sibson: SLINK: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal* **16** (1): 30–34 (1973). https://doi.org/10.1093/comjnl/16.1.30

[Slater2005] Guy St C. Slater, Ewan Birney: Automated generation of heuristics for biological sequence comparison. *BMC Bioinformatics* **6**: 31 (2005). https://doi.org/10.1186/1471-2105-6-31

[Snedecor1989] George W. Snedecor, William G. Cochran: *Statistical methods*. Ames, Iowa: Iowa State University Press (1989).

[Steinegger2019] Martin Steinegger, Markus Meier, Milot Mirdita, Harald Vöhringer, Stephan J. Haunsberger, Johannes Söding: HH-suite3 for fast remote homology detection and deep protein annotation. *BMC Bioinformatics* **20**: 473 (2019). https://doi.org/10.1186/s12859-019-3019-7

[Talevich2012] Eric Talevich, Brandon M. Invergo, Peter J.A. Cock, Brad A. Chapman: Bio.Phylo: A unified toolkit for processing, analyzing and visualizing phylogenetic trees in Biopython. *BMC Bioinformatics* **13**: 209 (2012). https://doi.org/10.1186/1471-2105-13-209

[Tamayo1999] Pablo Tamayo, Donna Slonim, Jill Mesirov, Qing Zhu, Sutisak Kitareewan, Ethan Dmitrovsky, Eric S. Lander, Todd R. Golub: Interpreting patterns of gene expression with self-organizing maps: Methods and

application to hematopoietic differentiation. *Proceedings of the National Academy of Sciences USA* **96** (6): 2907–2912 (1999). https://doi.org/10.1073/pnas.96.6.2907

[Toth2006] Ian K. Toth, Leighton Pritchard, Paul R. J. Birch: Comparative genomics reveals what makes an enterobacterial plant pathogen. *Annual Review of Phytopathology* **44**: 305–336 (2006). https://doi.org/10.1146/annurev.phyto.44.070505.143444

[Vanderauwera2009] Géraldine A. van der Auwera, Jaroslaw E. Król, Haruo Suzuki, Brian Foster, Rob van Houdt, Celeste J. Brown, Max Mergeay, Eva M. Top: Plasmids captured in C. metallidurans CH34: defining the PromA family of broad-host-range plasmids. *Antonie van Leeuwenhoek* **96** (2): 193–204 (2009). https://doi.org/10.1007/s10482-009-9316-9

[Waterman1987] Michael S. Waterman, Mark Eggert: A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons. *Journal of Molecular Biology* **197** (4): 723–728 (1987). https://doi.org/10.1016/0022-2836(87)90478-5

[Yang2000] Ziheng Yang and Rasmus Nielsen: Estimating synonymous and nonsynonymous substitution rates under realistic evolutionary models. *Molecular Biology and Evolution* **17** (1): 32–43 (2000). https://doi.org/10.1093/oxfordjournals.molbev.a026236

[Yeung2001] Ka Yee Yeung, Walter L. Ruzzo: Principal Component Analysis for clustering gene expression data. *Bioinformatics* **17** (9): 763–774 (2001). https://doi.org/10.1093/bioinformatics/17.9.763

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z