biopython

Biopython Tutorial and Cookbook

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

Last Update – 25 May 2020 (Biopython 1.77)

Contents

Chapter 1

Introduction

1.1 What is Biopython?

The Biopython Project is an international association of developers of freely available Python (https://www.python.org) tools for computational molecular biology. Python is an object oriented, interpreted, flexible language that is becoming increasingly popular 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.

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. Basically, the goal of Biopython is to make it as easy as possible to use Python for bioinformatics by creating high-quality, reusable modules and classes. Biopython features include parsers for various Bioinformatics file formats (BLAST, Clustalw, FASTA, Genbank,...), access to online services (NCBI, Expasy,...), interfaces to common and not-so-common programs (Clustalw, DSSP, MSMS...), a standard sequence class, various clustering modules, a KD tree data structure etc. and even documentation.

Basically, we just like to program in Python and want to make it as easy as possible to use Python for bioinformatics by creating high-quality, reusable modules and scripts.

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 [1, Cock *et al.*, 2009] 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: [13, Chapman and Chang, 2000];
   - For `Bio.PDB`: [20, Hamelryck and Manderick, 2003];
   - For `Bio.Cluster`: [15, De Hoon *et al.*, 2004];
   - For `Bio.Graphics.GenomeDiagram`: [2, Pritchard *et al.*, 2006];
   - For `Bio.Phylo` and `Bio.Phylo.PAML`: [9, Talevich *et al.*, 2012];

- For the FASTQ file format as supported in Biopython, BioPerl, BioRuby, BioJava, and EMBOSS: [7, Cock *et al.*, 2010].

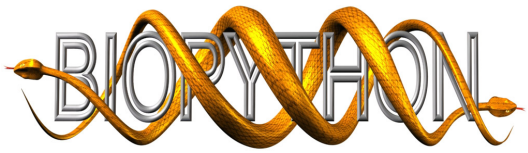
2. *How should I capitalize "Biopython"? Is "BioPython" OK?*
   The correct capitalization is "Biopython", not "BioPython" (even though that would have matched BioPerl, BioJava and BioRuby).

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 developement 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 (ignoring the alphabet other than if giving a warning), 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 3.11.

10. *Why is the* `Seq` *object missing the upper & lower methods described in this Tutorial?*
    You need Biopython 1.53 or later. Alternatively, use `str(my_seq).upper()` to get an upper case string. If you need a Seq object, try `Seq(str(my_seq).upper())` but be careful about blindly re-using the same alphabet.

11. *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.

12. *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 24.1). It is especially important to remember to close output handles explicitly after writing your data.

13. *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.

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

15. *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.

16. *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.

17. *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.

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

19. *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).

20. *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.

21. *Why can't I run command line tools directly from the application wrappers?*
You need Biopython 1.55 or later. Alternatively, use the Python `subprocess` module directly.

22. *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`.

23. *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 2

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/2/).

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 20, this has some cools tricks and tips), the Advanced section (Chapter 22), the built in "docstrings" (via the Python help command, or the API documentation) 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 3.

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")
>>> my_seq
Seq('AGTACACTGGT')
>>> print(my_seq)
AGTACACTGGT
>>> my_seq.alphabet
Alphabet()
```

What we have here is a sequence object with a *generic* alphabet - reflecting the fact we have *not* specified if this is a DNA or protein sequence (okay, a protein with a lot of Alanines, Glycines, Cysteines and Threonines!). We'll talk more about alphabets in Chapter 3.

In addition to having an alphabet, 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 5.

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 2.4, 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 9, extract data from Swiss-Prot from certain orchid proteins in Chapter 10, and work with ClustalW multiple sequence alignments of orchid proteins in Section 6.4.1.

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 5. 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:80/entrez/query.fcgi?db=Nucleotide) 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 `docs/tutorial/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 2.5 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 favourite 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', SingleLetterAlphabet())
740
...
gi|2765564|emb|Z78439.1|PBZ78439
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC', SingleLetterAlphabet())
592
```

Notice that the FASTA format does not specify the alphabet, so `Bio.SeqIO` has defaulted to the rather generic `SingleLetterAlphabet()` rather than something DNA specific.

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', IUPACAmbiguousDNA())
740
...
Z78439.1
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC', IUPACAmbiguousDNA())
592
```

This time `Bio.SeqIO` has been able to choose a sensible alphabet, IUPAC Ambiguous DNA. You'll also 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 5 covers `Bio.SeqIO` in more detail, while Chapter 6 introduces `Bio.AlignIO` 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 20 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 9.
- ExPASy – See Chapter 10.
- 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 20), 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 3

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 4 will introduce the related `SeqRecord` object, which combines the sequence information with any annotation, used again in Chapter 5 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.

There are two important differences between `Seq` objects and standard Python strings. First of all, 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()`. Secondly, the `Seq` object has an important attribute, `alphabet`, which is an object describing what the individual characters making up the sequence string "mean", and how they should be interpreted. For example, is `AGTACACTGGT` a DNA sequence, or just a protein sequence that happens to be rich in Alanines, Glycines, Cysteines and Threonines?

3.1 Sequences and Alphabets

The alphabet object is perhaps the important thing that makes the `Seq` object more than just a string. The currently available alphabets for Biopython are defined in the `Bio.Alphabet` module. We'll use the IUPAC alphabets here to deal with some of our favorite objects: DNA, RNA and Proteins.

`Bio.Alphabet.IUPAC` provides basic definitions for proteins, DNA and RNA, but additionally provides the ability to extend and customize the basic definitions. For instance, for proteins, there is a basic IUPACProtein class, but there is an additional ExtendedIUPACProtein class providing for the additional elements "U" (or "Sec" for selenocysteine) and "O" (or "Pyl" for pyrrolysine), plus the ambiguous symbols "B" (or "Asx" for asparagine or aspartic acid), "Z" (or "Glx" for glutamine or glutamic acid), "J" (or "Xle" for leucine isoleucine) and "X" (or "Xxx" for an unknown amino acid). For DNA you've got choices of IUPACUnambiguousDNA, which provides for just the basic letters, IUPACAmbiguousDNA (which provides for ambiguity letters for every possible situation) and ExtendedIUPACDNA, which allows letters for modified bases. Similarly, RNA can be represented by IUPACAmbiguousRNA or IUPACUnambiguousRNA.

The advantages of having an alphabet class are two fold. First, this gives an idea of the type of information the Seq object contains. Secondly, this provides a means of constraining the information, as a means of type checking.

Now that we know what we are dealing with, let's look at how to utilize this class to do interesting work. You can create an ambiguous sequence with the default generic alphabet like this:

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

```
>>> my_seq
Seq('AGTACACTGGT')
>>> my_seq.alphabet
Alphabet()
```

However, where possible you should specify the alphabet explicitly when creating your sequence objects - in this case an unambiguous DNA alphabet object:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("AGTACACTGGT", IUPAC.unambiguous_dna)
>>> my_seq
Seq('AGTACACTGGT', IUPACUnambiguousDNA())
>>> my_seq.alphabet
IUPACUnambiguousDNA()
```

Unless of course, this really is an amino acid sequence:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_prot = Seq("AGTACACTGGT", IUPAC.protein)
>>> my_prot
Seq('AGTACACTGGT', IUPACProtein())
>>> my_prot.alphabet
IUPACProtein()
```

3.2 Sequences act like strings

In many 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
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCG", IUPAC.unambiguous_dna)
>>> 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
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
>>> len(my_seq)
32
>>> my_seq.count("G")
9
>>> 100 * float(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.Alphabet import IUPAC
>>> from Bio.SeqUtils import GC
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
>>> GC(my_seq)
46.875
```

Note that using the `Bio.SeqUtils.GC()` 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 3.12 below which talks about the `MutableSeq` object.

3.3 Slicing a sequence

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

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

Two things are interesting to note. First, this follows the normal conventions for Python strings. So the first element of the sequence is 0 (which is normal for computer science, but not so normal for biology). 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), which is the way things work in Python, but of course not necessarily the way everyone in the world would expect. The main goal is to stay consistent with what Python does.

The second thing to notice is that the slice is performed on the sequence data string, but the new object produced is another `Seq` object which retains the alphabet information from the original `Seq` object.

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', IUPACUnambiguousDNA())
>>> my_seq[1::3]
Seq('AGGCATGCATC', IUPACUnambiguousDNA())
>>> my_seq[2::3]
Seq('TAGCTAAGAC', IUPACUnambiguousDNA())
```

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', IUPACUnambiguousDNA())
```

3.4 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
<BLANKLINE>
```

This line of code constructs a simple FASTA format record (without worrying about line wrapping). Section 4.6 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 5.

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

3.5 Concatenating or adding sequences

Naturally, you can in principle add any two Seq objects together - just like you can with Python strings to concatenate them. However, you can't add sequences with incompatible alphabets, such as a protein sequence and a DNA sequence:

```
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Seq import Seq
>>> protein_seq = Seq("EVRNAK", IUPAC.protein)
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> protein_seq + dna_seq
Traceback (most recent call last):
...
TypeError: Incompatible alphabets IUPACProtein() and IUPACUnambiguousDNA()
```

If you *really* wanted to do this, you'd have to first give both sequences generic alphabets:

```
>>> from Bio.Alphabet import generic_alphabet
>>> protein_seq.alphabet = generic_alphabet
>>> dna_seq.alphabet = generic_alphabet
>>> protein_seq + dna_seq
Seq('EVRNAKACGT')
```

Here is an example of adding a generic nucleotide sequence to an unambiguous IUPAC DNA sequence, resulting in an ambiguous nucleotide sequence:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_nucleotide
>>> from Bio.Alphabet import IUPAC
>>> nuc_seq = Seq("GATCGATGC", generic_nucleotide)
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> nuc_seq
Seq('GATCGATGC', NucleotideAlphabet())
>>> dna_seq
Seq('ACGT', IUPACUnambiguousDNA())
>>> nuc_seq + dna_seq
Seq('GATCGATGCACGT', NucleotideAlphabet())
```

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

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

Or, a more elegant approach is to the use built in `sum` function with its optional start value argument (which otherwise defaults to zero):

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> list_of_seqs = [Seq("ACGT", generic_dna), Seq("AACC", generic_dna), Seq("GGTT", generic_dna)]
>>> sum(list_of_seqs, Seq("", generic_dna))
Seq('ACGTAACCGGTT', DNAAlphabet())
```

Unlike the Python string, the Biopython `Seq` does not (currently) have a `.join` method.

3.6 Changing case

Python strings have very useful `upper` and `lower` methods for changing the case. As of Biopython 1.53, the `Seq` object gained similar methods which are alphabet aware. For example,

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

These are useful for doing case insensitive matching:

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

Note that strictly speaking the IUPAC alphabets are for upper case sequences only, thus:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> dna_seq
Seq('ACGT', IUPACUnambiguousDNA())
>>> dna_seq.lower()
Seq('acgt', DNAAlphabet())
```

3.7 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
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
>>> my_seq
Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', IUPACUnambiguousDNA())
>>> my_seq.complement()
Seq('CTAGCTACCCGGATATATCCTAGCTTTTAGCG', IUPACUnambiguousDNA())
>>> my_seq.reverse_complement()
Seq('GCGATTTTCGATCCTATATAGGCCCATCGATC', IUPACUnambiguousDNA())
```

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', IUPACUnambiguousDNA())
```

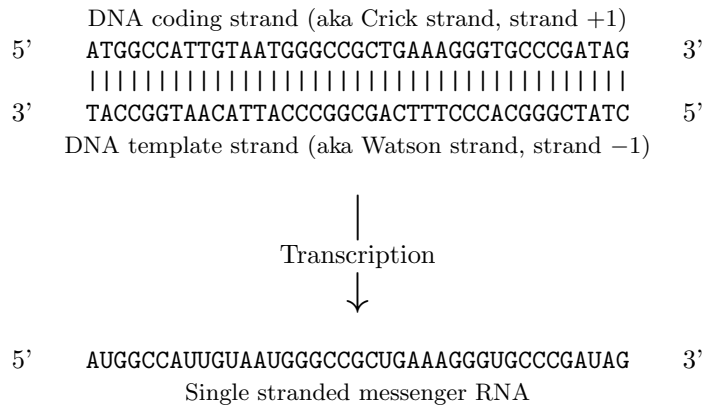
In all of these operations, the alphabet property is maintained. This is very useful in case you accidentally end up trying to do something weird like take the (reverse)complement of a protein sequence:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> protein_seq = Seq("EVRNAK", IUPAC.protein)
>>> protein_seq.complement()
Traceback (most recent call last):
...
ValueError: Proteins do not have complements!
```

The example in Section 5.5.3 combines the `Seq` object's reverse complement method with `Bio.SeqIO` for sequence input/output.

3.8 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:



The actual biological transcription process works from the template strand, doing a reverse complement (TCAG → 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 → 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
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous_dna)
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> template_dna = coding_dna.reverse_complement()
>>> template_dna
Seq('CTATCGGGCACCCTTTCAGCGGCCCATTACAATGGCCAT', IUPACUnambiguousDNA())
```

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', IUPACUnambiguousDNA())
>>> messenger_rna = coding_dna.transcribe()
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
```

As you can see, all this does is switch T → U, and adjust the alphabet.

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', IUPACUnambiguousRNA())
```

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 → T substitution and associated change of alphabet:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG", IUPAC.unambiguous_rna)
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
>>> messenger_rna.back_transcribe()
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
```

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 3.14.

3.9 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
>>> from Bio.Alphabet import IUPAC
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG", IUPAC.unambiguous_rna)
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
>>> messenger_rna.translate()
Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

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

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous_dna)
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> coding_dna.translate()
Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

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*', HasStopCodon(IUPACProtein(), '*'))
```

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*', HasStopCodon(IUPACProtein(), '*'))
```

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

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@', HasStopCodon(IUPACProtein(), '@'))
```

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
>>> from Bio.Alphabet import generic_dna
>>> gene = Seq("GTGAAAAAGATGCAATCTATCGTACTCGCACTTTCCCTGGTTCTGGTCGCTCCCATGGCA" + \
...            "GCACAGGCTGCGGAAATTACGTTAGTCCCGTCAGTAAAATTACAGATAGGCGATCGTGAT" + \
...            "AATCGTGGCTATTACTGGGATGGAGGTCACTGGCGCGACCACGGCTGGTGGAAACAACAT" + \
...            "TATGAATGGCGAGGCAATCGCTGGCACCTACACGGACCGCCGCCACCGCCGCGCCACCAT" + \
...            "AAGAAAGCTCCTCATGATCATCACGGCGGTCATGGTCCAGGCAAACATCACCGCTAA",
...            generic_dna)
>>> gene.translate(table="Bacterial")
Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HR*',
HasStopCodon(ExtendedIUPACProtein(), '*')
```

```
>>> gene.translate(table="Bacterial", to_stop=True)
Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HHR',
ExtendedIUPACProtein())
```

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',
ExtendedIUPACProtein())
```

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 20.1.3 combines the `Seq` object's translate method with `Bio.SeqIO` for sequence input/output.

3.10 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 3.14). 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      |
--+---------+---------+---------+---------+--
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
['TAA', 'TAG', 'AGA', 'AGG']
>>> mito_table.start_codons
['ATT', 'ATC', 'ATA', 'ATG', 'GTG']
>>> mito_table.forward_table["ACG"]
'T'
```

3.11 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 uses alphabet objects as part of each `Seq` object to try to capture this information - so comparing two `Seq` objects could mean considering both the sequence strings *and* the alphabets.

For example, you might argue that the two DNA `Seq` objects `Seq("ACGT", IUPAC.unambiguous_dna)` and `Seq("ACGT", IUPAC.ambiguous_dna)` should be equal, even though they do have different alphabets. Depending on the context this could be important.

This gets worse – suppose you think `Seq("ACGT", IUPAC.unambiguous_dna)` and `Seq("ACGT")` (i.e. the default generic alphabet) should be equal. Then, logically, `Seq("ACGT", IUPAC.protein)` and `Seq("ACGT")` should also be equal. Now, in logic if $A = B$ and $B = C$, by transitivity we expect $A = C$. So for logical consistency we'd require `Seq("ACGT", IUPAC.unambiguous_dna)` and `Seq("ACGT", IUPAC.protein)` to be equal – which most people would agree is just not right. This transitivity also has implications for using `Seq` objects as Python dictionary keys.

Now, in everyday use, your sequences will probably all have the same alphabet, or at least all be the same type of sequence (all DNA, all RNA, or all protein). What you probably want is to just compare the sequences as strings – which you can do explicitly:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> seq1 = Seq("ACGT", IUPAC.unambiguous_dna)
>>> seq2 = Seq("ACGT", IUPAC.ambiguous_dna)
>>> str(seq1) == str(seq2)
True
>>> str(seq1) == str(seq1)
True
```

So, what does Biopython do? Well, as of Biopython 1.65, sequence comparison only looks at the sequence, essentially ignoring the alphabet:

```
>>> seq1 == seq2
True
>>> seq1 == "ACGT"
True
```

As an extension to this, using sequence objects as keys in a Python dictionary is now equivalent to using the sequence as a plain string for the key. See also Section 3.4.

Note if you compare sequences with incompatible alphabets (e.g. DNA vs RNA, or nucleotide versus protein), then you will get a warning but for the comparison itself only the string of letters in the sequence is used:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna, generic_protein
>>> dna_seq = Seq("ACGT", generic_dna)
>>> prot_seq = Seq("ACGT", generic_protein)
>>> dna_seq == prot_seq
BiopythonWarning: Incompatible alphabets DNAAlphabet() and ProteinAlphabet()
True
```

WARNING: Older versions of Biopython instead used to check if the `Seq` objects were the same object in memory. This is important if you need to support scripts on both old and new versions of Biopython. Here make the comparison explicit by wrapping your sequence objects with either `str(...)` for string based comparison or `id(...)` for object instance based comparison.

3.12 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
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous_dna)
```

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:

```
>>> mutable_seq = my_seq.tomutable()
>>> mutable_seq
MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
```

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

```
>>> from Bio.Seq import MutableSeq
>>> from Bio.Alphabet import IUPAC
>>> mutable_seq = MutableSeq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous_dna)
```

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

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

Do note that unlike the Seq object, the MutableSeq object's methods like reverse_complement() and reverse() act in-situ!

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:

```
>>> new_seq = mutable_seq.toseq()
>>> new_seq
Seq('AGCCCGTGGGAAAGTCGCCGGGTAATGCACCG', IUPACUnambiguousDNA())
```

You can also get a string from a MutableSeq object just like from a Seq object (Section 3.4).

3.13 UnknownSeq objects

The `UnknownSeq` object is a subclass of the basic `Seq` object and its purpose is to represent a sequence where we know the length, but not the actual letters making it up. You could of course use a normal `Seq` object in this situation, but it wastes rather a lot of memory to hold a string of a million "N" characters when you could just store a single letter "N" and the desired length as an integer.

```
>>> from Bio.Seq import UnknownSeq
>>> unk = UnknownSeq(20)
>>> unk
UnknownSeq(20, character='?')
>>> print(unk)
????????????????????
>>> len(unk)
20
```

You can of course specify an alphabet, meaning for nucleotide sequences the letter defaults to "N" and for proteins "X", rather than just "?".

```
>>> from Bio.Seq import UnknownSeq
>>> from Bio.Alphabet import IUPAC
>>> unk_dna = UnknownSeq(20, alphabet=IUPAC.ambiguous_dna)
>>> unk_dna
UnknownSeq(20, alphabet=IUPACAmbiguousDNA(), character='N')
>>> print(unk_dna)
NNNNNNNNNNNNNNNNNNNN
```

You can use all the usual `Seq` object methods too, note these give back memory saving `UnknownSeq` objects where appropriate as you might expect:

```
>>> unk_dna
UnknownSeq(20, alphabet=IUPACAmbiguousDNA(), character='N')
>>> unk_dna.complement()
UnknownSeq(20, alphabet=IUPACAmbiguousDNA(), character='N')
>>> unk_dna.reverse_complement()
UnknownSeq(20, alphabet=IUPACAmbiguousDNA(), character='N')
>>> unk_dna.transcribe()
UnknownSeq(20, alphabet=IUPACAmbiguousRNA(), character='N')
>>> unk_protein = unk_dna.translate()
>>> unk_protein
UnknownSeq(6, alphabet=ProteinAlphabet(), character='X')
>>> print(unk_protein)
XXXXXX
>>> len(unk_protein)
6
```

You may be able to find a use for the `UnknownSeq` object in your own code, but it is more likely that you will first come across them in a `SeqRecord` object created by `Bio.SeqIO` (see Chapter 5). Some sequence file formats don't always include the actual sequence, for example GenBank and EMBL files may include a list of features but for the sequence just present the contig information. Alternatively, the QUAL files used in sequencing work hold quality scores but they *never* contain a sequence – instead there is a partner FASTA file which *does* have the sequence.

3.14 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 (including `UnknownSeq` 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 4

Sequence annotation objects

Chapter 3 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 5.

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 (also online – SeqRecord and 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 3), and is the basic data type for the `Bio.SeqIO` sequence input/output interface (see Chapter 5).

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 20.1.6) 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 4.3.

.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 5 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 2 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 5 for details):

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.fna", "fasta")
>>> record
SeqRecord(seq=Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG',
SingleLetterAlphabet()), 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 ... 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', SingleLetterAlphabet())
```

Here `Bio.SeqIO` has defaulted to a generic alphabet, rather than guessing that this is DNA. If you know in advance what kind of sequence your FASTA file contains, you can tell `Bio.SeqIO` which alphabet to use (see Chapter 5).

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 ... 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 5 for details):

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.gb", "genbank")
>>> record
SeqRecord(seq=Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG',
IUPACAmbiguousDNA()), id='NC_005816.1', name='NC_005816',
description='Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.',
dbxrefs=['Project:10638'])
```

You should be able to spot some differences already! But taking the attributes individually, the sequence string is the same as before, but this time `Bio.SeqIO` has been able to automatically assign a more specific alphabet (see Chapter 5 for details):

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

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)
11
>>> record.annotations["source"]
'Yersinia pestis biovar Microtus str. 91001'
```

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

```
>>> record.dbxrefs
['Project:10638']
```

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)
29
```

We'll talk about `SeqFeature` objects next, in Section 4.3.

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 4.3.2 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 4.3.2 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.

4.3.2.1 FeatureLocation 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 `FeatureLocation` object does.

In practise 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).

4.3.2.2 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.

4.3.2.3 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 do 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. To get the information in this class you have to look at two attributes. The `position` attribute specifies the lower boundary of the range we are looking at, so in our example case this would be one. The `extension` attribute specifies the range to the higher boundary, so in this case it would be 4. So `object.position` is the lower boundary and `object.position + object.extension` is the upper boundary.

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.FeatureLocation(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 `FeatureLocation` 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
```

For compatibility with older versions of Biopython you can ask for the `nofuzzy_start` and `nofuzzy_end` attributes of the location which are plain integers:

```
>>> my_location.nofuzzy_start
5
>>> my_location.nofuzzy_end
9
```

Notice that this just gives you back the position attributes of the fuzzy locations.

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.FeatureLocation(5, 9)
>>> print(exact_location)
[5:9]
>>> exact_location.start
ExactPosition(5)
>>> int(exact_location.start)
5
>>> exact_location.nofuzzy_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!

4.3.2.4 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")
```

```
>>> 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 4.3.2) 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, FeatureLocation
>>> example_parent = Seq("ACCGAGACGGCAAAGGCTAGCATAGGTATGAGACTTCCTTCCTGCCAGTGCTGAGGAACTGGGAGCCTAC")
>>> example_feature = SeqFeature(FeatureLocation(5, 18), type="gene", strand=-1)
```

You could take the parent sequence, slice it to extract 5:18, and then take the reverse complement. If you are using Biopython 1.59 or later, the feature location's start and end are integer like so this works:

```
>>> feature_seq = example_parent[example_feature.location.start:example_feature.location.end].reverse_c
>>> 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:

```
>>> feature_seq = example_feature.extract(example_parent)
>>> print(feature_seq)
AGCCTTTGCCGTC
```

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

```
>>> print(example_feature.extract(example_parent))
AGCCTTTGCCGTC
>>> print(len(example_feature.extract(example_parent)))
13
>>> print(len(example_feature))
13
>>> print(len(example_feature.location))
13
```

For simple `FeatureLocation` 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 attri
```

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 3.11).

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 4.3.2.

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
from Bio.Alphabet import generic_protein

record = SeqRecord(
    Seq(
        "MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQALFGD"
        "GAGAVIVGSDPDLSVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKDVPGLISK"
        "NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRATREVLSEYGNM"
        "SSAC",
        generic_protein,
    ),
    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 5). 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 5.5.4.

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',
IUPACAmbiguousDNA()), 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
```

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']
<BLANKLINE>

>>> 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 ...']
    Key: product, Value: ['pesticin immunity protein']
    Key: protein_id, Value: ['NP_995571.1']
    Key: transl_table, Value: ['11']
    Key: translation, Value: ['MGGGMISKLFCLALIFLSSSGLAEKNTYTAKDILQNLELNTFGNSLSH...']
```

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',
IUPACAmbiguousDNA()), 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
location: [42:480](+)
```

```
qualifiers:
    Key: db_xref, Value: ['GeneID:2767712']
    Key: gene, Value: ['pim']
    Key: locus_tag, Value: ['YP_pPCP05']
<BLANKLINE>

>>> 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 ...']
    Key: product, Value: ['pesticin immunity protein']
    Key: protein_id, Value: ['NP_995571.1']
    Key: transl_table, Value: ['11']
    Key: translation, Value: ['MGGGMISKLFCLALIFLSSSGLAEKNTYTAKDILQNLELNTFGNSLSH...']
```

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, 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
{}
>>> sub_record.dbxrefs
[]
```

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 4.6:

```
>>> sub_record.description = "Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, partial."
>>> print(sub_record.format("genbank"))
...
```

See Sections 20.1.7 and 20.1.8 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 5 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

>>> 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',
IUPACAmbiguousDNA()), id='NC_005816.1', name='NC_005816',
description='Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.',
dbxrefs=['Project:10638'])

>>> len(record)
9609
>>> len(record.features)
41
>>> record.dbxrefs
['Project:58037']

>>> record.annotations.keys()
['comment', 'sequence_version', 'source', 'taxonomy', 'keywords', 'references',
'accessions', 'data_file_division', 'date', 'organism', 'gi']
```

You can shift the origin like this:

```
>>> shifted = record[2000:] + record[:2000]

>>> shifted
SeqRecord(seq=Seq('GATACGCAGTCATATTTTTTACACAATTCTCTAATCCCGACAAGGTCGTAGGTC...GGA',
IUPACAmbiguousDNA()), 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 and one of the features (the source feature) have been lost:

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

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()
>>> shifted.dbxrefs
['Project:10638']
>>> shifted.annotations.keys()
['comment', 'sequence_version', 'source', 'taxonomy', 'keywords', 'references',
'accessions', 'data_file_division', 'date', 'organism', 'gi']
```

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
>>> record = SeqIO.read("NC_005816.gb", "genbank")
>>> print("%s %i %i %i %i" % (record.id, len(record), len(record.features), len(record.dbxrefs), len(re
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 = record.reverse_complement(id="TESTING")
>>> print("%s %i %i %i %i" % (rc.id, len(rc), len(rc.features), len(rc.dbxrefs), len(rc.annotations)))
TESTING 9609 41 0 0
```

Chapter 5

Sequence Input/Output

In this chapter we'll discuss in more detail the `Bio.SeqIO` module, which was briefly introduced in Chapter 2 and also used in Chapter 4. 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 (also online):

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

The "catch" is that you have to work with `SeqRecord` objects (see Chapter 4), which contain a `Seq` object (see Chapter 3) 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 5.6).

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 5.3). See Section 24.1 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.

There is an optional argument `alphabet` to specify the alphabet to be used. This is useful for file formats like FASTA where otherwise `Bio.SeqIO` will default to a generic alphabet.

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 2.4, 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 (also online).

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 20.2 (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")
```

```
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
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC', IUPACAmbiguousDNA())
```

```
592
The first record
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', IUPACAmbiguousDNA())
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 4. 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', IUPACAmbiguousDNA())
```

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 a list of 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 standardised 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)
```

This gives:

```
['C.irapeanum', 'C.californicum', 'C.fasciculatum', 'C.margaritaceum', ..., '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)
```

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 5.5), then you should modify both the `id` and `description` attributes. To ensure the correct behaviour, 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 24.1), 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 5.4.4.

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 9.6 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 9 for more about the `Bio.Entrez` module, and make sure to read about the NCBI guidelines for using Entrez (Section 9.1).

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 10. 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', ProteinAlphabet())
Length 394
['Acyltransferase', 'Flavonoid biosynthesis', 'Transferase']
```

5.4 Sequence files as Dictionaries

We're now going to introduce three related functions in the `Bio.SeqIO` 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 5.4.1). 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 5.4.2).
- `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 5.4.3), but will be a little bit slower.

See the discussion for an broad overview (Section 5.4.5).

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 5.4.2 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
>>> print(repr(seq_record.seq))
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGATCACAT...GGT', IUPACAmbiguousDNA())
```

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.

5.4.1.1 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 recognise these strings from when we parsed the FASTA file earlier in Section 2.4.1. 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!

5.4.1.2 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', IUPACAmbiguousDNA())
>>> 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 20.1.11). However, alignment formats like PHYLIP or Clustal are not supported. Finally as an optional argument you can supply an alphabet, or 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']
```

5.4.2.1 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 5.4.1.1, 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?

5.4.2.2 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:
...     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 20.1.5 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")
>>> 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.
```

5.4.3.1 Getting the raw data for a record

Just as with the `Bio.SeqIO.index()` function discussed above in Section 5.4.2.2, 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) popularised 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
from Bio.Alphabet import generic_protein

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

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

rec3 = SeqRecord(
    Seq(
        "MVTVEEFRRAQCAEGPATVMAIGTATPSNCVDQSTYPDYYFRITNSEHKVELKEKFKRMC"
        "EKSMIKKRYMHLTEEILKENPNICAYMAPSLDARQDIVVVEVPKLGKEAAQKAIKEWGQP"
        "KSKITHLVFCTTSGVDMPGCDYQLTKLLGLRPSVKRFMMYQQGCFAGGTVLRMAKDLAEN"
```

```
        "NKGARVLVVCSEITAVTFRGPNDTHLDSLVGQALFGDGAAAVIIGSDPIPEVERPLFELV"
        "SAAQTLLPDSEGAIDGHLREVGLTFHLLKDVPGLISKNIEKSLVEAFQPLGISDWNSLFW"
        "IAHPGGPAILDQVELKLGLKQEKLKATRKVLSNYGNMSSACVLFILDEMRKASAKEGLGT"
        "TGEGLEWGVLFGFGPGLTVETVVLHSVAT",
        generic_protein,
    ),
    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 favourite 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 5.4.2.2 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 20.1.9 and 20.1.10 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 optimisations 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 3.7):

```
>>> 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 4.9) 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 20.1.3, 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 4.6).

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-througput data), then outputing 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 20), 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 6

Multiple Sequence Alignment objects

This chapter is about Multiple Sequence Alignments, by which 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 our command line wrappers for common multiple sequence alignment tools like ClustalW and MUSCLE.

6.1 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 *i*terator 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 *h*andle to read the data from, typically an open file (see Section 24.1), 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 6.1.3 below for dealing with ambiguous file formats which may contain more than one alignment.

A further optional `alphabet` argument allowing you to specify the expected alphabet. This can be useful as many alignment file formats do not explicitly label the sequences as RNA, DNA or protein – which means `Bio.AlignIO` will default to using a generic alphabet.

6.1.1 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--HHHHHHHHHHHHHHHHHHHHH----
Q9T0Q8_BPIKE/1-52             AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA
COATB_BPI22/32-83             DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA
COATB_BPM13/24-72             AEGDDP...AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
#=GR COATB_BPM13/24-72  SS    ---S-T...CHCHHHHCCCCTCCCTTCHHHHHHHHHHHHHHHHHHHHCTT--
COATB_BPZJ2/1-49              AEGDDP...AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA
Q9T0Q9_BPFD/1-49              AEGDDP...AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
#=GR Q9T0Q9_BPFD/1-49   SS    ------...-HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH--
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)
SingleLetterAlphabet() 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 6.2.2 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 (also online):

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

6.1.2 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
```

```
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:

```
SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAACCA Alpha
AAACCC Beta
ACCCCA Gamma
CCCAAC Delta
CCCAAA Epsilon

SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAACAA Alpha
AAACCC Beta
ACCCAA Gamma
CCCACC Delta
CCCAAA Epsilon

SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAAAAC Alpha
AAACCC Beta
AACAAC Gamma
CCCCCA Delta
CCCAAC Epsilon

...

SingleLetterAlphabet() 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]
```

6.1.3 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
>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.

6.2 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.Alphabet import generic_dna
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> align1 = MultipleSeqAlignment(
...     [
...         SeqRecord(Seq("ACTGCTAGCTAG", generic_dna), id="Alpha"),
...         SeqRecord(Seq("ACT-CTAGCTAG", generic_dna), id="Beta"),
...         SeqRecord(Seq("ACTGCTAGDTAG", generic_dna), id="Gamma"),
...     ]
... )
>>> align2 = MultipleSeqAlignment(
...     [
...         SeqRecord(Seq("GTCAGC-AG", generic_dna), id="Delta"),
...         SeqRecord(Seq("GACAGCTAG", generic_dna), id="Epsilon"),
...         SeqRecord(Seq("GTCAGCTAG", generic_dna), id="Zeta"),
...     ]
... )
>>> align3 = MultipleSeqAlignment(
...     [
...         SeqRecord(Seq("ACTAGTACAGCTG", generic_dna), id="Eta"),
...         SeqRecord(Seq("ACTAGTACAGCT-", generic_dna), id="Theta"),
...         SeqRecord(Seq("-CTACTACAGGTG", generic_dna), 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 favourite 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
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.

6.2.1 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 5.5.2). 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
```

```
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 identifers to be given in full:

```
 7 52
COATB_BPIKE/30-81  AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIRLFKKFSS
```

```
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 identifers 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: 'CO
>>> 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
```

```
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.

6.2.2 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")) # doctest:+ELLIPSIS
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 4.6, 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
...
```

6.3 Manipulating Alignments

Now that we've covered loading and saving alignments, we'll look at what else you can do with them.

6.3.1 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)
SingleLetterAlphabet() 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])
SingleLetterAlphabet() 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])
SingleLetterAlphabet() 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])
SingleLetterAlphabet() 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])
SingleLetterAlphabet() 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:])
SingleLetterAlphabet() 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)
SingleLetterAlphabet() 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 4.8 for how adding `SeqRecord` objects works). You may find it helpful to first sort the alignment rows alphabetically by id:

```
>>> edited.sort()
>>> print(edited)
SingleLetterAlphabet() 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.

6.3.2 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([list(rec) for rec in alignment], np.character)
>>> print("Array shape %i by %i" % align_array.shape)
Array shape 7 by 52
```

If you will be working heavily with the columns, you can tell NumPy to store the array by column (as in Fortran) rather then its default of by row (as in C):

```
>>> align_array = np.array([list(rec) for rec in alignment], np.character, order="F")
```

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!

6.4 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 Biopython contains the `Bio.pairwise2` module , which is supplemented by functions written in C for speed enhancements and the new `PairwiseAligner` (see Section 6.5). In addition, you can use Biopython to invoke a command line tool on your behalf. 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 5).
2. Call the command line tool to process this input file, typically via one of Biopython's command line wrappers (which we'll discuss here).
3. Read the output from the tool, i.e. your aligned sequences, typically using `Bio.AlignIO` (see earlier in this chapter).

All the command line wrappers we're going to talk about in this chapter follow the same style. You create a command line object specifying the options (e.g. the input filename and the output filename), then invoke this command line via a Python operating system call (e.g. using the `subprocess` module).

Most of these wrappers are defined in the `Bio.Align.Applications` module:

```
>>> import Bio.Align.Applications
>>> dir(Bio.Align.Applications) # doctest:+ELLIPSIS
['ClustalOmegaCommandline', 'ClustalwCommandline', 'DialignCommandline', 'MSAProbsCommandline', 'MafftC
```

(Ignore the entries starting with an underscore – these have special meaning in Python.) The module `Bio.Emboss.Applications` has wrappers for some of the EMBOSS suite, including `needle` and `water`, which are described below in Section 6.4.5, and wrappers for the EMBOSS packaged versions of the PHYLIP tools (which EMBOSS refer to as one of their EMBASSY packages - third party tools with an EMBOSS style interface). We won't explore all these alignment tools here in the section, just a sample, but the same principles apply.

6.4.1 ClustalW

ClustalW is a popular command line tool for multiple sequence alignment (there is also a graphical interface called ClustalX). Biopython's `Bio.Align.Applications` module has a wrapper for this alignment tool (and several others).

Before trying to use ClustalW from within Python, you should first try running the ClustalW tool yourself by hand at the command line, to familiarise yourself the other options. You'll find the Biopython wrapper is very faithful to the actual command line API:

```
>>> from Bio.Align.Applications import ClustalwCommandline
>>> help(ClustalwCommandline)
```

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:

```
>>> from Bio.Align.Applications import ClustalwCommandline
>>> cline = ClustalwCommandline("clustalw2", infile="opuntia.fasta")
>>> print(cline)
clustalw2 -infile=opuntia.fasta
```

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. For example:

```
>>> import os
>>> from Bio.Align.Applications import ClustalwCommandline
>>> clustalw_exe = r"C:\Program Files\new clustal\clustalw2.exe"
>>> clustalw_cline = ClustalwCommandline(clustalw_exe, infile="opuntia.fasta")

>>> assert os.path.isfile(clustalw_exe), "Clustal W executable missing"
>>> stdout, stderr = clustalw_cline()
```

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.

Internally this uses the `subprocess` module which is now the recommended way to run another program in Python. This replaces older options like the `os.system()` and the `os.popen*` functions.

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.

When you run the command line tool like this via the Biopython wrapper, it will wait for it to finish, and check the return code. If this is non zero (indicating an error), an exception is raised. The wrapper then returns two strings, stdout and stderr.

In the case of ClustalW, 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 (via stdout or stderr) is boring and can be ignored (assuming it worked).

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)
SingleLetterAlphabet() 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)
                             _____________ 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
<BLANKLINE>
```

Chapter 13 covers Biopython's support for phylogenetic trees in more depth.

6.4.2 MUSCLE

MUSCLE is a more recent multiple sequence alignment tool than ClustalW, and Biopython also has a wrapper for it under the `Bio.Align.Applications` module. As before, we recommend you try using MUSCLE from the command line before trying it from within Python, as the Biopython wrapper is very faithful to the actual command line API:

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> help(MuscleCommandline)
```

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:

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> cline = MuscleCommandline(input="opuntia.fasta", out="opuntia.txt")
>>> print(cline)
muscle -in opuntia.fasta -out opuntia.txt
```

Note that MUSCLE uses "-in" and "-out" but in Biopython we have to use "input" and "out" as the keyword arguments or property names. This is because "in" is a reserved word in Python.

By default MUSCLE will output the alignment as a FASTA file (using gapped sequences). The `Bio.AlignIO` module should be able to read this alignment using `format="fasta"`. You can also ask for ClustalW-like output:

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> cline = MuscleCommandline(input="opuntia.fasta", out="opuntia.aln", clw=True)
>>> print(cline)
muscle -in opuntia.fasta -out opuntia.aln -clw
```

Or, strict ClustalW output where the original ClustalW header line is used for maximum compatibility:

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> cline = MuscleCommandline(input="opuntia.fasta", out="opuntia.aln", clwstrict=True)
>>> print(cline)
muscle -in opuntia.fasta -out opuntia.aln -clwstrict
```

The `Bio.AlignIO` module should be able to read these alignments using `format="clustal"`.

MUSCLE can also output in GCG MSF format (using the `msf` argument), but Biopython can't currently parse that, or using HTML which would give a human readable web page (not suitable for parsing).

You can also set the other optional parameters, for example the maximum number of iterations. See the built in help for details.

You would then run MUSCLE command line string as described above for ClustalW, and parse the output using `Bio.AlignIO` to get an alignment object.

6.4.3 MUSCLE using stdout

Using a MUSCLE command line as in the examples above will write the alignment to a file. This means there will be no important information written to the standard out (stdout) or standard error (stderr) handles. However, by default MUSCLE will write the alignment to standard output (stdout). We can take advantage of this to avoid having a temporary output file! For example:

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> muscle_cline = MuscleCommandline(input="opuntia.fasta")
>>> print(muscle_cline)
muscle -in opuntia.fasta
```

If we run this via the wrapper, we get back the output as a string. In order to parse this we can use `StringIO` to turn it into a handle. Remember that MUSCLE defaults to using FASTA as the output format:

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> muscle_cline = MuscleCommandline(input="opuntia.fasta")
>>> stdout, stderr = muscle_cline()
>>> from io import StringIO
>>> from Bio import AlignIO
>>> align = AlignIO.read(StringIO(stdout), "fasta")
>>> print(align)
SingleLetterAlphabet() 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
```

The above approach is fairly simple, but if you are dealing with very large output text the fact that all of stdout and stderr is loaded into memory as a string can be a potential drawback. Using the `subprocess` module we can work directly with handles instead:

```
>>> import subprocess
>>> from Bio.Align.Applications import MuscleCommandline
>>> muscle_cline = MuscleCommandline(input="opuntia.fasta")
>>> child = subprocess.Popen(str(muscle_cline),
...                          stdout=subprocess.PIPE,
```

```
...                             stderr=subprocess.PIPE,
...                             universal_newlines=True,
...                             shell=(sys.platform!="win32"))
...
>>> from Bio import AlignIO
>>> align = AlignIO.read(child.stdout, "fasta")
>>> print(align)
SingleLetterAlphabet() 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
```

6.4.4 MUSCLE using stdin and stdout

We don't actually *need* to have our FASTA input sequences prepared in a file, because by default MUSCLE will read in the input sequence from standard input! Note this is a bit more advanced and fiddly, so don't bother with this technique unless you need to.

First, we'll need some unaligned sequences in memory as `SeqRecord` objects. For this demonstration I'm going to use a filtered version of the original FASTA file (using a generator expression), taking just six of the seven sequences:

```
>>> from Bio import SeqIO
>>> records = (r for r in SeqIO.parse("opuntia.fasta", "fasta") if len(r) < 900)
```

Then we create the MUSCLE command line, leaving the input and output to their defaults (stdin and stdout). I'm also going to ask for strict ClustalW format as for the output.

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> muscle_cline = MuscleCommandline(clwstrict=True)
>>> print(muscle_cline)
muscle -clwstrict
```

Now for the fiddly bits using the `subprocess` module, stdin and stdout:

```
>>> import subprocess
>>> import sys
>>> child = subprocess.Popen(str(cline),
...                          stdin=subprocess.PIPE,
...                          stdout=subprocess.PIPE,
...                          stderr=subprocess.PIPE,
...                          universal_newlines=True,
...                          shell=(sys.platform!="win32"))
```

That should start MUSCLE, but it will be sitting waiting for its FASTA input sequences, which we must supply via its stdin handle:

```
>>> SeqIO.write(records, child.stdin, "fasta")
6
>>> child.stdin.close()
```

After writing the six sequences to the handle, MUSCLE will still be waiting to see if that is all the FASTA sequences or not – so we must signal that this is all the input data by closing the handle. At that point MUSCLE should start to run, and we can ask for the output:

```
>>> from Bio import AlignIO
>>> align = AlignIO.read(child.stdout, "clustal")
>>> print(align)
SingleLetterAlphabet() alignment with 6 rows and 900 columns
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.1|AF19166
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.1|AF19166
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.1|AF19166
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.1|AF19166
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.1|AF19165
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.1|AF19165
```

Wow! There we are with a new alignment of just the six records, without having created a temporary FASTA input file, or a temporary alignment output file. However, a word of caution: Dealing with errors with this style of calling external programs is much more complicated. It also becomes far harder to diagnose problems, because you can't try running MUSCLE manually outside of Biopython (because you don't have the input file to supply). There can also be subtle cross platform issues (e.g. Windows versus Linux), and how you run your script can have an impact (e.g. at the command line, from IDLE or an IDE, or as a GUI script). These are all generic Python issues though, and not specific to Biopython.

If you find working directly with `subprocess` like this scary, there is an alternative. If you execute the tool with `muscle_cline()` you can supply any standard input as a big string, `muscle_cline(stdin=...)`. So, provided your data isn't very big, you can prepare the FASTA input in memory as a string using `StringIO` (see Section 24.1):

```
>>> from Bio import SeqIO
>>> records = (r for r in SeqIO.parse("opuntia.fasta", "fasta") if len(r) < 900)
>>> from io import StringIO
>>> handle = StringIO()
>>> SeqIO.write(records, handle, "fasta")
6
>>> data = handle.getvalue()
```

You can then run the tool and parse the alignment as follows:

```
>>> stdout, stderr = muscle_cline(stdin=data)
>>> from Bio import AlignIO
>>> align = AlignIO.read(StringIO(stdout), "clustal")
>>> print(align)
SingleLetterAlphabet() alignment with 6 rows and 900 columns
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.1|AF19166
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.1|AF19166
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.1|AF19166
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.1|AF19166
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.1|AF19165
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.1|AF19165
```

You might find this easier, but it does require more memory (RAM) for the strings used for the input FASTA and output Clustal formatted data.

6.4.5 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.

Let's start by creating a complete `needle` command line object in one go:

```
>>> from Bio.Emboss.Applications import NeedleCommandline
>>> needle_cline = NeedleCommandline(asequence="alpha.faa", bsequence="beta.faa",
...                                  gapopen=10, gapextend=0.5, outfile="needle.txt")
>>> print(needle_cline)
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 tell Biopython the full path to the tool, for example:

```
>>> from Bio.Emboss.Applications import NeedleCommandline
>>> needle_cline = NeedleCommandline(r"C:\EMBOSS\needle.exe",
...                                  asequence="alpha.faa", bsequence="beta.faa",
...                                  gapopen=10, gapextend=0.5, outfile="needle.txt")
```

Remember in Python that for a default string `\n` or `\t` means a new line or a tab – which is why we're put a letter "r" at the start for a raw string.

At this point it might help to try running the EMBOSS tools yourself by hand at the command line, to familiarise yourself the other options and compare them to the Biopython help text:

```
>>> from Bio.Emboss.Applications import NeedleCommandline
>>> help(NeedleCommandline)
```

Note that you can also specify (or change or look at) the settings like this:

```
>>> from Bio.Emboss.Applications import NeedleCommandline
>>> needle_cline = NeedleCommandline()
>>> needle_cline.asequence="alpha.faa"
>>> needle_cline.bsequence="beta.faa"
>>> needle_cline.gapopen=10
>>> needle_cline.gapextend=0.5
>>> needle_cline.outfile="needle.txt"
>>> print(needle_cline)
needle -outfile=needle.txt -asequence=alpha.faa -bsequence=beta.faa -gapopen=10 -gapextend=0.5
>>> print(needle_cline.outfile)
needle.txt
```

Next we want to use Python to run this command for us. As explained above, for full control, we recommend you use the built in Python `subprocess` module, but for simple usage the wrapper object usually suffices:

```
>>> stdout, stderr = needle_cline()
>>> print(stdout + 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)
SingleLetterAlphabet() 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 `stdout=True` rather than the `outfile` argument), and also to read *one* of the one of the inputs from stdin (e.g. `asequence="stdin"`, much like in the MUSCLE example in the section above).

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.

6.5 Pairwise sequence alignment

Pairwise sequence alignment is the process of aligning two sequences to each other by optimizing the similarity score between them. Biopython includes two built-in pairwise aligners: the 'old' `Bio.pairwise2` module and the new `PairwiseAligner` class within the `Bio.Align` module (since Biopython version 1.72). Both can perform global and local alignments and offer numerous options to change the alignment parameters. Although `pairwise2` has gained some speed and memory enhancements recently, the new `PairwiseAligner` is much faster; so if you need to make many alignments with larger sequences, the latter would be the tool to choose. `pairwise2`, on the contrary, is also able to align lists, which can be useful if your sequences do not consist of single characters only.

Given that the parameters and sequences are the same, both aligners will return the same alignments and alignment score (if the number of alignments is too high they may return different subsets of all valid alignments).

6.5.1 pairwise2

`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 more options see Biopython's API). 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]) # doctest:+ELLIPSIS
Alignment(seqA='MV-LSPADKTNV---K-A--A-WGKVGAHAG...YR-', seqB='MVHL-----T--PEEKSAVTALWGKV----...Y-H', sc
```

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])) # doctest:+ELLIPSIS
MV-LSPADKTNV---K-A--A-WGKVGAHAG---EY-GA-EALE-RMFLSF----PTTK-TY--F...YR-
||  |     |     | |  | ||||        |  |  |||  |  |      |    |   |...|
MVHL-----T--PEEKSAVTALWGKV-----NVDE-VG-GEAL-GR--L--LVVYP---WT-QRF...Y-H
  Score=72
<BLANKLINE>
```

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.SubsMat.MatrixInfo import 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])) # doctest:+ELLIPSIS
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.SubsMat.MatrixInfo import blosum62
>>> alignments = pairwise2.align.localds("LSPADKTNVKAA", "PEEKSAV", blosum62, -10, -1)
>>> print(pairwise2.format_alignment(*alignments[0]))
3 PADKTNV
  |..|..|
1 PEEKSAV
  Score=16
<BLANKLINE>
```

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.SubsMat.MatrixInfo import blosum62
>>> alignments = pairwise2.align.localds("LSPADKTNVKAA", "PEEKSAV", blosum62, -10, -1)
>>> print(pairwise2.format_alignment(*alignments[0], full_sequences=True))
LSPADKTNVKAA
  |..|..|
--PEEKSAV---
  Score=16
<BLANKLINE>
```

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 serin/aspartate (S/D) and lysin/asparagin (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.SubsMat.MatrixInfo import blosum62
>>> alignments = pairwise2.align.localds("LSSPADKTNVKKAA", "DDPEEKSAVNN", blosum62, -10, -1)
>>> print(pairwise2.format_alignment(*alignments[0]))
```

```
4 PADKTNV
  |..|..|
3 PEEKSAV
  Score=16
<BLANKLINE>
```

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
<BLANKLINE>
```

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 modules documentation in Biopython's API.

6.5.2 PairwiseAligner

The new `Bio.Align.PairwiseAligner` implements the Needleman-Wunsch, Smith-Waterman, Gotoh (three-state), and Waterman-Smith-Beyer global and local pairwise alignment algorithms. We refer to Durbin *et* al. [16] for in-depth information on sequence alignment algorithms.

6.5.2.1 Basic usage

To generate pairwise alignments, first create a `PairwiseAligner` object:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
```

The `PairwiseAligner` object `aligner` (see Section 6.5.2.2) stores the alignment parameters to be used for the pairwise alignments.

These attributes can be set in the constructor of the object or after the object is made.

```
>>> aligner = Align.PairwiseAligner(match_score=1.0)
```

Or, equivalently:

```
>>> aligner.match_score = 1.0
```

Use the `aligner.score` method to calculate the alignment score between two sequences:

```
>>> seq1 = "GAACT"
>>> seq2 = "GAT"
>>> score = aligner.score(seq1, seq2)
>>> score
3.0
```

To see the actual alignments, use the `aligner.align` method and iterate over the `PairwiseAlignment` objects returned:

```
>>> alignments = aligner.align(seq1, seq2)
>>> for alignment in alignments:
...     print(alignment)
...
GAACT
||--|
GA--T
<BLANKLINE>
GAACT
|-|-|
G-A-T
<BLANKLINE>
```

By default, a global pairwise alignment is performed, which finds the optimal alignment over the whole length of `seq1` and `seq2`. Instead, a local alignment will find the subsequence of `seq1` and `seq2` with the highest alignment score. Local alignments can be generated by setting `aligner.mode` to `"local"`:

```
>>> aligner.mode = 'local'
>>> seq1 = "AGAACTC"
>>> seq2 = "GAACT"
>>> score = aligner.score(seq1, seq2)
>>> score
5.0
>>> alignments = aligner.align(seq1, seq2)
>>> for alignment in alignments:
...     print(alignment)
...
AGAACTC
 |||||
 GAACT
<BLANKLINE>
```

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 [41] and disallow such extensions.

6.5.2.2 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

```
>>> print(aligner)
Pairwise sequence aligner with parameters
  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
<BLANKLINE>
```

See Sections 6.5.2.3, 6.5.2.4, and 6.5.2.5 below for the definition of these parameters. The attribute `mode` (described above in Section 6.5.2.1) can be set equal to `"global"` or `"local"` to specify global or local pairwise alignment, respectively.

Depending on the gap scoring parameters (see Sections 6.5.2.4 and 6.5.2.5) 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.

6.5.2.3 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 [26] 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 6.5.2.4 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, the character `X` is interpreted as an unknown character and gets a zero score in alignments, irrespective of the value of the match or mismatch score:

```
>>> score = aligner.score("ACGT","ACXT")
>>> 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 [26] 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()  #doctest: +ELLIPSIS
['BENNER22', 'BENNER6', 'BENNER74', 'BLOSUM45', 'BLOSUM50', 'BLOSUM62', ..., 'STR']
>>> matrix = substitution_matrices.load("BLOSUM62")
>>> print(matrix)  #doctest: +ELLIPSIS
#  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 substutition 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`.

6.5.2.4 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 6.1.

Table 6.1: 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
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_open_gap_score
target_extend_gap_score	target_internal_extend_gap_score, target_left_extend_gap_score, target_right_extend_gap_score
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_gap_score
query_extend_gap_score	query_internal_extend_gap_score, query_left_extend_gap_score, query_right_extend_gap_score
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

6.5.2.5 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)
...
AACTT
-|.||
-AATT
<BLANKLINE>
AACTT
|-.||
A-ATT
<BLANKLINE>
AACTT
||.-|
AAT-T
<BLANKLINE>
AACTT
||.|-
AATT-
<BLANKLINE>
```

6.5.2.6 Iterating over alignments

The `alignments` returned by `aligner.align` are a kind of immutable iterable objects (similar to `range`). While they appear similarto a `tuple` or `list` of `PairwiseAlignment` objects, they are different in the sense that each `PairwiseAlignment` 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 6.5.2.8 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])
AAA
-||
-AA
<BLANKLINE>
>>> print(alignments[0])
AAA
||-
AA-
<BLANKLINE>
```

- You can iterate over alignments, for example as in

```
>>> for alignment in alignments:
...     print(alignment)
...
```

Note that `alignments` can be reused, i.e. you can iterate over alignments multiple times:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> alignments = aligner.align("AAA", "AA")
>>> for alignment in alignments:
...     print(alignment)
...
AAA
||-
AA-
<BLANKLINE>
AAA
|-|
A-A
<BLANKLINE>
AAA
-||
-AA
<BLANKLINE>
>>> for alignment in alignments:
...     print(alignment)
...
AAA
||-
AA-
<BLANKLINE>
AAA
```

```
|-|
A-A
<BLANKLINE>
AAA
-||
-AA
<BLANKLINE>
```

You can also convert the `alignments` iterator 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
```

6.5.2.7 Alignment objects

The `aligner.align` method returns `PairwiseAlignment` objects, each representing one alignment between the two sequences.

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> seq1 = "GAACT"
>>> seq2 = "GAT"
>>> alignments = aligner.align(seq1, seq2)
>>> alignment = alignments[0]
>>> alignment # doctest: +SKIP
<Bio.Align.PairwiseAlignment object at 0x10204d250>
```

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'
```

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

```
>>> print(alignment)
GAACT
||--|
GA--T
<BLANKLINE>
```

You can also represent the alignment as a string in PSL (Pattern Space Layout, as generated by BLAT [29]) format:

```
>>> format(alignment, 'psl')
'3\t0\t0\t0\t0\t0\t1\t2\t+\tquery\t3\t0\t3\ttarget\t5\t0\t5\t2\t2,1,\t0,2,\t0,4,\n'
```

Use the `aligned` property to find the start and end indices of subsequences in the target and query sequence that were aligned to each other. Generally, 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)),
)
```

In the current example, 'alignment.aligned' returns two tuples of length 2:

```
>>> alignment.aligned
(((0, 2), (4, 5)), ((0, 2), (2, 3)))
```

while for the alternative alignment, two tuples of length 3 are returned:

```
>>> alignment = alignments[1]
>>> print(alignment)
GAACT
|-|-|
G-A-T
<BLANKLINE>
>>> alignment.aligned
(((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.mismatch_score = -10
>>> alignments = aligner.align("AAACAAA", "AAAGAAA")
>>> len(alignments)
2
>>> print(alignments[0])
AAAC-AAA
|||--|||
AAA-GAAA
<BLANKLINE>
>>> alignments[0].aligned
(((0, 3), (4, 7)), ((0, 3), (4, 7)))
>>> print(alignments[1])
AAA-CAAA
|||--|||
AAAG-AAA
<BLANKLINE>
>>> alignments[1].aligned
(((0, 3), (4, 7)), ((0, 3), (4, 7)))
```

The `aligned` property can be used to identify alignments that are identical to each other in terms of their aligned sequences.

6.5.2.8 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)  #doctest: +ELLIPSIS
MV-LS-PAD--KTN--VK-AA-WGKV-----GAHAGEYGAEALE-RMFLSF----P-TTKTY--FPHF--...
||-|--|----|----|--|--||||-----|---||--|--|--|--|------|-|------|--|--...
MVHL-TP--EEK--SAV-TA-LWGKVNVDEVG---GE--A--L-GR--L--LVVYPWT----QRF--FES...
```

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])  #doctest: +ELLIPSIS
MV-LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF-DLS-----HGSAQVKGHGKKV...
||-|.|..|..|.|.||||--...|.|.|||.|.....|.|...|..|-|||-----.|...||.|||||...
MVHLTPEEKSAVTALWGKV--NVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPKVKAHGKKV...
<BLANKLINE>
```

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)
LSPADKTNVKAA
  |..|..|
  PEEKSAV
<BLANKLINE>
>>> print(alignment.score)
16.0
```

6.5.2.9 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.

Generalized pairwise alignments using a substitution matrix and alphabet Schneider *et al.* [35] created a substitution matrix for aligning three-nucleotide codons (see below in section 6.6 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  #doctest: +ELLIPSIS
('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  ---
<BLANKLINE>
>>> print(alignments[1])
AAT CTG TTT TTT
|||  ...  ---  |||
AAT TTA ---  TTT
<BLANKLINE>
```

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
<BLANKLINE>
>>> print(m['CTC', 'TTA'])
6.5
```

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']
```

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
<BLANKLINE>
>>> print(alignments[1])
Asn Leu Leu Phe
|||  ---  |||  |||
Asn  ---  Leu Phe
<BLANKLINE>
>>> print(alignments.score)
18.0
```

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
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> s1 = numpy.array([2, 10, 10, 13], numpy.int32)
>>> s2 = numpy.array([2, 10, 13], numpy.int32)
>>> aligner.match = +6
>>> aligner.mismatch = -1
>>> alignments = aligner.align(s1, s2)
>>> print(len(alignments))
2
>>> print(alignments[0])
2 10 10 13
|  ||  --  ||
2 10  --  13
<BLANKLINE>
>>> print(alignments[1])
2 10 10 13
|  --  ||  ||
2  --  10 13
```

```
<BLANKLINE>
>>> print(alignments.score)
18.0
```

Note that the indices should consist of 32-bit integers, as specified in this example by `numpy.int32`.

Negative indices are interpreted as unknown letters, and receive a zero score:

```
>>> s2 = numpy.array([2, -5, 13], numpy.int32)
>>> aligner.gap_score = -3
>>> alignments = aligner.align(s1, s2)
>>> print(len(alignments))
2
>>> print(alignments[0])
2 10 10 13
| .. -- ||
2 -5 -- 13
<BLANKLINE>
>>> print(alignments[1])
2 10 10 13
| -- .. ||
2 -- -5 13
<BLANKLINE>
>>> print(alignments.score)
9.0
```

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
>>> aligner = Align.PairwiseAligner()
>>> m = numpy.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 = numpy.array([0, 2, 3, 4], numpy.int32)
>>> s2 = numpy.array([0, 3, 2, 1], numpy.int32)
>>> alignments = aligner.align(s1, s2)
>>> print(len(alignments))
2
>>> print(alignments[0])
0 - 2 3 4
| - | . -
```

```
0 3 2 1 -
<BLANKLINE>
>>> print(alignments[1])
0 - 2 3 4
| - | - .
0 3 2 - 1
<BLANKLINE>
>>> print(alignments.score)
2.0
```

6.6 Substitution matrices

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.

Creating an Array object

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
<BLANKLINE>
```

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
<BLANKLINE>
>>> 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):
    ...
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
<BLANKLINE>
```

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
<BLANKLINE>
>>> 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
>>> x = Array("ACGT")
>>> x['C'] = 5

>>> x
Array([0., 5., 0., 0.],
      alphabet='ACGT')
>>> a = numpy.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 a `Array` is usually a string, you may also use a tuple of (immutable) objects. This is used for example for a codon substitution matrix, where the keys are not individual nucleotides or amino acids but instead three-nucleotide codons.

Calculating a substitution matrix from a 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` and initialize it with the default scores used by `blastn`:

```
>>> from Bio.Align import PairwiseAligner
>>> aligner = PairwiseAligner()
>>> aligner.mode = 'local'
>>> aligner.match_score = 2
>>> aligner.mismatch_score = -3
>>> aligner.open_gap_score = -7
>>> aligner.extend_gap_score = -2
```

Next, we read in the 16S ribosomal RNA gene sequence of *Escherichia coli* and *Bacillus subtilis* (provided in `Tests/scoring_matrices/ecoli.fa` and `Tests/scoring_matrices/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.seq, sequence2.seq)
```

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]
>>> from Bio.Align.substitution_matrices import Array
>>> frequency = Array("ACGT", dims=2)
>>> for (start1, end1), (start2, end2) in zip(*alignment.aligned):
...     seq1 = sequence1[start1:end1]
...     seq2 = sequence2[start2:end2]
...     for c1, c2 in zip(seq1, seq2):
...         frequency[c1, c2] += 1
...
>>> print(frequency)
      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
<BLANKLINE>
```

We normalize against the total number to find the probability of each substitution, and create a symmetric matrix:

```
>>> import numpy
>>> probabilities = frequency / numpy.sum(frequency)
>>> probabilities = (probabilities + probabilities.transpose()) / 2.0
>>> print(format(probabilities, "%.4f"))
       A      C      G      T
A 0.2026 0.0112 0.0224 0.0142
C 0.0112 0.1848 0.0162 0.0215
G 0.0224 0.0162 0.2647 0.0132
T 0.0142 0.0215 0.0132 0.1505
<BLANKLINE>
```

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 = numpy.sum(probabilities, 0)
>>> print(format(background, "%.4f"))
A 0.2505
C 0.2337
G 0.3165
T 0.1993
<BLANKLINE>
```

The number of substitutions expected at random is simply the product of the background distribution with itself:

```
>>> expected = numpy.dot(background[:,None], background[None, :])
>>> print(format(expected, "%.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
<BLANKLINE>
```

The scoring matrix can then be calculated as the logarithm of the odds-ratio of the observed and the expected probabilities:

```
>>> oddsratios = probabilities / expected
>>> scoring_matrix = numpy.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
<BLANKLINE>
```

The matrix can be used to set the substitution matrix for the pairwise aligner:

```
>>> aligner.substitution_matrix = scoring_matrix
```

A `ValueError` is triggered if the `Array` objects appearing in a mathematical operation have different alphabets:

```
>>> from Bio.Align.substitution_matrices import Array
>>> d = Array("ACGT")
>>> r = Array("ACGU")
>>> d + r
Traceback (most recent call last):
    ...
ValueError: alphabets are inconsistent
```

Reading `Array` object 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
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)  #doctest: +ELLIPSIS
chr1 248956422.0
chr2 242193529.0
chr3 198295559.0
chr4 190214555.0
...
chrUn_KI270423v1       981.0
chrUn_KI270392v1       971.0
chrUn_KI270394v1       970.0
<BLANKLINE>
```

Use `dtype=int` to read the values as integers:

```
>>> with open("hg38.chrom.sizes") as handle:
...     table = substitution_matrices.read(handle, int)
...
>>> print(table)  #doctest: +ELLIPSIS
chr1 248956422
chr2 242193529
chr3 198295559
chr4 190214555
...
chrUn_KI270423v1       981
chrUn_KI270392v1       971
chrUn_KI270394v1       970
<BLANKLINE>
```

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 [26] 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
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 = format(matrix)
>>> print(text)  #doctest: +ELLIPSIS
#  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.

Loading predefined substitution matrices

Biopython contains a large set of substitution matrices defined in the literature, including BLOSUM (Blocks Substitution Matrix) [24] and PAM (Point Accepted Mutation) matrices [14]. These matrices are available as flat files in the `Bio/Align/scoring_matrices/data` directory, and can be loaded into Python using the `load` function in the `scoring_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()  #doctest: +ELLIPSIS
['BENNER22', 'BENNER6', 'BENNER74', 'BLOSUM45', 'BLOSUM50', ..., 'STR']
```

Note that the substitution matrix provided by Schneider *et al.* [35] uses an alphabet consisting of three-nucleotide codons:

```
>>> m = substitution_matrices.load("SCHNEIDER")
>>> m.alphabet  #doctest: +ELLIPSIS
('AAA', 'AAC', 'AAG', 'AAT', 'ACA', 'ACC', 'ACG', 'ACT', ..., 'TTG', 'TTT')
```

Chapter 7

BLAST

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 categorised 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 8 describes `Bio.SearchIO`, an *experimental* module in Biopython. 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, until that is declared stable, for production code please continue to use the `Bio.Blast` module for dealing with NCBI BLAST.

7.1 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 ftp://ftp.ncbi.nlm.nih.gov/pub/factsheets/HowTo_BLASTGuide.pdf.
- 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 take 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 7.3 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 5).

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 7.3), 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 7.3) 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 7.3 below). You may want to jump ahead to that now ....

7.2 Running BLAST locally

7.2.1 Introduction

Running BLAST locally (as opposed to over the internet, see Section 7.1) 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.

7.2.2 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 6.4 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 Biopython we can use the NCBI BLASTX wrapper from the `Bio.Blast.Applications` module to build the command line string, and run it:

```
>>> from Bio.Blast.Applications import NcbiblastxCommandline
>>> help(NcbiblastxCommandline)
...
>>> blastx_cline = NcbiblastxCommandline(query="opuntia.fasta", db="nr", evalue=0.001,
...                                      outfmt=5, out="opuntia.xml")
>>> blastx_cline
NcbiblastxCommandline(cmd='blastx', out='opuntia.xml', outfmt=5, query='opuntia.fasta',
db='nr', evalue=0.001)
>>> print(blastx_cline)
blastx -out opuntia.xml -outfmt 5 -query opuntia.fasta -db nr -evalue 0.001
>>> stdout, stderr = blastx_cline()
```

In this example there shouldn't be any output from BLASTX to the terminal, so stdout and stderr should be empty. 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 7.3.

7.2.3 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.

7.3 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 deprectaed 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 7.1.
- You can use Biopython to run BLAST locally, as described in section 7.2.

- 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 24.1).

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.AlignIO` (see Chapters 5 and 6), 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:
...     # 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.

7.4 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.

Here are my attempts at UML class diagrams for the `Blast` and `PSIBlast` record classes. If you are good at UML and see mistakes/improvements that can be made, please let me know. The Blast class diagram is shown in Figure 7.1.

The PSIBlast record object is similar, but has support for the rounds that are used in the iteration steps of PSIBlast. The class diagram for PSIBlast is shown in Figure 7.2.

7.5 Dealing with PSI-BLAST

You can run the standalone version of PSI-BLAST (the legacy NCBI command line tool `blastpgp`, or its replacement `psiblast`) using the wrappers in `Bio.Blast.Applications` 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.

7.6 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) using the wrappers in `Bio.Blast.Applications` 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.

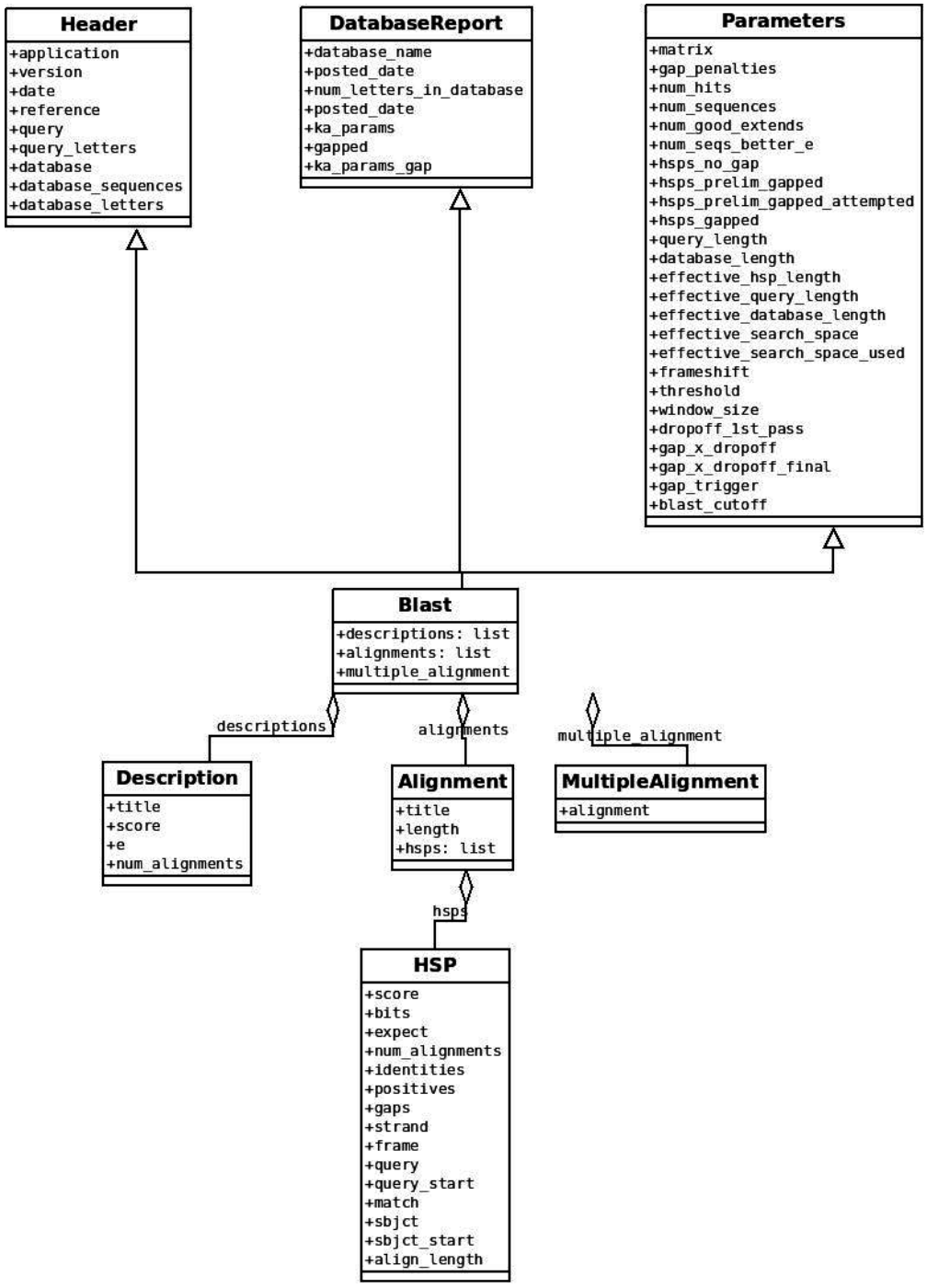


Figure 7.1: Class diagram for the Blast Record class representing all of the info in a BLAST report

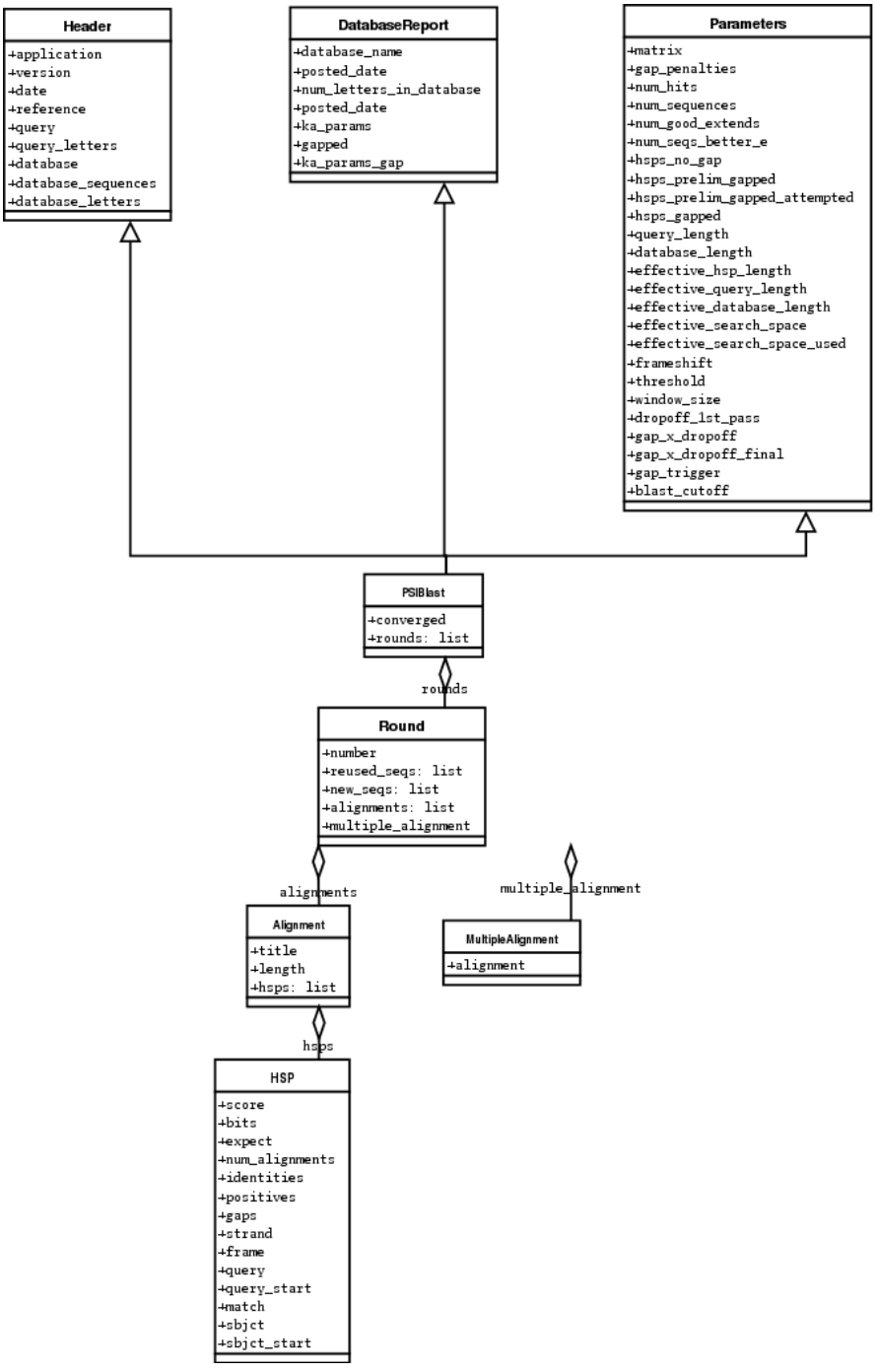


Figure 7.2: Class diagram for the PSIBlast Record class.

Chapter 8

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.

8.1 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`.

8.1.1 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: ----  -----  ------------------------------------------------------------
             #  # 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. Here are the ones for BLAST and for BLAT.

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)
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):
...     hit.id = hit.id.split("|")[3]   # renames "gi|301171322|ref|NR_035857.1|" to "NR_035857.1"
...     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.

8.1.2 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]
        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]
          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.

8.1.3 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 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', DNAAlphabet()), id='4
>>> blast_hsp.hit
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG', DNAAlphabet()), id='g
```

They are `SeqRecord` objects you saw earlier in Section 4! 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)
DNAAlphabet() 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
             ---  --------------  ----------------------  ----------------------
               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 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.

8.1.4 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, alphabets, 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', DNAAlphabet()), id='g
```

8.2 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.

8.3 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)
>>> 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
```

8.4 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 5.4.2. 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|"]
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.

8.5 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 9

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.

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 9.13.

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.

9.1 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 9.16). 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 20.5).

9.2 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
>>> handle = Entrez.einfo()
>>> result = handle.read()
>>> handle.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>
        <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
>>> handle = Entrez.einfo()
>>> record = Entrez.read(handle)
```

Now `record` is a dictionary with exactly one key:

```
>>> record.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
>>> handle = Entrez.einfo(db="pubmed")
>>> record = Entrez.read(handle)
>>> 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)
...
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.

9.3 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
>>> handle = Entrez.esearch(db="pubmed", term="biopython[title]", retmax="40" )
>>> record = Entrez.read(handle)
>>> "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 9.6).

You can also use ESearch to search GenBank. Here we'll do a quick search for the *matK* gene in *Cypripedioideae* orchids (see Section 9.2 about EInfo for one way to find out which fields you can search in each Entrez database):

```
>>> handle = Entrez.esearch(db="nucleotide", term="Cypripedioideae[Orgn] AND matK[Gene]", idtype="acc")
>>> record = Entrez.read(handle)
>>> 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 9.6 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:

```
>>> handle = Entrez.esearch(db="nlmcatalog", term="computational[Journal]", retmax="20")
>>> record = Entrez.read(handle)
>>> 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.

9.4 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
>>> 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 9.16 shows how to use the history feature.

9.5 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
>>> handle = Entrez.esummary(db="nlmcatalog", id="101660833")
>>> record = Entrez.read(handle)
>>> 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.
```

9.6 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 5.3.1 and 9.6). 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
>>> handle = Entrez.efetch(db="nucleotide", id="EU490707", rettype="gb", retmode="text")
>>> print(handle.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;
            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 tttttttctt 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
      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
//
<BLANKLINE>
```

```
<BLANKLINE>
```

Please be aware that as of October 2016 GI identifiers are discontinued in favour 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 5), 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
>>> handle = Entrez.efetch(db="nucleotide", id="EU490707", rettype="gb", retmode="text")
>>> record = SeqIO.read(handle, "genbank")
>>> handle.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
>>> print(repr(record.seq))
Seq('ATTTTTTACGAACCTGTGGAAATTTTTGGTTATGACAATAAATCTAGTTTAGTA...GAA', IUPACAmbiguousDNA())
```

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
filename = "EU490707.gbk"
if not os.path.isfile(filename):
    # Downloading...
    net_handle = Entrez.efetch(
        db="nucleotide", id="EU490707", rettype="gb", retmode="text"
    )
    out_handle = open(filename, "w")
    out_handle.write(net_handle.read())
    out_handle.close()
    net_handle.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
>>> handle = Entrez.efetch(db="nucleotide", id="EU490707", retmode="xml")
>>> record = Entrez.read(handle)
>>> handle.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 9.13.

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 9.16.

9.7 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'
>>> 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 162
pubmed pubmed_pubmed_alsoviewed 3
pubmed pubmed_pubmed_citedin 430
pubmed pubmed_pubmed_combined 6
pubmed pubmed_pubmed_five 6
pubmed pubmed_pubmed_refs 17
pubmed pubmed_pubmed_reviews 7
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 9.16.3).

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.

9.8 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 9.15.2 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
>>> handle = Entrez.egquery(term="biopython")
>>> record = Entrez.read(handle)
>>> for row in record["eGQueryResult"]:
```

```
...     print(row["DbName"], row["Count"])
...
pubmed 6
pmc 62
journals 0
...
```

See the EGQuery help page for more information.

9.9 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
>>> handle = Entrez.espell(term="biopythooon")
>>> record = Entrez.read(handle)
>>> 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.

9.10 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
>>> handle = open("Homo_sapiens.xml", "b")
>>> records = Entrez.parse(handle)
```

```
>>> 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
...
```

9.11 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(handle, 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`.

9.12 Handling errors

Three things can go wrong when parsing an XML file:

- The file may not be an XML file to begin with;

- The file may end prematurely or otherwise be corrupted;
- The file may be correct XML, but contain items that are not represented in the associated DTD.

The first case occurs if, for example, you try to parse a Fasta file as if it were an XML file:

```
>>> from Bio import Entrez
>>> handle = open("NC_005816.fna", "rb") # a Fasta file
>>> record = Entrez.read(handle)
Traceback (most recent call last):
  ...
Bio.Entrez.Parser.NotXMLError: Failed to parse the XML data (syntax error: line 1, column 0). Please mal
```

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.

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.ncbi.nlm.nih.gov/e
<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(handle)
Traceback (most recent call last):
  ...
Bio.Entrez.Parser.CorruptedXMLError: Failed to parse the XML data (no element found: line 16, column 0)
```

Note that the error message tells you at what point in the XML file the error was detected.

The third type of error occurs if the XML file contains tags that do not have a description in the corresponding DTD file. This is an example of such an XML file:

```
<?xml version="1.0"?>
<!DOCTYPE eInfoResult PUBLIC "-//NLM//DTD eInfoResult, 11 May 2002//EN" "https://www.ncbi.nlm.nih.gov/e
<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>
...
        </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
>>> handle = open("einfo3.xml", "rb")
>>> record = Entrez.read(handle)
Traceback (most recent call last):
  ...
Bio.Entrez.Parser.ValidationError: Failed to find tag 'DocsumList' in the DTD. To skip all tags that ar
```

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
>>> handle = open("einfo3.xml", "rb")
>>> record = Entrez.read(handle, validate=False)
>>> handle.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`.

9.13 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 5.3.1 and 9.6). For the literature databases, Biopython contains a parser for the `MEDLINE` format used in PubMed.

9.13.1 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 handle:
...     record = Medline.read(handle)
...
```

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 customised 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 handle:
...     for record in Medline.parse(handle):
...         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
>>> handle = Entrez.esearch(db="pubmed", term="biopython")
>>> record = Entrez.read(handle)
>>> record["IdList"]
['19304878', '18606172', '16403221', '16377612', '14871861', '14630660', '12230038']
```

We now use `Bio.Entrez.efetch` to download these Medline records:

```
>>> idlist = record["IdList"]
>>> handle = 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(handle)
>>> 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:

```
>>> handle = Entrez.efetch(db="pubmed", id=idlist, rettype="medline", retmode="xml")
>>> records = Entrez.read(handle)
>>> 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 9.16.

9.13.2 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
>>> handle = open("GSE16.txt")
>>> records = Geo.parse(handle)
>>> 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
>>> handle = Entrez.esearch(db="gds", term="GSE16")
>>> record = Entrez.read(handle)
>>> handle.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).

9.13.3 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
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=44
...
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)
```

9.14 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.

9.15 Examples

9.15.1 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 2.3 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
>>> handle = Entrez.egquery(term="orchid")
>>> record = Entrez.read(handle)
>>> 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
>>> handle = Entrez.esearch(db="pubmed", term="orchid", retmax=463)
>>> record = Entrez.read(handle)
>>> handle.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
>>> handle = Entrez.efetch(db="pubmed", id=idlist, rettype="medline",
...                        retmode="text")
>>> records = Medline.parse(handle)
```

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 9.16.

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.

9.15.2 Searching, downloading, and parsing Entrez Nucleotide records

Here we'll show a simple example of performing a remote Entrez query. In section 2.3 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
>>> handle = Entrez.egquery(term="Cypripedioideae")
>>> record = Entrez.read(handle)
>>> 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
>>> handle = Entrez.esearch(db="nucleotide", term="Cypripedioideae", retmax=814, idtype="acc")
>>> record = Entrez.read(handle)
>>> handle.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 9.16.

```
>>> idlist = ",".join(record["IdList"][:5])
>>> print(idlist)
KX265015.1, KX265014.1, KX265013.1, KX265012.1, KX265011.1]
>>> handle = Entrez.efetch(db="nucleotide", id=idlist, retmode="xml")
>>> records = Entrez.read(handle)
>>> 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 9.16.

9.15.3 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 5.3.1. For simplicity, this example *does not* take advantage of the WebEnv history feature – see Section 9.16 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
>>> handle = Entrez.egquery(term="Opuntia AND rpl16")
>>> record = Entrez.read(handle)
>>> for row in record["eGQueryResult"]:
...     if row["DbName"]=="nuccore":
...         print(row["Count"])
```

```
...
9
```

Now we download the list of GenBank identifiers:

```
>>> handle = Entrez.esearch(db="nuccore", term="Opuntia AND rpl16")
>>> record = Entrez.read(handle)
>>> 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)
>>> handle = 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 handle and print out the result:

```
>>> text = handle.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 5):

```
>>> from Bio import SeqIO
>>> handle = Entrez.efetch(db="nuccore", id=gi_str, rettype="gb", retmode="text")
>>> records = SeqIO.parse(handle, "gb")
```

We can now step through the records and look at the information we are interested in:

```
>>> for record in records:
>>> ...     print("%s, length %i, with %i features"
>>> ...           % (record.name, len(record), len(record.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 9.1. 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 9.16.

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.

9.15.4 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
>>> handle = Entrez.esearch(db="Taxonomy", term="Cypripedioideae")
>>> record = Entrez.read(handle)
>>> record["IdList"]
['158330']
>>> record["IdList"][0]
'158330'
```

Now, we use `efetch` to download this entry in the Taxonomy database, and then parse it:

```
>>> handle = Entrez.efetch(db="Taxonomy", id="158330", retmode="xml")
>>> records = Entrez.read(handle)
```

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.

9.16 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).

9.16.1 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 9.15.3, 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
>>> search_handle = Entrez.esearch(db="nucleotide",term="Opuntia[orgn] and rpl16",
...                                usehistory="y", idtype="acc")
>>> search_results = Entrez.read(search_handle)
>>> search_handle.close()
```

When you get the XML output back, it will still include the usual search results.

```
>>> acc_list = search_results["IdList"]
>>> count = int(search_results["Count"])
>>> count == len(acc_list)
True
```

(Remember from Section 9.15.2 that the number of records retrieved will not necessarily be the same as the `Count`, especially if the argument `retmax` is used.)
However, 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
```

```
out_handle = 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))
    fetch_handle = Entrez.efetch(
        db="nucleotide",
        rettype="fasta",
        retmode="text",
        retstart=start,
        retmax=batch_size,
        webenv=webenv,
        query_key=query_key,
        idtype="acc",
    )
    data = fetch_handle.read()
    fetch_handle.close()
    out_handle.write(data)
out_handle.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.

9.16.2 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
out_handle = open("recent_orchid_papers.txt", "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))
    fetch_handle = Entrez.efetch(
        db="pubmed",
        rettype="medline",
        retmode="text",
        retstart=start,
        retmax=batch_size,
        webenv=search_results["WebEnv"],
        query_key=search_results["QueryKey"],
    )
```

```
    data = fetch_handle.read()
    fetch_handle.close()
    out_handle.write(data)
out_handle.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 9.13.1 above, you can then use `Bio.Medline` to parse the saved records.

9.16.3 Searching for citations

Back in Section 9.7 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 9.16).

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"]]
>>> 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 10

Swiss-Prot and ExPASy

10.1 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.

10.1.1 Parsing Swiss-Prot records

In Section 5.3.2, 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("myswissprotfile.dat")
  ```

- 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
  >>> from io import TextIOWrapper
  >>> url = "https://raw.githubusercontent.com/biopython/biopython/master/Tests/SwissProt/F2CXE6.txt"
  >>> handle = TextIOWrapper(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 10.5.1):

  ```
  >>> from Bio import ExPASy
  >>> handle = ExPASy.get_sprot_raw(myaccessionnumber)
  ```

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.

We can use `Bio.SeqIO` as described in Section 5.3.2 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
>>> for ref in record.references:
...     print("authors:", ref.authors)
...     print("title:", ref.title)
...         print(record.organism_classification)
...
authors: Matsumoto T., Tanaka T., Sakai H., Amano N., Kanamori H., Kurita K., Kikuta A., Kamiya K., Yam
title: Comprehensive sequence analysis of 24,783 barley full-length cDNAs derived from 12 clone librari
['Eukaryota', 'Viridiplantae', 'Streptophyta', 'Embryophyta', 'Tracheophyta', 'Spermatophyta', 'Magnoli
authors: Shibasaka M., Sasano S., Utsugi S., Katsuhara M.
title: Functional characterization of a novel plasma membrane intrinsic protein2 in barley.
['Eukaryota', 'Viridiplantae', 'Streptophyta', 'Embryophyta', 'Tracheophyta', 'Spermatophyta', 'Magnoli
authors: Shibasaka M., Katsuhara M., Sasano S.
title:
['Eukaryota', 'Viridiplantae', 'Streptophyta', 'Embryophyta', 'Tracheophyta', 'Spermatophyta', 'Magnoli
```

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 ExPAYs 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']
```

10.1.2 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;
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.
...
```

10.2 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.

10.3 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.

10.4 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"]
'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]
```

10.5 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.

10.5.1 Retrieving a Swiss-Prot record

Let's say we are looking at chalcone synthases for Orchids (see section 2.3 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)
```

10.5.2 Searching Swiss-Prot

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, right now you could use https://www.uniprot.org/ but we do not have a Python wrapper for searching this from a script. Perhaps you could contribute here?

10.5.3 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.

10.6 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
6
>>> len(result)
```

```
6
>>> result[0]
{'signature_ac': u'PS50948', 'level': u'0', 'stop': 98, 'sequence_ac': u'USERSEQ1', 'start': 16, 'score
>>> result[1]
{'start': 37, 'stop': 39, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00005'}
>>> result[2]
{'start': 45, 'stop': 48, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00006'}
>>> result[3]
{'start': 60, 'stop': 62, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00005'}
>>> result[4]
{'start': 80, 'stop': 83, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00004'}
>>> result[5]
{'start': 106, 'stop': 111, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00008'}
```

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
7
```

Chapter 11

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!).

11.1 Reading and writing crystal structure files

11.1.1 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"]
```

11.1.2 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)
...
```

11.1.3 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 11.7.1) 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)
...
```

11.1.4 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)
```

11.1.5 Reading files in the PDB XML format

That's not yet supported, but we are definitely planning to support that in the future (it's not a lot of work). Contact the Biopython developers via the mailing list if you need this.

11.1.6 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")
```

11.1.7 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:
...             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.

11.1.8 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")
```

11.1.9 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`.

11.2 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. 11.1. 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 11.7.1.

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"))
```

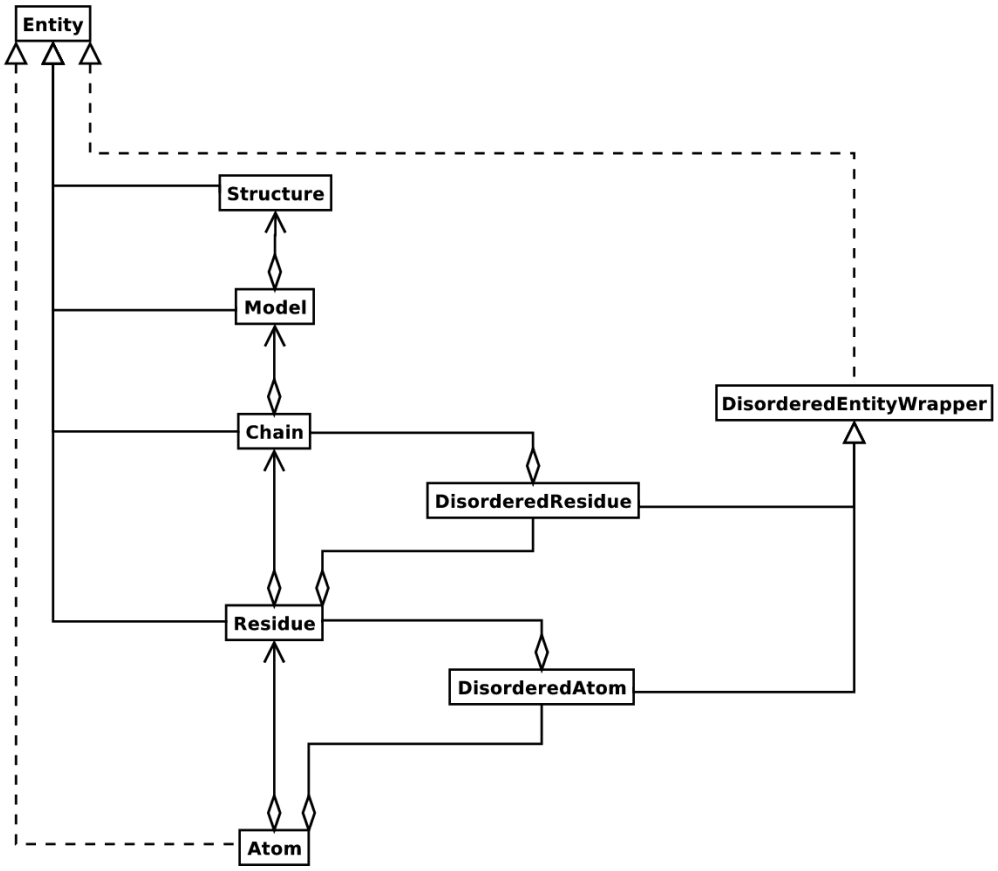


Figure 11.1: UML diagram of SMCRA architecture of the `Structure` class used to represent a macromolecular structure. Full lines with diamonds denote aggregation, full lines with arrows denote referencing, full lines with triangles denote inheritance and dashed lines with triangles denote interface realization.

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.

11.2.1 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.

11.2.2 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.

11.2.3 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.

11.2.4 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 11.4.1.
- 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 11.3.3.

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.

11.2.5 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 11.3.2.

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.

11.2.6 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"]
```

11.3 Disorder

Bio.PDB can handle both disordered atoms and point mutations (i.e. a Gly and an Ala residue in the same position).

11.3.1 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.

11.3.2 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. 11.1). 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"
```

11.3.3 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. 11.1).

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.

11.4 Hetero residues

11.4.1 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.

11.4.2 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, " ").

11.4.3 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, " ").

11.5 Navigating through a Structure object

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"]
```

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)
...
```

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.

Extract a hetero residue from a chain (e.g. a glucose (GLC) moiety with resseq 10)

```
>>> residue_id = ("H_GLC", 10, " ")
>>> residue = chain[residue_id]
```

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)
...
```

Print out the coordinates of all CA atoms in a structure with B factor greater than 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())
...
```

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)
...
```

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")
...
```

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.

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, and its alphabet is defined by a `ProteinAlphabet` object.

Example:

```
>>> seq = polypeptide.get_sequence()
>>> print(seq)
Seq('SNVVE...', <class Bio.Alphabet.ProteinAlphabet>)
```

11.6 Analyzing structures

11.6.1 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
```

11.6.2 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)
```

11.6.3 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)
```

11.6.4 Internal coordinates for standard residues

The `internal_coords` module is provided to facilitate working with canonical bond lengths, angles and torsion angles for a standard protein. `atom_to_internal_coordinates()` for `Structure`, `Model`, and `Chain` entities will extend `Chain` and `Residue` classes with `internal_coord` attributes referencing `IC_Chain` and `IC_Residue` classes respectively. `IC_Residue` provides `get_angle()` and `get_length()` to query the computed values with various specifiers and synonyms:

```
>>> model.atom_to_internal_coordinates()
>>> for r in model.get_residues():
...     if r.internal_coord:
...         print(
...             r,
...             r.internal_coord.get_angle("psi"),
...             r.internal_coord.get_angle("phi"),
...             r.internal_coord.get_angle("omega"),  # or "omg"
...             r.internal_coord.get_angle("chi2"),
...             r.internal_coord.get_angle("CB:CA:C"),
...             (r.internal_coord.get_length("-1C:0N")  # i-1 to i peptide bond
...               if r.internal_coord.rprev else "None")
...         )
```

Class	Attribute	Default	Effect
AtomKey	d2h	False	Convert D atoms to H if True
IC_Chain	MaxPeptideBond	1.4	Max C-N length w/o chain break; make large to link over missing residues for 3D models
IC_Residue	accept_atoms	mainchain, hydrogen atoms	override to remove some or all H's, D's
	accept_resnames	CYG, YCM, UNK	3-letter names for HETATMs to process, backbone only unless added to ic_data.py
	gly_Cbeta	False	override to generate Gly Cβ atoms based on database averages

Table 11.1: Control attributes in Bio.PDB.internal_coords.

Note that only angles, dihedral angles and residue configurations specified in `ic_data.py` are computed, however these data structures can be extended to add support for HETATMS as needed.

Corresponding `set_angle()` and `set_length()` routines are also provided, and the atom coordinates may be updated using `internal_to_atom_coordinates()`.

Missing atoms will cause problems for rebuilding structures from internal coordinates, however chain breaks and disordered residues and atoms as described above are handled correctly. `structure_rebuild_test(entity)` will compare a structure to a copy built from internal coordinates and return a report of success or failure as a dictionary.

An entity specification consisting of only internal coordinates (and optional positioning information) may be exported as a *.pic* file with `write_PIC()`. This format uses 3- and 4-tuples of `AtomKeys` to specify 3-atom hedra and 4-atom dihedra geometries. `AtomKeys` consist of up to six fields, capturing residue position, insertion code, residue name, atom name, altloc and occupancy. A *.pic* file includes sufficient information to regenerate the ATOM records of a *.pdb* file.

The `internal_to_atom_coords()` assembly algorithm is also implemented in OpenSCAD (http://www.openscad.org/), a language for describing 3D solid CAD models. `write_SCAD()` can thus generate a protein model suitable for rendering on a 3D printer, however the printing process is non-trivial and it is likely that you will want to modify the *.scad* file for your own purposes, just as you might select different rendering options for a 3D protein visualizer. A modified example is available at https://www.thingiverse.com/thing:3957471.

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 11.1,

11.6.5 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.

11.6.6 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 [19, Golub & Van Loan] 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.

11.6.7 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.

11.6.8 Calculating the Half Sphere Exposure

Half Sphere Exposure (HSE) is a new, 2D measure of solvent exposure [22]. 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$\mathring{A}$). 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])
```

11.6.9 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 11.2. Note that DSSP (the program, and thus by consequence the class) cannot handle multiple models!

The `DSSP` class can also be used to calculate the accessible surface area of a residue. But see also section 11.6.10.

Code	Secondary structure
H	α-helix
B	Isolated β-bridge residue
E	Strand
G	3-10 helix
I	II-helix
T	Turn
S	Bend
-	Other

Table 11.2: DSSP codes in Bio.PDB.

11.6.10 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.

11.7 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 [20, Hamelryck and Manderick, 2003]). 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.

11.7.1 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.

11.7.1.1 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).

11.7.1.2 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.

11.7.2 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.

11.7.2.1 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.

11.7.2.2 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.

11.7.3 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.

11.7.3.1 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.

11.7.3.2 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.

11.8 Accessing the Protein Data Bank

11.8.1 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.

11.8.2 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.

11.8.3 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.

11.9 General questions

11.9.1 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.

11.9.2 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.

11.9.3 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/

11.9.4 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/), and COLUMBA, a website that provides annotated protein structures (http://www.columba-db.de/). Bio.PDB has also been used to perform a large scale search for active sites similarities between protein structures in the PDB [21, Hamelryck, 2003], and to develop a new algorithm that identifies linear secondary structure elements [32, Majumdar *et al.*, 2005].

Judging from requests for features and information, Bio.PDB is also used by several LPCs (Large Pharmaceutical Companies :-).