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 – 20 December 2019 (Biopython 1.76)

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].

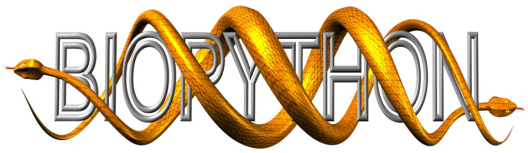
2. *How should I capitalize "Biopython"? Is "BioPython" OK?*
   The correct capitalization is "Biopython", not "BioPython" (even though that would have matched BioPerl, BioJava and BioRuby).

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?*
   This tutorial now uses the Python 3 style print *function*. As of Biopython 1.62, we support both Python 2 and Python 3. The most obvious language difference is the print *statement* in Python 2 became a print *function* in Python 3.

   For example, this will only work under Python 2:

   ```
   >>> print "Hello World!"
   Hello World!
   ```

   If you try that on Python 3 you'll get a `SyntaxError`. Under Python 3 you must write:

```
>>> print("Hello World!")
Hello World!
```

Surprisingly that will also work on Python 2 – but only for simple examples printing one thing. In general you need to add this magic line to the start of your Python scripts to use the print function under Python 2.6 and 2.7:

```
from __future__ import print_function
```

If you forget to add this magic import, under Python 2 you'll see extra brackets produced by trying to use the print function when Python 2 is interpreting it as a print statement and a tuple.

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 (and the print statement under Python 2):

```
>>> 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())
```

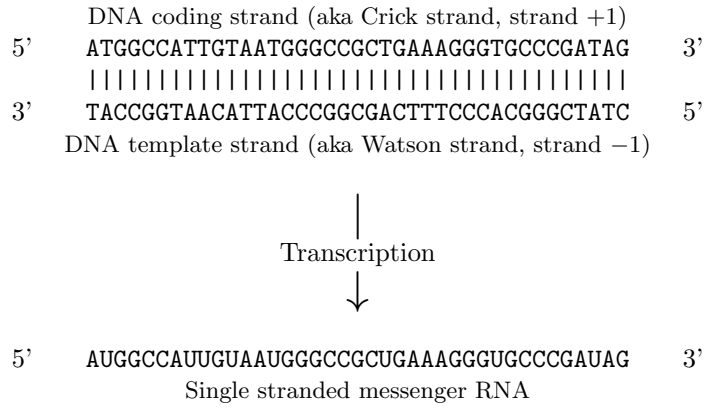
In all of these operations, the alphabet property is maintained. This is very useful in case you accidentally end up trying to do something weird like take the (reverse)complement of a protein sequence:

```
>>> 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 (sadly the function name isn't quite as consistent under Python 2):

```
>>> import bz2
>>> from Bio import SeqIO
>>> if hasattr(bz2, "open"):
...     handle = bz2.open("ls_orchid.gbk.bz2", "rt")  # Python 3
... else:
...     handle = bz2.BZ2File("ls_orchid.gbk.bz2", "r")  # Python 2
...
>>> with 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']
```

You can leave out the "list(...)" bit if you are still using Python 2. 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 StringIO 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))
...
```