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 – December 22, 2023 (Biopython 1.82)

Contents

Chapter 1

Introduction

1.1 What is Biopython?

Biopython is a collection of freely available Python (https://www.python.org) modules for computational molecular biology. Python is an object oriented, interpreted, flexible language that is widely used for scientific computing. Python is easy to learn, has a very clear syntax and can easily be extended with modules written in C, C++ or FORTRAN. Since its inception in 2000 [4], Biopython has been continuously developed and maintained by a large group of volunteers worldwide.

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

1.2 What can I find in the Biopython package

The main Biopython releases have lots of functionality, including:

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

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

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

1.3 Installing Biopython

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

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

1.4 Frequently Asked Questions (FAQ)

1. *How do I cite Biopython in a scientific publication?*
   Please cite our application note [5, 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: [4, Chapman and Chang, 2000];
   - For `Bio.PDB`: [16, Hamelryck and Manderick, 2003];
   - For `Bio.Cluster`: [10, De Hoon *et al.*, 2004];
   - For `Bio.Graphics.GenomeDiagram`: [35, Pritchard *et al.*, 2006];
   - For `Bio.Phylo` and `Bio.Phylo.PAML`: [45, Talevich *et al.*, 2012];
   - For the FASTQ file format as supported in Biopython, BioPerl, BioRuby, BioJava, and EMBOSS: [6, Cock *et al.*, 2010].

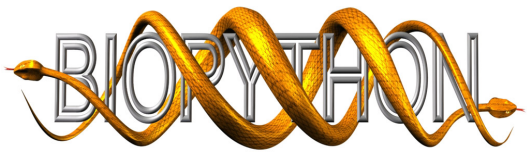
2. *How should I capitalize "Biopython"? Is "BioPython" OK?*
   The correct capitalization is "Biopython", not "BioPython" (even though that would have matched BioPerl, BioJava and BioRuby).

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

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

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

5. *Do you have a change-log listing what's new in each release?*
   See the file `NEWS.rst` included with the source code (originally called just `NEWS`), or read the latest NEWS file on GitHub.

6. *What is going wrong with my print commands?*
   As of Biopython 1.77, we only support Python 3, so this tutorial uses the Python 3 style print *function*.

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

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

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

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

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

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

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

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

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

   If you still need to support old versions of Biopython, use these explicit forms to avoid problems. See Section 3.10.

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

11. *Why won't the* `Bio.SeqIO` *and* `Bio.AlignIO` *functions* `parse`*,* `read` *and* `write` *take filenames? They insist on handles!*
    You need Biopython 1.54 or later, or just use handles explicitly (see Section 25.1). It is especially important to remember to close output handles explicitly after writing your data.

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 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/3/).

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

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

2.1 General overview of what Biopython provides

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

One thing to note about Biopython is that it often provides multiple ways of "doing the same thing." Things have improved in recent releases, but this can still be frustrating as in Python there should ideally be one right way to do something. However, this can also be a real benefit because it gives you lots of flexibility and control over the libraries. The tutorial helps to show you the common or easy ways to do things so that you can just make things work. To learn more about the alternative possibilities, look in the Cookbook (Chapter 22, this has some cools tricks and tips), and 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
```

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 12, extract data from Swiss-Prot from certain orchid proteins in Chapter 13, and work with ClustalW multiple sequence alignments of orchid proteins in Section 6.7.2.

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/nuccore/?term=Cypripedioideae) for everything mentioning the text Cypripedioideae (this is the subfamily of lady slipper orchids).

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

If you run the search today, you'll get hundreds of results! When following the tutorial, if you want to see the same list of genes, just download the two files above or copy them from `docs/examples/` in the Biopython source code. In Section 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 favorite text editor, you'll see that the file starts like this:

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

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

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

You should get something like this on your screen:

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

2.4.2 Simple GenBank parsing example

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

```
>>> from Bio import SeqIO
>>> for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank"):
```

```
...     print(seq_record.id)
...     print(repr(seq_record.seq))
...     print(len(seq_record))
...
```

This should give:

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

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

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

Biopython has a lot of parsers, and each has its own little special niches based on the sequence format it is parsing and all of that. Chapter 5 covers `Bio.SeqIO` in more detail, while Chapter 6 introduces `Bio.Align` for sequence alignments.

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

The next place to look for information about specific parsers and how to do cool things with them is in the Cookbook (Chapter 22 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 12.
- ExPASy – See Chapter 13.
- 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 22), 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.

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

3.1 Sequences act like strings

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

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

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

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

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

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

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

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

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

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

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

Also note that just like a normal Python string, the `Seq` object is in some ways "read-only". If you need to edit your sequence, for example simulating a point mutation, look at the Section 3.13 below which talks about the `MutableSeq` object.

3.2 Slicing a sequence

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

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

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

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

```
>>> my_seq[0::3]
Seq('GCTGTAGTAAG')
>>> my_seq[1::3]
Seq('AGGCATGCATC')
>>> my_seq[2::3]
Seq('TAGCTAAGAC')
```

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

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

3.3 Turning Seq objects into strings

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

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

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

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

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

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

3.4 Concatenating or adding sequences

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

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

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

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

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

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

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

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

3.5 Changing case

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

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

These are useful for doing case insensitive matching:

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

3.6 Nucleotide sequences and (reverse) complements

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

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

```
>>> my_seq
Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC')
>>> my_seq.complement()
Seq('CTAGCTACCCGGATATATCCTAGCTTTTAGCG')
>>> my_seq.reverse_complement()
Seq('GCGATTTTCGATCCTATATAGGCCCATCGATC')
```

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

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

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

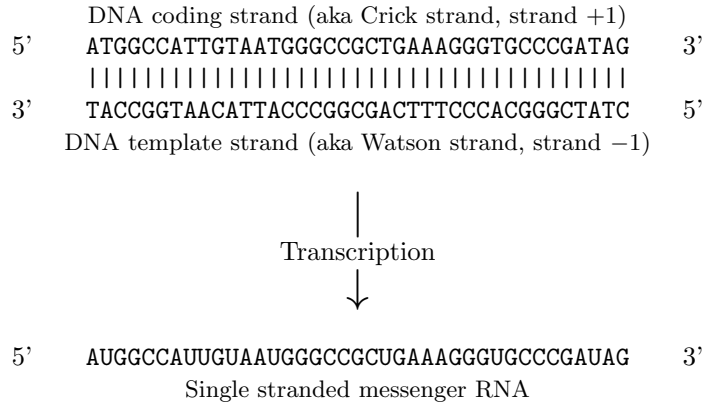
```
>>> from Bio.Seq import Seq
>>> protein_seq = Seq("EVRNAK")
>>> protein_seq.complement()
Seq('EBYNTM')
```

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

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

3.7 Transcription

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



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

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

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

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

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

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

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

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

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

Note: The `Seq` object's `transcribe` and `back_transcribe` methods were added in Biopython 1.49. For older releases you would have to use the `Bio.Seq` module's functions instead, see Section 3.15.

3.8 Translation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
>>> from Bio.Seq import Seq
>>> gene = Seq(
...     "GTGAAAAAGATGCAATCTATCGTACTCGCACTTTCCCTGGTTCTGGTCGCTCCCATGGCA"
...     "GCACAGGCTGCGGAAATTACGTTAGTCCCGTCAGTAAAATTACAGATAGGCGATCGTGAT"
```

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

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

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

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

The example in Section 22.1.3 combines the `Seq` object's translate method with `Bio.SeqIO` for sequence input/output.

3.9 Translation Tables

In the previous sections we talked about the `Seq` object translation method (and mentioned the equivalent function in the `Bio.Seq` module – see Section 3.15). 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'
```

3.10 Comparing Seq objects

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

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

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

As an extension to this, using sequence objects as keys in a Python dictionary is equivalent to using the sequence as a plain string for the key. See also Section 3.3.

3.11 Sequences with unknown sequence contents

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

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

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

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

3.12 Sequences with partially defined sequence contents

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

```
s hg38.chr7     117512683 36 + 159345973 TTGAAAACCTGAATGTGAGAGTCAGTCAAGGATAGT
s panTro4.chr7  119000876 36 + 161824586 TTGAAAACCTGAATGTGAGAGTCACTCAAGGATAGT
```

```
s rheMac3.chr3  156330991 36 + 198365852 CTGAAATCCTGAATGTGAGAGTCAATCAAGGATGGT
s mm10.chr6      18207101 36 + 149736546 CTGAAAACCTAAGTAGGAGAATCAACTAAGGATAAT
s rn5.chr4       42326848 36 + 248343840 CTGAAAACCTAAGTAGGAGAGACAGTTAAAGATAAT
s canFam3.chr14  56325207 36 +  60966679 TTGAAAAACTGATTATTAGAGTCAATTAAGGATAGT
s monDom5.chr8  173163865 36 + 312544902 TTAAGAAACTGGAAATGAGGGTTGAATGACAAACTT
```

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

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

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

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

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

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

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

3.13 MutableSeq objects

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

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

Observe what happens if you try to edit the sequence:

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

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

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

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

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

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

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

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

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

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

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

You can also get a string from a `MutableSeq` object just like from a `Seq` object (Section 3.3).

3.14 Finding subsequences

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

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

```
>>> seq.index(Seq("ATGGGCCGC"))
9
>>> seq.index(MutableSeq("ATGGGCCGC"))
9
```

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

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

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

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

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

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

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

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

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

3.15 Working with strings directly

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

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

```
'GCTGTTATGGGTCGTTGGAAGGGTGGTCGTGCTGCTGGTTAG'
>>> translate(my_string)
'AVMGRWKGGRAAG*'
```

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

Chapter 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 22.1.6) 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'),
id='gi|45478711|ref|NC_005816.1|', name='gi|45478711|ref|NC_005816.1|',
description='gi|45478711|ref|NC_005816.1| Yersinia pestis biovar Microtus str. 91001
plasmid pPCP1, complete sequence', dbxrefs=[])
```

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

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

Next, the identifiers and description:

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

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

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

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

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

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

4.2.3 SeqRecord objects from GenBank files

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

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

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

Again, we'll use `Bio.SeqIO` to read this file in, and the code is almost identical to that for used above for the FASTA file (see Chapter 5 for details):

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

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

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

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

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

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

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

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

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

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

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

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

We'll talk about `SeqFeature` objects next, in Section 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 SimpleLocation object

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.4 Comparison

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

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

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

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

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

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

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

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

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

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

Beware that comparing complex objects quickly gets complicated (see also Section 3.10).

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

which should give:

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

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'),
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 protein
entries of Yersinia pestis plasmid pPCP, e.g. gi| 16082683|,ref|NP_395230.1| (NC_003132)
, gi|1200166|emb|CAA90861.1| (Z54145 ) , gi|1488655| emb|CAA63439.1| (X92856) ,
gi|2996219|gb|AAC62543.1| (AF053945) , and gi|5763814|emb|CAB531 67.1| (AL109969)']
    Key: product, Value: ['pesticin immunity protein']
    Key: protein_id, Value: ['NP_995571.1']
    Key: transl_table, Value: ['11']
    Key: translation, Value: ['MGGGMISKLFCLALIFLSSSGLAEKNTYTAKDILQNLELNTFGNSLSHGIYGKQTTFK
QTEFTNIKSNTKKHIALINKDNSWMISLKILGIKRDEYTVCFEDFSLIRPPTYVAIHPLLIKKVKSGNFIVVKEIKKSIPGCTVYYH']
<BLANKLINE>
```

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

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

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

```
>>> print(sub_record.features[0])
type: gene
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 protein
entries of Yersinia pestis plasmid pPCP, e.g. gi| 16082683|,ref|NP_395230.1| (NC_003132)
, gi|1200166|emb|CAA90861.1| (Z54145 ) , gi|1488655| emb|CAA63439.1| (X92856) ,
gi|2996219|gb|AAC62543.1| (AF053945) , and gi|5763814|emb|CAB531 67.1| (AL109969)']
    Key: product, Value: ['pesticin immunity protein']
    Key: protein_id, Value: ['NP_995571.1']
    Key: transl_table, Value: ['11']
    Key: translation, Value: ['MGGGMISKLFCLALIFLSSSGLAEKNTYTAKDILQNLELNTFGNSLSHGIYGKQTTFK
QTEFTNIKSNTKKHIALINKDNSWMISLKILGIKRDEYTVCFEDFSLIRPPTYVAIHPLLIKKVKSGNFIVVKEIKKSIPGCTVYYH']
<BLANKLINE>
```

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

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

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

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

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

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

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

This illustrates the problem nicely though, our new sub-record is *not* the complete sequence of the plasmid, so the description is wrong! Let's fix this and then view the sub-record as a reduced GenBank file using the `format` method described above in Section 4.6:

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

See Sections 22.1.7 and 22.1.8 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'),
id='NC_005816.1', name='NC_005816', description='Yersinia pestis biovar Microtus str.
91001 plasmid pPCP1, complete sequence', dbxrefs=['Project:58037'])
>>> len(record)
9609
>>> len(record.features)
41
>>> record.dbxrefs
['Project:58037']
>>> record.annotations.keys()
dict_keys(['molecule_type', 'topology', 'data_file_division', 'date', 'accessions',
'sequence_version', 'gi', 'keywords', 'source', 'organism', 'taxonomy', 'references',
'comment'])
```

You can shift the origin like this:

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

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

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

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

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

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

4.9 Reverse-complementing SeqRecord objects

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

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

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

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

Consider this example record:

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

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

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

Chapter 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 25.1 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.

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 22.2 (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')
592
The first record
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC')
740
```

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

5.1.4 Extracting data

The `SeqRecord` object and its annotation structures are described more fully in Chapter 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')
```

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

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

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

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

or values:

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

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

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

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

or:

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

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

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

```
from Bio import SeqIO

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

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

```
from Bio import SeqIO

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

In either case, the result is:

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

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

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

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

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

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

The concise alternative using list comprehensions would be:

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

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

5.1.5 Modifying data

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

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

Note, if you want to change the way FASTA is output when written to a file (see Section 5.5), then you should modify both the `id` and `description` attributes. To ensure the correct behavior, it is best to include the `id` plus a space at the start of the desired `description`:

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

5.2 Parsing sequences from compressed files

In the previous section, we looked at parsing sequence data from a file. Instead of using a filename, you can give `Bio.SeqIO` a handle (see Section 25.1), and in this section we'll use handles to parse sequence from compressed files.

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

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

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

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

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

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

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

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

Similarly if we had a bzip2 compressed file:

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

There is a gzip (GNU Zip) variant called BGZF (Blocked GNU Zip Format), which can be treated like an ordinary gzip file for reading, but has advantages for random access later which we'll talk about later in Section 5.4.4.

5.3 Parsing sequences from the net

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

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

5.3.1 Parsing GenBank records from the net

Section 12.6 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 12 for more about the `Bio.Entrez` module, and make sure to read about the NCBI guidelines for using Entrez (Section 12.1).

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 13. As mentioned above, when you expect the handle to contain one and only one record, use the `Bio.SeqIO.read()` function:

```
from Bio import ExPASy
from Bio import SeqIO

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

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

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

5.4 Sequence files as Dictionaries

Looping over the iterator returned by `SeqIO.parse` once will exhaust the file. For self-indexed files, such as files in the twoBit format, the return value of `SeqIO.parse` can also be used as a dictionary, allowing

random access to the sequence contents. As in this case parsing is done on demand, the file must remain open as long as the sequence data is being accessed:

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

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

- `Bio.SeqIO.to_dict()` is the most flexible but also the most memory demanding option (see Section 5.4.1). This is basically a helper function to build a normal Python `dictionary` with each entry held as a `SeqRecord` object in memory, allowing you to modify the records.
- `Bio.SeqIO.index()` is a useful middle ground, acting like a read only dictionary and parsing sequences into `SeqRecord` objects on demand (see Section 5.4.2).
- `Bio.SeqIO.index_db()` also acts like a read only dictionary but stores the identifiers and file offsets in a file on disk (as an SQLite3 database), meaning it has very low memory requirements (see Section 5.4.3), but will be a little bit slower.

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

5.4.1 Sequence files as Dictionaries – In memory

The next thing that we'll do with our ubiquitous orchid files is to show how to index them and access them like a database using the Python `dictionary` data type (like a hash in Perl). This is very useful for moderately large files where you only need to access certain elements of the file, and makes for a nice quick 'n dirty database. For dealing with larger files where memory becomes a problem, see Section 5.4.2 below.

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

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

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

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

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

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

Under Python 3 the dictionary methods like ".keys()" and ".values()" are iterators rather than lists. If you really want to, you can even look at all the records at once:

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

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

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

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

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

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 recognize 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')
>>> 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 22.1.11). However, alignment formats like PHYLIP or Clustal are not supported. Finally as an optional argument you can supply a key function.

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

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

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 22.1.5 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) popularized by the BAM file format, samtools, and tabix.

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

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

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

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

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

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

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

or:

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

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

5.4.5 Discussion

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

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

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

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

- Faster to build the index (more noticeable in simple file formats)
- Slightly faster access as SeqRecord objects (but the difference is only really noticeable for simple to parse file formats).

- Can use any immutable Python object as the dictionary keys (e.g. a tuple of strings, or a frozen set) not just strings.
- Don't need to worry about the index database being out of date if the sequence file being indexed has changed.

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

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

5.5 Writing Sequence Files

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

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

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

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

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

rec3 = SeqRecord(
```

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

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

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

```
from Bio import SeqIO

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

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

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

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

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

5.5.1 Round trips

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

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

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

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

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

Occasionally preserving the original layout (with any quirks it may have) is important. See Section 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 22.1.9 and 22.1.10 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 optimizations and tricks.

5.5.3 Converting a file of sequences to their reverse complements

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

To start with, we'll use `Bio.SeqIO.parse()` to load some nucleotide sequences from a file, then print out their reverse complements using the `Seq` object's built in `.reverse_complement()` method (see Section 3.6):

```
>>> 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 22.1.3, translating each record in a FASTA file from nucleotides to amino acids.

5.5.4 Getting your SeqRecord objects as formatted strings

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

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

```
from Bio import SeqIO
from io import StringIO

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

This isn't entirely straightforward the first time you see it! On the bright side, for the special case where you would like a string containing a *single* record in a particular file format, use the the `SeqRecord` class' `format()` method (see Section 4.6).

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

```
from Bio import SeqIO

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

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

```
from Bio import SeqIO

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

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

5.6 Low level FASTA and FASTQ parsers

Working with the low-level `SimpleFastaParser` or `FastqGeneralIterator` is often more practical than `Bio.SeqIO.parse` when dealing with large high-throughput FASTA or FASTQ sequencing files where speed

matters. As noted in the introduction to this chapter, the file-format neutral `Bio.SeqIO` interface has the overhead of creating many objects even for simple formats like FASTA.

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

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

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

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

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

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

There are more examples of this in the Cookbook (Chapter 22), including how to output FASTQ efficiently from strings using this code snippet:

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

Chapter 6

Sequence alignments

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

Alignments may extend over the full length of each sequence, or may be limited to a subsection of each sequence. In Biopython, all sequence alignments are represented by an `Alignment` object, described in section 6.1. `Alignment` objects can be obtained by parsing the output of alignment software such as Clustal or BLAT (described in section 6.6. or by using Biopython's pairwise sequence aligner, which can align two sequences to each other (described in Chapter 7).

See Chapter 8 for a description of the older `MultipleSeqAlignment` class and the parsers in `Bio.AlignIO` that parse the output of sequence alignment software, generating `MultipleSeqAlignment` objects.

6.1 Alignment objects

The `Alignment` class is defined in `Bio.Align`. Usually you would get an `Alignment` object by parsing the output of alignment programs (section 6.6) or by running Biopython's pairwise aligner (Chapter 7). For the benefit of this section, however, we will create an `Alignment` object from scratch.

6.1.1 Creating an Alignment object from sequences and coordinates

Suppose you have three sequences:

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

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

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

These coordinates define the alignment for the following sequence segments:

- `SeqA[1:3]`, `SeqB[0:2]`, and `SeqC[0:2]` are aligned to each other;
- `SeqA[3:4]` and `SeqC[2:3]` are aligned to each other, with a gap of one nucleotide in `seqB`;
- `SeqA[4:7]`, `SeqB[2:5]`, and `SeqC[3:6]` are aligned to each other;

- `SeqA[7:9]` is not aligned to `seqB` or `seqC`.

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

Now we can create the `Alignment` object:

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

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

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

and an attribute `coordinates` with the alignment coordinates:

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

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

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

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

6.1.2 Creating an Alignment object from aligned sequences

If you start out with the aligned sequences, with dashes representing gaps, then you can calculate the coordinates using the `infer_coordinates` class method. This method is primarily employed in Biopython's alignment parsers (see Section 6.6), but it may be useful for other purposes. For example, you can construct the `Alignment` object from aligned sequences as follows:

```
>>> aligned_sequences = ["CGGTTTTT", "AG-TTT--", "AGGTTT--"]
>>> sequences = [aligned_sequence.replace("-", "")
...              for aligned_sequence in aligned_sequences]  # fmt: skip
...
>>> sequences
['CGGTTTTT', 'AGTTT', 'AGGTTT']
>>> coordinates = Alignment.infer_coordinates(aligned_sequences)
>>> coordinates
array([[0, 2, 3, 6, 8],
       [0, 2, 2, 5, 5],
       [0, 2, 3, 6, 6]])
```

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

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

Now we can create the `Alignment` object:

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

which identical to the `Alignment` object created above in section 6.1.1.

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

```
>>> ungapped_alignment = Alignment(["ACGTACGT", "AAGTACGT", "ACGTACCT"])
>>> ungapped_alignment  # doctest: +ELLIPSIS
<Alignment object (3 rows x 8 columns) at ...>
>>> ungapped_alignment.coordinates
array([[0, 8],
       [0, 8],
       [0, 8]])
>>> print(ungapped_alignment)
                  0 ACGTACGT 8
                  0 AAGTACGT 8
                  0 ACGTACCT 8
<BLANKLINE>
```

6.1.3 Common alignment attributes

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

- `sequences`: This is a list of the sequences aligned to each other. Depending on how the alignment was created, the sequences can have the following types:
  - plain Python string;
  - `Seq`;
  - `MutableSeq`;
  - `SeqRecord`;
  - `bytes`;
  - `bytearray`;
  - NumPy array with data type `numpy.int32`;

  - any other object with a contiguous buffer of format `"c"`, `"B"`, `"i"`, or `"I"`;
  - lists or tuples of objects defined in the `alphabet` attribute of the `PairwiseAligner` object that created the alignment (see section 7.11).

  For pairwise alignments (meaning an alignment of two sequences), the properties `target` and `query` are aliases for `sequences[0]` and `sequences[1]`, respectively.

- `coordinates`: A NumPy array of integers storing the sequence indices defining how the sequences are aligned to each other;
- `score`: The alignment score, as found by the parser in the alignment file, or as calculated by the `PairwiseAligner` (see section 7.1);
- `annotations`: A dictionary storing most other annotations associated with the alignment;
- `column_annotations`: A dictionary storing annotations that extend along the alignment and have the same length as the alignment, such as a consensus sequence (see section 6.7.2 for an example).

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

6.2 Slicing and indexing an alignment

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

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

```
>>> print(alignment)
                 1 CGGTTTTT 9
                 0 AG-TTT-- 5
                 0 AGGTTT-- 6
<BLANKLINE>
>>> alignment.length
8
```

To get the aligned sequence strings individually, use

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

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

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

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

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

To get specific columns in the alignment, use

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

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

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

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

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

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

Similarly, extracting the last 6 columns gives:

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

The column index can also be an iterable of integers:

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

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

6.3 Getting information about the alignment

6.3.1 Alignment shape

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

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

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

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

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

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

6.3.2 Comparing alignments

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

6.3.3 Finding the indices of aligned sequences

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

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

For example,

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

```
>>> pairwise_alignment.aligned
array([[[1, 3],
        [4, 7]],
<BLANKLINE>
       [[0, 2],
        [2, 5]]])
```

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

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

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

```
>>> print(alignment)
                  1 CGGTTTTT 9
                  0 AG-TTT-- 5
                  0 AGGTTT-- 6
<BLANKLINE>
>>> alignment.indices
array([[ 1,  2,  3,  4,  5,  6,  7,  8],
       [ 0,  1, -1,  2,  3,  4, -1, -1],
       [ 0,  1,  2,  3,  4,  5, -1, -1]])
```

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

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

6.3.4 Counting identities, mismatches, and gaps

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

```
>>> print(pairwise_alignment)
target            1 CGGTTTTT 9
                  0 .|-|||-- 8
query             0 AG-TTT-- 5
<BLANKLINE>
>>> pairwise_alignment.counts()
AlignmentCounts(gaps=3, identities=4, mismatches=1)
>>> print(alignment)
                  1 CGGTTTTT 9
                  0 AG-TTT-- 5
                  0 AGGTTT-- 6
<BLANKLINE>
>>> alignment.counts()
AlignmentCounts(gaps=8, identities=14, mismatches=2)
```

6.3.5 Letter frequencies

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

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

6.3.6 Substitutions

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

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

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

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

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

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

To get a symmetric matrix, use

```
>>> m += m.transpose()
>>> m /= 2.0
>>> print(m)
    A   C   G   T
A 1.0 1.0 0.0 0.0
C 1.0 0.0 0.0 0.0
G 0.0 0.0 4.0 0.0
T 0.0 0.0 0.0 9.0
<BLANKLINE>
>>> m["A", "C"]
1.0
>>> m["C", "A"]
1.0
```

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

6.3.7 Alignments as arrays

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

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

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

```
>>> align_array = np.array(alignment, dtype="U")
>>> align_array  # doctest: +NORMALIZE_WHITESPACE
array([['C', 'G', 'G', 'T', 'T', 'T', 'T', 'T'],
       ['A', 'G', '-', 'T', 'T', 'T', '-', '-'],
       ['A', 'G', 'G', 'T', 'T', 'T', '-', '-']], dtype='<U1')
```

Note that the `alignment` object and the NumPy array `align_array` are separate objects in memory - editing one will not update the other!

6.4 Operations on an alignment

6.4.1 Sorting an alignment

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

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

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

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

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

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

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

6.4.2 Reverse-complementing the alignment

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

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

```
                  2 --AAA-CT 7
<BLANKLINE>
>>> alignment[:, :4].sequences
['CCGGTTTTT', 'AGGTTT', 'AGTTTAA']
>>> print(alignment[:, :4])
                  1 CGGT 5
                  0 AGGT 4
                  0 AG-T 3
<BLANKLINE>
>>> rc_alignment = alignment[:, :4].reverse_complement()
>>> rc_alignment[:, :4].sequences
['AAAAACCGG', 'AAACCT', 'TTAAACT']
>>> print(rc_alignment[:, :4])
                  4 ACCG 8
                  2 ACCT 6
                  4 A-CT 7
<BLANKLINE>
```

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

6.4.3 Adding alignments

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

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

Now, let's look at these two alignments:

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

```
Alpha             0 GTT 3
Beta              0 -TT 2
Gamma             0 G-T 2
<BLANKLINE>
```

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

```
>>> combined = left + right
>>> print(combined)
Alpha             0 AAAACGTT 8
Beta              0 AAA-C-TT 6
Gamma             0 AAAAGG-T 7
<BLANKLINE>
```

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

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

The sequences are `SeqRecord` objects, which can be added together. Refer to Chapter 4 for details of how the annotation is handled. This example is a special case in that both original alignments shared the same names, meaning when the rows are added they also get the same name.

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

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

Similarly any common per-column-annotations are combined:

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

6.4.4 Mapping a pairwise sequence alignment

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

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

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

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

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

```
>>> alignment3 = alignment1.map(alignment2)
>>> print(alignment3)
target           11 CCCCAAAAAAAAAAAGGGG 30
                  0 ||||-----------|||| 19
query             0 CCCC-----------GGGG  8
<BLANKLINE>
>>> alignment3.coordinates
array([[11, 15, 26, 30],
       [ 0,  4,  4,  8]])
>>> format(alignment3, "psl")
'8\t0\t0\t0\t0\t0\t1\t11\t+\tquery\t8\t0\t8\ttarget\t40\t11\t30\t2\t4,4,\t0,4,\t11,26,\n'
```

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

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

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

showing that the range 122250000:122909835 of chr1 on chimpanzee genome assembly panTro5 aligns to range 111776384:112019978 of chr1 of chimpanzee genome assembly panTro6. See section 6.7.20 for more information about the chain file format.

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

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

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

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

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

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

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

This shows that nucleotide range 32:344 of expressed sequence tag DC525629 aligns to range 111982717:112009302 of chr1 of chimpanzee genome assembly panTro6. Note that the genome span of DC525629 on chimpanzee

genome assembly panTro5 is 122907314 - 122835789 = 71525 bp, while on panTro6 the genome span is 112009302 - 111982717 = 26585 bp.

6.4.5 Mapping a multiple sequence alignment

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

```
>>> from Bio import Align
>>> path = "panTro5.maf"
>>> genome_alignment = Align.read(path, "maf")
>>> for record in genome_alignment.sequences:
...     print(record.id, len(record.seq))
...
panTro5.chr1 228573443
hg19.chr1 249250621
rheMac8.chr1 225584828
calJac3.chr18 47448759
mm10.chr3 160039680
rn6.chr2 266435125
>>> genome_alignment.coordinates
array([[133922962, 133922962, 133922970, 133922970, 133922972, 133922972,
        133922995, 133922998, 133923010],
       [155784573, 155784573, 155784581, 155784581, 155784583, 155784583,
        155784606, 155784609, 155784621],
       [130383910, 130383910, 130383918, 130383918, 130383920, 130383920,
        130383943, 130383946, 130383958],
       [  9790455,   9790455,   9790463,   9790463,   9790465,   9790465,
          9790488,   9790491,   9790503],
       [ 88858039,  88858036,  88858028,  88858026,  88858024,  88858020,
         88857997,  88857997,  88857985],
       [188162970, 188162967, 188162959, 188162959, 188162957, 188162953,
        188162930, 188162930, 188162918]])
>>> print(genome_alignment)
panTro5.c 133922962 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAACTTAAAggcc
hg19.chr1 155784573 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAACTTAAAggcc
rheMac8.c 130383910 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAACTTAAAggcc
calJac3.c   9790455 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAGCTTAAAggct
mm10.chr3  88858039 TATAATAATTGTATATGTCACAGAAAAAAATGAATTTTCAAT---GACTTAATAGCC
rn6.chr2  188162970 TACAATAATTG--TATGTCATAGAAAAAAATGAATTTTCAAT---AACTTAATAGCC
<BLANKLINE>
panTro5.c 133923010
hg19.chr1 155784621
rheMac8.c 130383958
calJac3.c   9790503
mm10.chr3  88857985
rn6.chr2  188162918
<BLANKLINE>
```

Suppose we want to replace the older versions of the genome assemblies (`panTro5`, `hg19`, `rheMac8`, `calJac3`, `mm10`, and `rn6`) by their current versions (`panTro6`, `hg38`, `rheMac10`, `calJac4`, `mm39`, and `rn7`). To do so, we need the pairwise alignment between the old and the new assembly version for each species. These are provided by UCSC as chain files, typically used for UCSC's `liftOver` tool. The `.chain` files in the

`Tests/Align` subdirectory in the Biopython source distribution were extracted from UCSC's `.chain` files to only include the relevant genomic region. For example, to lift over `panTro5` to `panTro6`, we use the file `panTro5ToPanTro6.chain` with the following contents:

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

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

```
>>> paths = [
...     "panTro5ToPanTro6.chain",
...     "hg19ToHg38.chain",
...     "rheMac8ToRheMac10.chain",
...     "calJac3ToCalJac4.chain",
...     "mm10ToMm39.chain",
...     "rn6ToRn7.chain",
... ]
>>> liftover_alignments = [Align.read(path, "chain") for path in paths]
>>> for liftover_alignment in liftover_alignments:
...     print(liftover_alignment.target.id, liftover_alignment.coordinates[0, :])
...
chr1 [133919957 133924947 133924947 133926309 133926312 133932620]
chr1 [155184381 156354347 156354348 157128497 157128497 157137496]
chr1 [130382477 130383872 130383872 130384222 130384222 130388520]
chr18 [9786631 9787941 9788508 9788508 9795062 9795065 9795737]
chr3 [66807541 74196805 74196831 94707528 94707528 94708176 94708178 94708718]
chr2 [188111581 188158351 188158351 188171225 188171225 188228261 188228261
 188236997]
```

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

```
>>> genome_alignment = genome_alignment.mapall(liftover_alignments)
>>> for record in genome_alignment.sequences:
...     print(record.id, len(record.seq))
...
chr1 224244399
chr1 248956422
chr1 223616942
chr18 47031477
chr3 159745316
chr2 249053267
>>> genome_alignment.coordinates
array([[130611000, 130611000, 130611008, 130611008, 130611010, 130611010,
        130611033, 130611036, 130611048],
       [155814782, 155814782, 155814790, 155814790, 155814792, 155814792,
        155814815, 155814818, 155814830],
       [ 95186253,  95186253,  95186245,  95186245,  95186243,  95186243,
         95186220,  95186217,  95186205],
       [  9758318,   9758318,   9758326,   9758326,   9758328,   9758328,
          9758351,   9758354,   9758366],
```

```
       [ 88765346,  88765343,  88765335,  88765333,  88765331,  88765327,
         88765304,  88765304,  88765292],
       [174256702, 174256699, 174256691, 174256691, 174256689, 174256685,
        174256662, 174256662, 174256650]])
```

As the `.chain` files do not include the sequence contents, we cannot print the sequence alignment directly. Instead, we read in the genomic sequence separately (as a `.2bit` file, as it allows lazy loading; see section 5.4) for each species:

```
>>> from Bio import SeqIO
>>> names = ("panTro6", "hg38", "rheMac10", "calJac4", "mm39", "rn7")
>>> for i, name in enumerate(names):
...     filename = f"{name}.2bit"
...     genome = SeqIO.parse(filename, "twobit")
...     chromosome = genome_alignment.sequences[i].id
...     assert len(genome_alignment.sequences[i]) == len(genome[chromosome])
...     genome_alignment.sequences[i] = genome[chromosome]
...     genome_alignment.sequences[i].id = f"{name}.{chromosome}"
...
>>> print(genome_alignment)
panTro6.c 130611000 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAACTTAAAggcc
hg38.chr1 155814782 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAACTTAAAggcc
rheMac10.  95186253 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAACTTAAAggcc
calJac4.c   9758318 ---ACTAGTTA--CA----GTAACAGAaaataaaatttaaatagaagcttaaaggct
mm39.chr3  88765346 TATAATAATTGTATATGTCACAGAAAAAAATGAATTTTCAAT---GACTTAATAGCC
rn7.chr2  174256702 TACAATAATTG--TATGTCATAGAAAAAAATGAATTTTCAAT---AACTTAATAGCC
<BLANKLINE>
panTro6.c 130611048
hg38.chr1 155814830
rheMac10.  95186205
calJac4.c   9758366
mm39.chr3  88765292
rn7.chr2  174256650
<BLANKLINE>
```

The `mapall` method can also be used to create a multiple alignment of codon sequences from a multiple sequence alignment of the corresponding amino acid sequences (see Section 7.12.2 for details).

6.5 The Alignments class

The `Alignments` (plural) class inherits from `AlignmentsAbstractBaseClass` and from `list`, and can be used as a list to store `Alignment` objects. The behavior of `Alignments` objects is different from that of `list` objects in three important ways:

- An `Alignments` object is its own iterator, whereas calling `iter` on a list object creates a new iterator each time. An `Alignments` object iterator will therefore behave the same as an iterator returned by `Bio.Align.parse` (see section 6.6.1) or an iterator returned by the pairwise aligner (see Section 7).

  ```
  >>> from Bio.Align import Alignments

  >>> alignment_list = [alignment1, alignment2, alignment3]
  >>> for item in alignment_list:
  ...     print(repr(item))  # doctest: +ELLIPSIS
  ```

```
...
<Alignment object (2 rows x 24 columns) at ...>
<Alignment object (2 rows x 8 columns) at ...>
<Alignment object (2 rows x 19 columns) at ...>

>>> for item in alignment_list:
...     print(repr(item))  # doctest: +ELLIPSIS
...
<Alignment object (2 rows x 24 columns) at ...>
<Alignment object (2 rows x 8 columns) at ...>
<Alignment object (2 rows x 19 columns) at ...>

>>> alignments = Alignments([alignment1, alignment2, alignment3])
>>> for item in alignments:
...     print(repr(item))  # doctest: +ELLIPSIS
...
<Alignment object (2 rows x 24 columns) at ...>
<Alignment object (2 rows x 8 columns) at ...>
<Alignment object (2 rows x 19 columns) at ...>

>>> for item in alignments:
...     print(repr(item))  # doctest: +ELLIPSIS
...
```

The last loop does not print anything because the iterator is already exhausted.

- The `Alignments` class defines a `rewind` method that resets the iterator to its first item, allowing us to loop over it again:

```
>>> alignments.rewind()
>>> for item in alignments:
...     print(repr(item))  # doctest: +ELLIPSIS
...
<Alignment object (2 rows x 24 columns) at ...>
<Alignment object (2 rows x 8 columns) at ...>
<Alignment object (2 rows x 19 columns) at ...>
```

- Metadata can be stored as attributes on an `Alignments` object, whereas a plain `list` does not accept attributes:

```
>>> alignment_list.score = 100  # doctest: +ELLIPSIS
Traceback (most recent call last):
 ...
AttributeError: 'list' object has no attribute 'score'...
>>> alignments.score = 100
>>> alignments.score
100
```

6.6 Reading and writing alignments

Output from sequence alignment software such as Clustal can be parsed into `Alignment` objects by the `Bio.Align.read` and `Bio.Align.parse` functions. Their usage is analogous to the `read` and `parse` functions in `Bio.SeqIO` (see Section 5.1): The `read` function is used to read an output file containing a single alignment

and returns an `Alignment` object, while the `parse` function returns an iterator to iterate over alignments stored in an output file containing one or more alignments. Section 6.7 describes the alignment formats that can be parsed in `Bio.Align`. `Bio.Align` also provides a `write` function that can write alignments in most of these formats.

6.6.1 Reading alignments

The alignments iterator returned by `Bio.Align.parse` inherits from the `AlignmentsAbstractBaseClass` base class. This class defines a `rewind` method that resets the iterator to let it loop over the alignments from the beginning. You can also call `len` on the alignments to obtain the number of alignments. Depending on the file format, the number of alignments may be explicitly stored in the file (for example in the case of bigBed, bigPsl, and bigMaf files), or otherwise the number of alignments is counted by looping over them once (and returning the iterator to its original position). If the file is large, it may therefore take a considerable amount of time for `len` to return. As the number of alignments is cached, subsequent calls to `len` will return quickly.

Depending on the file format, the alignments object returned by `Bio.Align.parse` may contain attributes that store metadata found in the file, such as the version number of the software that was used to create the alignments. The specific attributes stored for each file format are described in Section 6.7.

6.6.2 Writing alignments

To write alignments to a file, use

```
>>> from Bio import Align
>>> target = "myfile.txt"
>>> Align.write(alignments, target, "clustal")
```

where `alignments` is either a single alignment or a list of alignments, `target` is a file name or an open file-like object, and `"clustal"` is the file format to be used. As some file formats allow or require metadata to be stored with the alignments, you may want to use the `Alignments` (plural) class instead of a plain list of alignments (see Section 6.5), alloing you to store a metadata dictionary as an attribute on the `alignments` object:

```
>>> from Bio import Align
>>> alignments = Align.Alignments(alignments)
>>> metadata = {"Program": "Biopython", "Version": "1.81"}
>>> alignments.metadata = metadata
>>> target = "myfile.txt"
>>> Align.write(alignments, target, "clustal")
```

6.6.3 Printing alignments

For text (non-binary) formats, you can call Python's built-in `format` function on an alignment to get a string showing the alignment in the requested format, or use `Alignment` objects in formatted (f-) strings. If called without an argument, the `format` function returns the string representation of the alignment:

```
>>> str(alignment)
'                  1 CGGTTTTT 9\n                  0 AGGTTT-- 6\n                  0 AG-TTT-- 5\n'
>>> format(alignment)
'                  1 CGGTTTTT 9\n                  0 AGGTTT-- 6\n                  0 AG-TTT-- 5\n'
>>> print(format(alignment))
                   1 CGGTTTTT 9
                   0 AGGTTT-- 6
```

```
                0 AG-TTT-- 5
<BLANKLINE>
```

By specifying one of the formats shown in Section 6.7, `format` will create a string showing the alignment in the requested format:

```
>>> format(alignment, "clustal")
'sequence_0                          CGGTTTTT\nsequence_1                          AGGTTT--\nsequence_2
>>> print(format(alignment, "clustal"))
sequence_0                          CGGTTTTT
sequence_1                          AGGTTT--
sequence_2                          AG-TTT--
<BLANKLINE>
<BLANKLINE>
<BLANKLINE>
>>> print(f"*** this is the alignment in Clustal format: ***\n{alignment:clustal}\n***")
*** this is the alignment in Clustal format: ***
sequence_0                          CGGTTTTT
sequence_1                          AGGTTT--
sequence_2                          AG-TTT--
<BLANKLINE>
<BLANKLINE>
<BLANKLINE>
***
>>> format(alignment, "maf")
'a\ns sequence_0 1 8 + 9 CGGTTTTT\ns sequence_1 0 6 + 6 AGGTTT--\ns sequence_2 0 5 + 7 AG-TTT--\n\n'
>>> print(format(alignment, "maf"))
a
s sequence_0 1 8 + 9 CGGTTTTT
s sequence_1 0 6 + 6 AGGTTT--
s sequence_2 0 5 + 7 AG-TTT--
<BLANKLINE>
<BLANKLINE>
```

As optional keyword arguments cannot be used with Python's built-in `format` function or with formatted strings, the `Alignment` class has a `format` method with optional arguments to customize the alignment format, as described in the subsections below. For example, we can print the alignment in BED format (see section 6.7.14) with a specific number of columns:

```
>>> print(pairwise_alignment)
target            1 CGGTTTTT 9
                  0 .|-|||-- 8
query             0 AG-TTT-- 5
<BLANKLINE>
>>> print(format(pairwise_alignment, "bed"))  # doctest: +NORMALIZE_WHITESPACE
target        1        7        query        0        +        1        7        0        2        2,3,
<BLANKLINE>
>>> print(pairwise_alignment.format("bed"))  # doctest: +NORMALIZE_WHITESPACE
target        1        7        query        0        +        1        7        0        2        2,3,
<BLANKLINE>
>>> print(pairwise_alignment.format("bed", bedN=3))  # doctest: +NORMALIZE_WHITESPACE
target        1        7
<BLANKLINE>
```

```
>>> print(pairwise_alignment.format("bed", bedN=6))  # doctest: +NORMALIZE_WHITESPACE
target        1        7        query        0        +
<BLANKLINE>
```

6.7 Alignment file formats

The table below shows the alignment formats that can be parsed in Bio.Align. The format argument `fmt` used in `Bio.Align` functions to specify the file format is case-insensitive. Most of these file formats can also be written by `Bio.Align`, as shown in the table.

File format `fmt`	Description	text / binary	Supported by `write`	Subsection
`a2m`	A2M	text	yes	6.7.11
`bed`	Browser Extensible Data (BED)	text	yes	6.7.14
`bigbed`	bigBed	binary	yes	6.7.15
`bigmaf`	bigMaf	binary	yes	6.7.19
`bigpsl`	bigPsl	binary	yes	6.7.17
`chain`	UCSC chain file	text	yes	6.7.20
`clustal`	ClustalW	text	yes	6.7.2
`emboss`	EMBOSS	text	no	6.7.5
`exonerate`	Exonerate	text	yes	6.7.7
`fasta`	Aligned FASTA	text	yes	6.7.1
`hhr`	HH-suite output files	text	no	6.7.10
`maf`	Multiple Alignment Format (MAF)	text	yes	6.7.18
`mauve`	Mauve eXtended Multi-FastA (xmfa) format	text	yes	6.7.12
`msf`	GCG Multiple Sequence Format (MSF)	text	no	6.7.6
`nexus`	NEXUS	text	yes	6.7.8
`phylip`	PHYLIP output files	text	yes	6.7.4
`psl`	Pattern Space Layout (PSL)	text	yes	6.7.16
`sam`	Sequence Alignment/Map (SAM)	text	yes	6.7.13
`stockholm`	Stockholm	text	yes	6.7.3
`tabular`	Tabular output from BLAST or FASTA	text	no	6.7.9

6.7.1 Aligned FASTA

Files in the aligned FASTA format store exactly one (pairwise or multiple) sequence alignment, in which gaps in the alignment are represented by dashes (`-`). Use `fmt="fasta"` to read or write files in the aligned FASTA format. Note that this is different from output generated by William Pearson's FASTA alignment program (parsing such output is described in section 6.7.9 instead).

The file `probcons.fa` in Biopython's test suite stores one multiple alignment in the aligned FASTA format. The contents of this file is as follows:

```
>plas_horvu
D-VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG-VD-VSKISQEEYLTAPGETFSVTLTV---PGTYGFYCEPHAGAGMVGKVTV
>plas_chlre
--VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG-VN-ADAISRDDYLNAPGETYSVKLTA---AGEYGYYCEPHQGAGMVGKIIV
>plas_anava
--VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKSADLAKSLSHKQLLMSPGQSTSTTFPADAPAGEYTFYCEPHRGAGMVGKITV
>plas_proho
VQIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG-ES-APALSNTKLRIAPGSFYSVTLGT---PGTYSFYCTPHRGAGMVGTITV
```

```
>azup_achcy
VHMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG-AE-A-------FKSKINENYKVTFTA---PGVYGVKCTPHYGMGMVGVVEV
```

To read this file, use

```
>>> from Bio import Align
>>> alignment = Align.read("probcons.fa", "fasta")
>>> alignment  # doctest: +ELLIPSIS
<Alignment object (5 rows x 101 columns) at ...>
```

We can print the alignment to see its default representation:

```
>>> print(alignment)
plas_horv         0 D-VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG-VD-VSKISQE
plas_chlr         0 --VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG-VN-ADAISRD
plas_anav         0 --VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKSADLAKSLSHK
plas_proh         0 VQIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG-ES-APALSNT
azup_achc         0 VHMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG-AE-A------
<BLANKLINE>
plas_horv        57 EYLTAPGETFSVTLTV---PGTYGFYCEPHAGAGMVGKVTV 95
plas_chlr        56 DYLNAPGETYSVKLTA---AGEYGYYCEPHQGAGMVGKIIV 94
plas_anav        58 QLLMSPGQSTSTTFPADAPAGEYTFYCEPHRGAGMVGKITV 99
plas_proh        56 KLRIAPGSFYSVTLGT---PGTYSFYCTPHRGAGMVGTITV 94
azup_achc        51 -FKSKINENYKVTFTA---PGVYGVKCTPHYGMGMVGVVEV 88
<BLANKLINE>
```

or we can print it in the aligned FASTA format:

```
>>> print(format(alignment, "fasta"))
>plas_horvu
D-VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG-VD-VSKISQEEYLTAPGETFSVTLTV---PGTYGFYCEPHAGAGMVGKVTV
>plas_chlre
--VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG-VN-ADAISRDDYLNAPGETYSVKLTA---AGEYGYYCEPHQGAGMVGKIIV
>plas_anava
--VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKSADLAKSLSHKQLLMSPGQSTSTTFPADAPAGEYTFYCEPHRGAGMVGKITV
>plas_proho
VQIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG-ES-APALSNTKLRIAPGSFYSVTLGT---PGTYSFYCTPHRGAGMVGTITV
>azup_achcy
VHMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG-AE-A-------FKSKINENYKVTFTA---PGVYGVKCTPHYGMGMVGVVEV
<BLANKLINE>
```

or any other available format, for example Clustal (see section 6.7.2):

```
>>> print(format(alignment, "clustal"))
plas_horvu                          D-VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG-
plas_chlre                          --VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG-
plas_anava                          --VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKS
plas_proho                          VQIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG-
azup_achcy                          VHMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG-
<BLANKLINE>
plas_horvu                          VD-VSKISQEEYLTAPGETFSVTLTV---PGTYGFYCEPHAGAGMVGKVT
plas_chlre                          VN-ADAISRDDYLNAPGETYSVKLTA---AGEYGYYCEPHQGAGMVGKII
plas_anava                          ADLAKSLSHKQLLMSPGQSTSTTFPADAPAGEYTFYCEPHRGAGMVGKIT
plas_proho                          ES-APALSNTKLRIAPGSFYSVTLGT---PGTYSFYCTPHRGAGMVGTIT
azup_achcy                          AE-A-------FKSKINENYKVTFTA---PGVYGVKCTPHYGMGMVGVVE
```

```
<BLANKLINE>
plas_horvu                          V
plas_chlre                          V
plas_anava                          V
plas_proho                          V
azup_achcy                          V
<BLANKLINE>
<BLANKLINE>
<BLANKLINE>
```

The sequences associated with the alignment are `SeqRecord` objects:

```
>>> alignment.sequences
[SeqRecord(seq=Seq('DVLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSGVDVSKI...VTV'), id='plas_horvu', na
```

Note that these sequences do not contain gaps ("-" characters), as the alignment information is stored in the `coordinates` attribute instead:

```
>>> alignment.coordinates
array([[ 0,  1,  1, 33, 34, 42, 44, 48, 48, 50, 50, 51, 58, 73, 73, 95],
       [ 0,  0,  0, 32, 33, 41, 43, 47, 47, 49, 49, 50, 57, 72, 72, 94],
       [ 0,  0,  0, 32, 33, 41, 43, 47, 48, 50, 51, 52, 59, 74, 77, 99],
       [ 0,  1,  2, 34, 35, 43, 43, 47, 47, 49, 49, 50, 57, 72, 72, 94],
       [ 0,  1,  2, 34, 34, 42, 44, 48, 48, 50, 50, 51, 51, 66, 66, 88]])
```

Use `Align.write` to write this alignment to a file (here, we'll use a `StringIO` object instead of a file):

```
>>> from io import StringIO
>>> stream = StringIO()
>>> Align.write(alignment, stream, "FASTA")
1
>>> print(stream.getvalue())
>plas_horvu
D-VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG-VD-VSKISQEEYLTAPGETFSVTLTV---PGTYGFYCEPHAGAGMVGKVTV
>plas_chlre
--VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG-VN-ADAISRDDYLNAPGETYSVKLTA---AGEYGYYCEPHQGAGMVGKIIV
>plas_anava
--VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKSADLAKSLSHKQLLMSPGQSTSTTFPADAPAGEYTFYCEPHRGAGMVGKITV
>plas_proho
VQIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG-ES-APALSNTKLRIAPGSFYSVTLGT---PGTYSFYCTPHRGAGMVGTITV
>azup_achcy
VHMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG-AE-A-------FKSKINENYKVTFTA---PGVYGVKCTPHYGMGMVGVVEV
<BLANKLINE>
```

Note that `Align.write` returns the number of alignments written (1, in this case).

6.7.2 ClustalW

Clustal is a set of multiple sequence alignment programs that are available both as standalone programs as as web servers. The file `opuntia.aln` (available online or in the `Doc/examples` subdirectory of the Biopython source code) is an output file generated by Clustal. Its first few lines are

```
CLUSTAL 2.1 multiple sequence alignment
```

```
gi|6273285|gb|AF191659.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273284|gb|AF191658.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273287|gb|AF191661.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273286|gb|AF191660.1|AF191      TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273290|gb|AF191664.1|AF191      TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273289|gb|AF191663.1|AF191      TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273291|gb|AF191665.1|AF191      TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
                                    ******* **** **********************************
```

...

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("opuntia.aln", "clustal")
```

The `metadata` attribute on `alignments` stores the information shown in the file header:

```
>>> alignments.metadata
{'Program': 'CLUSTAL', 'Version': '2.1'}
```

You can call `next` on the `alignments` to pull out the first (and only) alignment:

```
>>> alignment = next(alignments)
>>> print(alignment)  # doctest: +ELLIPSIS
gi|627328         0 TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT
gi|627328         0 TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT
gi|627328         0 TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT
gi|627328         0 TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT
gi|627329         0 TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT
gi|627328         0 TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT
gi|627329         0 TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAAAATGAAT
<BLANKLINE>
gi|627328        60 CTAAATGATATACGATTCCACTATGTAAGGTCTTTGAATCATATCATAAAAGACAATGTA
gi|627328        60 CTAAATGATATACGATTCCACTATGTAAGGTCTTTGAATCATATCATAAAAGACAATGTA
gi|627328        60 CTAAATGATATACGATTCCACTATGTAAGGTCTTTGAATCATATCATAAAAGACAATGTA
gi|627328        60 CTAAATGATATACGATTCCACTA...
```

If you are not interested in the metadata, then it is more convenient to use the `Align.read` function, as anyway each Clustal file contains only one alignment:

```
>>> from Bio import Align
>>> alignment = Align.read("opuntia.aln", "clustal")
```

The consensus line below each alignment block in the Clustal output file contains an asterisk if the sequence is conserved at each position. This information is stored in the `column_annotations` attribute of the `alignment`:

```
>>> alignment.column_annotations  # doctest: +ELLIPSIS
{'clustal_consensus': '******* **** **********************************...
```

Printing the `alignment` in `clustal` format will show the sequence alignment, but does not include the metadata:

```
>>> print(format(alignment, "clustal"))  # doctest: +ELLIPSIS
gi|6273285|gb|AF191659.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273284|gb|AF191658.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273287|gb|AF191661.1|AF191      TATACATT...
```

Writing the `alignments` in `clustal` format will include both the metadata and the sequence alignment:

```
>>> from io import StringIO
>>> stream = StringIO()
>>> alignments.rewind()
>>> Align.write(alignments, stream, "clustal")
1
>>> print(stream.getvalue())  # doctest: +ELLIPSIS
CLUSTAL 2.1 multiple sequence alignment
<BLANKLINE>
<BLANKLINE>
gi|6273285|gb|AF191659.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273284|gb|AF191658.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273287|gb|AF191661.1|AF191      TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA
gi|6273286|gb|AF191660.1|AF191      TATACATAAAAGAAG...
```

Use an `Alignments` (plural) object (see Section 6.5) if you are creating alignments by hand, and would like to include metadata information in the output.

6.7.3 Stockholm

This is an example of a protein sequence alignment in the Stockholm file format used by PFAM:

```
# STOCKHOLM 1.0
#=GF ID   7kD_DNA_binding
#=GF AC   PF02294.20
#=GF DE   7kD DNA-binding domain
#=GF AU   Mian N;0000-0003-4284-4749
#=GF AU   Bateman A;0000-0002-6982-4660
#=GF SE   Pfam-B_8148 (release 5.2)
#=GF GA   25.00 25.00;
#=GF TC   26.60 46.20;
#=GF NC   23.20 19.20;
#=GF BM   hmmbuild HMM.ann SEED.ann
#=GF SM   hmmsearch -Z 57096847 -E 1000 --cpu 4 HMM pfamseq
#=GF TP   Domain
#=GF CL   CL0049
#=GF RN   [1]
#=GF RM   3130377
#=GF RT   Microsequence analysis of DNA-binding proteins 7a, 7b, and 7e
#=GF RT   from the archaebacterium Sulfolobus acidocaldarius.
#=GF RA   Choli T, Wittmann-Liebold B, Reinhardt R;
#=GF RL   J Biol Chem 1988;263:7087-7093.
#=GF DR   INTERPRO; IPR003212;
#=GF DR   SCOP; 1sso; fa;
#=GF DR   SO; 0000417; polypeptide_domain;
#=GF CC   This family contains members of the hyper-thermophilic
#=GF CC   archaebacterium  7kD DNA-binding/endoribonuclease P2 family.
#=GF CC   There are five 7kD DNA-binding proteins, 7a-7e, found as
#=GF CC   monomers in the cell. Protein 7e shows the  tightest DNA-binding
#=GF CC   ability.
#=GF SQ   3
#=GS DN7_METS5/4-61   AC A4YEA2.1
```

```
#=GS DN7A_SACS2/3-61  AC P61991.2
#=GS DN7A_SACS2/3-61  DR PDB; 1SSO A; 2-60;
#=GS DN7A_SACS2/3-61  DR PDB; 1JIC A; 2-60;
#=GS DN7A_SACS2/3-61  DR PDB; 2CVR A; 2-60;
#=GS DN7A_SACS2/3-61  DR PDB; 1B4O A; 2-60;
#=GS DN7E_SULAC/3-60  AC P13125.2
DN7_METS5/4-61              KIKFKYKGQDLEVDISKVKKVWKVGKMVSFTYDD.NGKTGRGAVSEKDAPKELLNMIGK
DN7A_SACS2/3-61             TVKFKYKGEEKQVDISKIKKVWRVGKMISFTYDEGGGKTGRGAVSEKDAPKELLQMLEK
#=GR DN7A_SACS2/3-61  SS    EEEEESSSSEEEEETTTEEEEEESSSSEEEEEE-SSSSEEEEEEETTTS-CHHHHHHTT
DN7E_SULAC/3-60             KVRFKYKGEEKEVDTSKIKKVWRVGKMVSFTYDD.NGKTGRGAVSEKDAPKELMDMLAR
#=GC SS_cons                EEEEESSSSEEEEETTTEEEEEESSSSEEEEEE-SSSSEEEEEEETTTS-CHHHHHHTT
#=GC seq_cons               KVKFKYKGEEKEVDISKIKKVWRVGKMVSFTYDD.NGKTGRGAVSEKDAPKELLsMLuK
//
```

This is the seed alignment for the 7kD_DNA_binding (PF02294.20) PFAM entry, downloaded from the InterPro website (https://www.ebi.ac.uk/interpro/). This version of the PFAM entry is also available in the Biopython source distribution as the file `pfam2.seed.txt` in the subdirectory `Tests/Stockholm/`. We can load this file as follows:

```
>>> from Bio import Align
>>> alignment = Align.read("pfam2.seed.txt", "stockholm")
>>> alignment  # doctest: +ELLIPSIS
<Alignment object (3 rows x 59 columns) at ...>
```

We can print out a summary of the alignment:

```
>>> print(alignment)
DN7_METS5         0 KIKFKYKGQDLEVDISKVKKVWKVGKMVSFTYDD-NGKTGRGAVSEKDAPKELLNMIGK
DN7A_SACS         0 TVKFKYKGEEKQVDISKIKKVWRVGKMISFTYDEGGGKTGRGAVSEKDAPKELLQMLEK
DN7E_SULA         0 KVRFKYKGEEKEVDTSKIKKVWRVGKMVSFTYDD-NGKTGRGAVSEKDAPKELMDMLAR
<BLANKLINE>
DN7_METS5        58
DN7A_SACS        59
DN7E_SULA        58
<BLANKLINE>
```

You could also call Python's built-in `format` function on the alignment object to show it in a particular file format (see section 6.6.3 for details), for example in the Stockholm format to regenerate the file:

```
>>> print(format(alignment, "stockholm"))
# STOCKHOLM 1.0
#=GF ID   7kD_DNA_binding
#=GF AC   PF02294.20
#=GF DE   7kD DNA-binding domain
#=GF AU   Mian N;0000-0003-4284-4749
#=GF AU   Bateman A;0000-0002-6982-4660
#=GF SE   Pfam-B_8148 (release 5.2)
#=GF GA   25.00 25.00;
#=GF TC   26.60 46.20;
#=GF NC   23.20 19.20;
#=GF BM   hmmbuild HMM.ann SEED.ann
#=GF SM   hmmsearch -Z 57096847 -E 1000 --cpu 4 HMM pfamseq
#=GF TP   Domain
```

```
#=GF CL   CL0049
#=GF RN   [1]
#=GF RM   3130377
#=GF RT   Microsequence analysis of DNA-binding proteins 7a, 7b, and 7e from
#=GF RT   the archaebacterium Sulfolobus acidocaldarius.
#=GF RA   Choli T, Wittmann-Liebold B, Reinhardt R;
#=GF RL   J Biol Chem 1988;263:7087-7093.
#=GF DR   INTERPRO; IPR003212;
#=GF DR   SCOP; 1sso; fa;
#=GF DR   SO; 0000417; polypeptide_domain;
#=GF CC   This family contains members of the hyper-thermophilic
#=GF CC   archaebacterium  7kD DNA-binding/endoribonuclease P2 family. There
#=GF CC   are five 7kD DNA-binding proteins, 7a-7e, found as monomers in the
#=GF CC   cell. Protein 7e shows the  tightest DNA-binding ability.
#=GF SQ   3
#=GS DN7_METS5/4-61   AC A4YEA2.1
#=GS DN7A_SACS2/3-61  AC P61991.2
#=GS DN7A_SACS2/3-61  DR PDB; 1SSO A; 2-60;
#=GS DN7A_SACS2/3-61  DR PDB; 1JIC A; 2-60;
#=GS DN7A_SACS2/3-61  DR PDB; 2CVR A; 2-60;
#=GS DN7A_SACS2/3-61  DR PDB; 1B4O A; 2-60;
#=GS DN7E_SULAC/3-60  AC P13125.2
DN7_METS5/4-61                 KIKFKYKGQDLEVDISKVKKVWKVGKMVSFTYDD.NGKTGRGAVSEKDAPKELLNMIGK
DN7A_SACS2/3-61                TVKFKYKGEEKQVDISKIKKVWRVGKMISFTYDEGGGKTGRGAVSEKDAPKELLQMLEK
#=GR DN7A_SACS2/3-61  SS       EEEEESSSSEEEEETTTEEEEEESSSSEEEEEE-SSSSEEEEEEETTTS-CHHHHHHTT
DN7E_SULAC/3-60                KVRFKYKGEEKEVDTSKIKKVWRVGKMVSFTYDD.NGKTGRGAVSEKDAPKELMDMLAR
#=GC SS_cons                   EEEEESSSSEEEEETTTEEEEEESSSSEEEEEE-SSSSEEEEEEETTTS-CHHHHHHTT
#=GC seq_cons                  KVKFKYKGEEKEVDISKIKKVWRVGKMVSFTYDD.NGKTGRGAVSEKDAPKELLsMLuK
//
<BLANKLINE>
```

or alternatively as aligned FASTA (see section 6.7.1):

```
>>> print(format(alignment, "fasta"))
>DN7_METS5/4-61
KIKFKYKGQDLEVDISKVKKVWKVGKMVSFTYDD-NGKTGRGAVSEKDAPKELLNMIGK
>DN7A_SACS2/3-61
TVKFKYKGEEKQVDISKIKKVWRVGKMISFTYDEGGGKTGRGAVSEKDAPKELLQMLEK
>DN7E_SULAC/3-60
KVRFKYKGEEKEVDTSKIKKVWRVGKMVSFTYDD-NGKTGRGAVSEKDAPKELMDMLAR
<BLANKLINE>
```

or in the PHYLIP format (see section 6.7.4):

```
>>> print(format(alignment, "phylip"))
3 59
DN7_METS5/KIKFKYKGQDLEVDISKVKKVWKVGKMVSFTYDD-NGKTGRGAVSEKDAPKELLNMIGK
DN7A_SACS2TVKFKYKGEEKQVDISKIKKVWRVGKMISFTYDEGGGKTGRGAVSEKDAPKELLQMLEK
DN7E_SULACKVRFKYKGEEKEVDTSKIKKVWRVGKMVSFTYDD-NGKTGRGAVSEKDAPKELMDMLAR
<BLANKLINE>
```

General information of the alignment is stored under the `annotations` attribute of the `Alignment` object, for example

```
>>> alignment.annotations["identifier"]
'7kD_DNA_binding'
>>> alignment.annotations["clan"]
'CL0049'
>>> alignment.annotations["database references"]
[{'reference': 'INTERPRO; IPR003212;'}, {'reference': 'SCOP; 1sso; fa;'}, {'reference': 'SO; 0000417; p
```

The individual sequences in this alignment are stored under `alignment.sequences` as `SeqRecord`s, including any annotations associated with each sequence record:

```
>>> for record in alignment.sequences:
...     print("%s %s %s" % (record.id, record.annotations["accession"], record.dbxrefs))
...
DN7_METS5/4-61 A4YEA2.1 []
DN7A_SACS2/3-61 P61991.2 ['PDB; 1SSO A; 2-60;', 'PDB; 1JIC A; 2-60;', 'PDB; 2CVR A; 2-60;', 'PDB; 1B4O A
DN7E_SULAC/3-60 P13125.2 []
```

The secondary structure of the second sequence (`DN7A_SACS2/3-61`) is stored in the `letter_annotations` attribute of the `SeqRecord`:

```
>>> alignment.sequences[0].letter_annotations
{}
>>> alignment.sequences[1].letter_annotations
{'secondary structure': 'EEEEESSSSEEEEETTEEEEEESSSSEEEEEE-SSSSEEEEEEETTTS-CHHHHHHTT'}
>>> alignment.sequences[2].letter_annotations
{}
```

The consensus sequence and secondary structure are associated with the sequence alignment as a whole, and are therefore stored in the `column_annotations` attribute of the `Alignment` object:

```
>>> alignment.column_annotations  # doctest: +NORMALIZE_WHITESPACE
{'consensus secondary structure': 'EEEEESSSSEEEEETTEEEEEESSSSEEEEEE-SSSSEEEEEEETTTS-CHHHHHHTT',
 'consensus sequence': 'KVKFKYKGEEKEVDISKIKKVWRVGKMVSFTYDD.NGKTGRGAVSEKDAPKELLsMLuK'}
```

6.7.4 PHYLIP output files

The PHYLIP format for sequence alignments is derived from the PHYLogeny Interference Package from Joe Felsenstein. Files in the PHYLIP format start with two numbers for the number of rows and columns in the printed alignment. The sequence alignment itself can be in sequential format or in interleaved format. An example of the former is the `sequential.phy` file (provided in `Tests/Phylip/` in the Biopython source distribution):

```
 3 384
CYS1_DICDI   -----MKVIL LFVLAVFTVF VSS------- --------RG IPPEEQ---- --------SQ
             FLEFQDKFNK KY-SHEEYLE RFEIFKSNLG KIEELNLIAI NHKADTKFGV NKFADLSSDE
             FKNYYLNNKE AIFTDDLPVA DYLDDEFINS IPTAFDWRTR G-AVTPVKNQ GQCGSCWSFS
             TTGNVEGQHF ISQNKLVSLS EQNLVDCDHE CMEYEGEEAC DEGCNGGLQP NAYNYIIKNG
             GIQTESSYPY TAETGTQCNF NSANIGAKIS NFTMIP-KNE TVMAGYIVST GPLAIAADAV
             E-WQFYIGGV F-DIPCN--P NSLDHGILIV GYSAKNTIFR KNMPYWIVKN SWGADWGEQG
             YIYLRRGKNT CGVSNFVSTS II--
ALEU_HORVU   MAHARVLLLA LAVLATAAVA VASSSSFADS NPIRPVTDRA ASTLESAVLG ALGRTRHALR
             FARFAVRYGK SYESAAEVRR RFRIFSESLE EVRSTN---- RKGLPYRLGI NRFSDMSWEE
             FQATRL-GAA QTCSATLAGN HLMRDA--AA LPETKDWRED G-IVSPVKNQ AHCGSCWTFS
             TTGALEAAYT QATGKNISLS EQQLVDCAGG FNNF------ --GCNGGLPS QAFEYIKYNG
```

```
             GIDTEESYPY KGVNGV-CHY KAENAAVQVL DSVNITLNAE DELKNAVGLV RPVSVAFQVI
             DGFRQYKSGV YTSDHCGTTP DDVNHAVLAV GYGVENGV-- ---PYWLIKN SWGADWGDNG
             YFKMEMGKNM CAIATCASYP VVAA
CATH_HUMAN   ------MWAT LPLLCAGAWL LGV------- -PVCGAAELS VNSLEK---- --------FH
             FKSWMSKHRK TY-STEEYHH RLQTFASNWR KINAHN---- NGNHTFKMAL NQFSDMSFAE
             IKHKYLWSEP QNCSAT--KS NYLRGT--GP YPPSVDWRKK GNFVSPVKNQ GACGSCWTFS
             TTGALESAIA IATGKMLSLA EQQLVDCAQD FNNY------ --GCQGGLPS QAFEYILYNK
             GIMGEDTYPY QGKDGY-CKF QPGKAIGFVK DVANITIYDE EAMVEAVALY NPVSFAFEVT
             QDFMMYRTGI YSSTSCHKTP DKVNHAVLAV GYGEKNGI-- ---PYWIVKN SWGPQWGMNG
             YFLIERGKNM CGLAACASYP IPLV
```

In the sequential format, the complete alignment for one sequence is shown before proceeding to the next sequence. In the interleaved format, the alignments for different sequences are next to each other, for example in the file `interlaced.phy` (provided in `Tests/Phylip/` in the Biopython source distribution):

```
 3 384
CYS1_DICDI   -----MKVIL LFVLAVFTVF VSS------- --------RG IPPEEQ---- --------SQ
ALEU_HORVU   MAHARVLLLA LAVLATAAVA VASSSSFADS NPIRPVTDRA ASTLESAVLG ALGRTRHALR
CATH_HUMAN   ------MWAT LPLLCAGAWL LGV------- -PVCGAAELS VNSLEK---- --------FH

             FLEFQDKFNK KY-SHEEYLE RFEIFKSNLG KIEELNLIAI NHKADTKFGV NKFADLSSDE
             FARFAVRYGK SYESAAEVRR RFRIFSESLE EVRSTN---- RKGLPYRLGI NRFSDMSWEE
             FKSWMSKHRK TY-STEEYHH RLQTFASNWR KINAHN---- NGNHTFKMAL NQFSDMSFAE

             FKNYYLNNKE AIFTDDLPVA DYLDDEFINS IPTAFDWRTR G-AVTPVKNQ GQCGSCWSFS
             FQATRL-GAA QTCSATLAGN HLMRDA--AA LPETKDWRED G-IVSPVKNQ AHCGSCWTFS
             IKHKYLWSEP QNCSAT--KS NYLRGT--GP YPPSVDWRKK GNFVSPVKNQ GACGSCWTFS

             TTGNVEGQHF ISQNKLVSLS EQNLVDCDHE CMEYEGEEAC DEGCNGGLQP NAYNYIIKNG
             TTGALEAAYT QATGKNISLS EQQLVDCAGG FNNF------ --GCNGGLPS QAFEYIKYNG
             TTGALESAIA IATGKMLSLA EQQLVDCAQD FNNY------ --GCQGGLPS QAFEYILYNK

             GIQTESSYPY TAETGTQCNF NSANIGAKIS NFTMIP-KNE TVMAGYIVST GPLAIAADAV
             GIDTEESYPY KGVNGV-CHY KAENAAVQVL DSVNITLNAE DELKNAVGLV RPVSVAFQVI
             GIMGEDTYPY QGKDGY-CKF QPGKAIGFVK DVANITIYDE EAMVEAVALY NPVSFAFEVT

             E-WQFYIGGV F-DIPCN--P NSLDHGILIV GYSAKNTIFR KNMPYWIVKN SWGADWGEQG
             DGFRQYKSGV YTSDHCGTTP DDVNHAVLAV GYGVENGV-- ---PYWLIKN SWGADWGDNG
             QDFMMYRTGI YSSTSCHKTP DKVNHAVLAV GYGEKNGI-- ---PYWIVKN SWGPQWGMNG

             YIYLRRGKNT CGVSNFVSTS II--
             YFKMEMGKNM CAIATCASYP VVAA
             YFLIERGKNM CGLAACASYP IPLV
```

The parser in `Bio.Align` detects from the file contents if it is in the sequential or in the interleaved format, and then parses it appropriately.

```
>>> from Bio import Align
>>> alignment = Align.read("sequential.phy", "phylip")
>>> alignment  # doctest: +ELLIPSIS
<Alignment object (3 rows x 384 columns) at ...>
>>> alignment2 = Align.read("interlaced.phy", "phylip")
>>> alignment2  # doctest: +ELLIPSIS
```

```
<Alignment object (3 rows x 384 columns) at ...>
>>> alignment == alignment2
True
```

Here, two alignments are considered to be equal if they have the same sequence contents and the same alignment coordinates.

```
>>> alignment.shape
(3, 384)
>>> print(alignment)
CYS1_DICD         0 -----MKVILLFVLAVFTVFVSS---------------RGIPPEEQ------------SQ
ALEU_HORV         0 MAHARVLLLALAVLATAAVAVASSSSFADSNPIRPVTDRAASTLESAVLGALGRTRHALR
CATH_HUMA         0 ------MWATLPLLCAGAWLLGV--------PVCGAAELSVNSLEK------------FH
<BLANKLINE>
CYS1_DICD        28 FLEFQDKFNKKY-SHEEYLERFEIFKSNLGKIEELNLIAINHKADTKFGVNKFADLSSDE
ALEU_HORV        60 FARFAVRYGKSYESAAEVRRRFRIFSESLEEVRSTN----RKGLPYRLGINRFSDMSWEE
CATH_HUMA        34 FKSWMSKHRKTY-STEEYHHRLQTFASNWRKINAHN----NGNHTFKMALNQFSDMSFAE
<BLANKLINE>
CYS1_DICD        87 FKNYYLNNKEAIFTDDLPVADYLDDEFINSIPTAFDWRTRG-AVTPVKNQGQCGSCWSFS
ALEU_HORV       116 FQATRL-GAAQTCSATLAGNHLMRDA--AALPETKDWREDG-IVSPVKNQAHCGSCWTFS
CATH_HUMA        89 IKHKYLWSEPQNCSAT--KSNYLRGT--GPYPPSVDWRKKGNFVSPVKNQGACGSCWTFS
<BLANKLINE>
CYS1_DICD       146 TTGNVEGQHFISQNKLVSLSEQNLVDCDHECMEYEGEEACDEGCNGGLQPNAYNYIIKNG
ALEU_HORV       172 TTGALEAAYTQATGKNISLSEQQLVDCAGGFNNF--------GCNGGLPSQAFEYIKYNG
CATH_HUMA       145 TTGALESAIAIATGKMLSLAEQQLVDCAQDFNNY--------GCQGGLPSQAFEYILYNK
<BLANKLINE>
CYS1_DICD       206 GIQTESSYPYTAETGTQCNFNSANIGAKISNFTMIP-KNETVMAGYIVSTGPLAIAADAV
ALEU_HORV       224 GIDTEESYPYKGVNGV-CHYKAENAAVQVLDSVNITLNAEDELKNAVGLVRPVSVAFQVI
CATH_HUMA       197 GIMGEDTYPYQGKDGY-CKFQPGKAIGFVKDVANITIYDEEAMVEAVALYNPVSFAFEVT
<BLANKLINE>
CYS1_DICD       265 E-WQFYIGGVF-DIPCN--PNSLDHGILIVGYSAKNTIFRKNMPYWIVKNSWGADWGEQG
ALEU_HORV       283 DGFRQYKSGVYTSDHCGTTPDDVNHAVLAVGYGVENGV-----PYWLIKNSWGADWGDNG
CATH_HUMA       256 QDFMMYRTGIYSSTSCHKTPDKVNHAVLAVGYGEKNGI-----PYWIVKNSWGPQWGMNG
<BLANKLINE>
CYS1_DICD       321 YIYLRRGKNTCGVSNFVSTSII-- 343
ALEU_HORV       338 YFKMEMGKNMCAIATCASYPVVAA 362
CATH_HUMA       311 YFLIERGKNMCGLAACASYPIPLV 335
<BLANKLINE>
```

When outputting the alignment in PHYLIP format, `Bio.Align` writes each of the aligned sequences on one line:

```
>>> print(format(alignment, "phylip"))
3 384
CYS1_DICDI-----MKVILLFVLAVFTVFVSS---------------RGIPPEEQ------------SQFLEFQDKFNKKY-SHEEYLERFEIFKSNLGKIE
ALEU_HORVUMAHARVLLLALAVLATAAVAVASSSSFADSNPIRPVTDRAASTLESAVLGALGRTRHALRFARFAVRYGKSYESAAEVRRRFRIFSESLEEVR
CATH_HUMAN------MWATLPLLCAGAWLLGV--------PVCGAAELSVNSLEK------------FHFKSWMSKHRKTY-STEEYHHRLQTFASNWRKIN
<BLANKLINE>
```

We can write the alignment in PHYLIP format, parse the result, and confirm it is the same as the original alignment object:

```
>>> from io import StringIO
>>> stream = StringIO()
```

```
>>> Align.write(alignment, stream, "phylip")
1
>>> stream.seek(0)
0
>>> alignment3 = Align.read(stream, "phylip")
>>> alignment == alignment3
True
>>> [record.id for record in alignment.sequences]
['CYS1_DICDI', 'ALEU_HORVU', 'CATH_HUMAN']
>>> [record.id for record in alignment3.sequences]
['CYS1_DICDI', 'ALEU_HORVU', 'CATH_HUMAN']
```

6.7.5 EMBOSS

EMBOSS (European Molecular Biology Open Software Suite) is a set of open-source software tools for molecular biology and bioinformatics [37]. It includes software such as `needle` and `water` for pairwise sequence alignment. This is an example of output generated by the `water` program for Smith-Waterman local pairwise sequence alignment (available as `water.txt` in the `Tests/Emboss` directory of the Biopython distribution):

```
########################################
# Program:  water
# Rundate:  Wed Jan 16 17:23:19 2002
# Report_file: stdout
########################################
#=======================================
#
# Aligned_sequences: 2
# 1: IXI_234
# 2: IXI_235
# Matrix: EBLOSUM62
# Gap_penalty: 10.0
# Extend_penalty: 0.5
#
# Length: 131
# Identity:     112/131 (85.5%)
# Similarity:   112/131 (85.5%)
# Gaps:          19/131 (14.5%)
# Score: 591.5
#
#
#=======================================

IXI_234            1 TSPASIRPPAGPSSRPAMVSSRRTRPSPPGPRRPTGRPCCSAAPRRPQAT     50
                     |||||||||||||||         |||||||||||||||||||||||||
IXI_235            1 TSPASIRPPAGPSSR---------RPSPPGPRRPTGRPCCSAAPRRPQAT     41

IXI_234           51 GGWKTCSGTCTTSTSTRHRGRSGWSARTTTAACLRASRKSMRAACSRSAG    100
                     |||||||||||||||||||||||||          |||||||||||||||||
IXI_235           42 GGWKTCSGTCTTSTSTRHRGRSGW----------RASRKSMRAACSRSAG     81

IXI_234          101 SRPNRFAPTLMSSCITSTTGPPAWAGDRSHE    131
```

```
                    ||||||||||||||||||||||||||||||
IXI_235           82 SRPNRFAPTLMSSCITSTTGPPAWAGDRSHE    112


#---------------------------------------
#---------------------------------------
```

As this output file contains only one alignment, we can use `Align.read` to extract it directly. Here, instead we will use `Align.parse` so we can see the metadata of this `water` run:

```
>>> from Bio import Align
>>> alignments = Align.parse("water.txt", "emboss")
```

The `metadata` attribute of `alignments` stores the information shown in the header of the file, including the program used to generate the output, the date and time the program was run, the output file name, and the specific alignment file format that was used (assumed to be `srspair` by default):

```
>>> alignments.metadata
{'Align_format': 'srspair', 'Program': 'water', 'Rundate': 'Wed Jan 16 17:23:19 2002', 'Report_file': '
```

To pull out the alignment, we use

```
>>> alignment = next(alignments)
>>> alignment  # doctest: +ELLIPSIS
<Alignment object (2 rows x 131 columns) at ...>
>>> alignment.shape
(2, 131)
>>> print(alignment)
IXI_234           0 TSPASIRPPAGPSSRPAMVSSRRTRPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTC
                  0 ||||||||||||||||---------|||||||||||||||||||||||||||||||||||
IXI_235           0 TSPASIRPPAGPSSR---------RPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTC
<BLANKLINE>
IXI_234          60 TTSTSTRHRGRSGWSARTTTAACLRASRKSMRAACSRSAGSRPNRFAPTLMSSCITSTTG
                  60 ||||||||||||||----------|||||||||||||||||||||||||||||||||||
IXI_235          51 TTSTSTRHRGRSGW----------RASRKSMRAACSRSAGSRPNRFAPTLMSSCITSTTG
<BLANKLINE>
IXI_234         120 PPAWAGDRSHE 131
                120 |||||||||||  131
IXI_235         101 PPAWAGDRSHE 112
<BLANKLINE>
>>> alignment.coordinates
array([[  0,  15,  24,  74,  84, 131],
       [  0,  15,  15,  65,  65, 112]])
```

We can use indices to extract specific parts of the alignment:

```
>>> alignment[0]
'TSPASIRPPAGPSSRPAMVSSRRTRPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTCTTSTSTRHRGRSGWSARTTTAACLRASRKSMRAACSRSAGSR
>>> alignment[1]
'TSPASIRPPAGPSSR---------RPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTCTTSTSTRHRGRSGW----------RASRKSMRAACSRSAGSR
>>> alignment[1, 10:30]
'GPSSR---------RPSPPG'
```

The `annotations` attribute of the `alignment` stores the information associated with this alignment specifically:

```
>>> alignment.annotations
{'Matrix': 'EBLOSUM62', 'Gap_penalty': 10.0, 'Extend_penalty': 0.5, 'Identity': 112, 'Similarity': 112,
```

The number of gaps, identities, and mismatches can also be obtained by calling the `counts` method on the `alignment` object:

```
>>> alignment.counts()
AlignmentCounts(gaps=19, identities=112, mismatches=0)
```

where `AlignmentCounts` is a `namedtuple` in the `collections` module in Python's standard library.

The consensus line shown between the two sequences is stored in the `column_annotations` attribute:

```
>>> alignment.column_annotations
{'emboss_consensus': '|||||||||||||||         ||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

Use the `format` function (or the `format` method) to print the alignment in other formats, for example in the PHYLIP format (see section 6.7.4):

```
>>> print(format(alignment, "phylip"))
2 131
IXI_234   TSPASIRPPAGPSSRPAMVSSRRTRPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTCTTSTSTRHRGRSGWSARTTTAACLRASRKSMRA
IXI_235   TSPASIRPPAGPSSR---------RPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTCTTSTSTRHRGRSGW----------RASRKSMRA
<BLANKLINE>
```

We can use `alignment.sequences` to get the individual sequences. However, as this is a pairwise alignment, we can also use `alignment.target` and `alignment.query` to get the target and query sequences:

```
>>> alignment.target
SeqRecord(seq=Seq('TSPASIRPPAGPSSRPAMVSSRRTRPSPPGPRRPTGRPCCSAAPRRPQATGGWK...SHE'), id='IXI_234', name='
>>> alignment.query
SeqRecord(seq=Seq('TSPASIRPPAGPSSRRPSPPGPRRPTGRPCCSAAPRRPQATGGWKTCSGTCTTS...SHE'), id='IXI_235', name='
```

Currently, Biopython does not support writing sequence alignments in the output formats defined by EMBOSS.

6.7.6 GCG Multiple Sequence Format (MSF)

The Multiple Sequence Format (MSF) was created to store multiple sequence alignments generated by the GCG (Genetics Computer Group) set of programs. The file `W_prot.msf` in the `Tests/msf` directory of the Biopython distribution is an example of a sequence alignment file in the MSF format This file shows an alignment of 11 protein sequences:

```
!!AA_MULTIPLE_ALIGNMENT

   MSF: 99  Type: P  Oct 18, 2017  11:35  Check: 0 ..

 Name: W*01:01:01:01    Len:    99  Check: 7236  Weight:  1.00
 Name: W*01:01:01:02    Len:    99  Check: 7236  Weight:  1.00
 Name: W*01:01:01:03    Len:    99  Check: 7236  Weight:  1.00
 Name: W*01:01:01:04    Len:    99  Check: 7236  Weight:  1.00
 Name: W*01:01:01:05    Len:    99  Check: 7236  Weight:  1.00
 Name: W*01:01:01:06    Len:    99  Check: 7236  Weight:  1.00
 Name: W*02:01          Len:    93  Check: 9483  Weight:  1.00
 Name: W*03:01:01:01    Len:    93  Check: 9974  Weight:  1.00
 Name: W*03:01:01:02    Len:    93  Check: 9974  Weight:  1.00
```

```
 Name: W*04:01          Len:    93  Check: 9169  Weight:  1.00
 Name: W*05:01          Len:    99  Check: 7331  Weight:  1.00
//

  W*01:01:01:01  GLTPFNGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
  W*01:01:01:02  GLTPFNGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
  W*01:01:01:03  GLTPFNGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
  W*01:01:01:04  GLTPFNGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
  W*01:01:01:05  GLTPFNGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
  W*01:01:01:06  GLTPFNGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
        W*02:01  GLTPSNGYTA ATWTRTAASS VGMNIPYDGA SYLVRNQELR SWTAADKAAQ
  W*03:01:01:01  GLTPSSGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
  W*03:01:01:02  GLTPSSGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ
        W*04:01  GLTPSNGYTA ATWTRTAASS VGMNIPYDGA SYLVRNQELR SWTAADKAAQ
        W*05:01  GLTPSSGYTA ATWTRTAVSS VGMNIPYHGA SYLVRNQELR SWTAADKAAQ

  W*01:01:01:01  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
  W*01:01:01:02  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
  W*01:01:01:03  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
  W*01:01:01:04  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
  W*01:01:01:05  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
  W*01:01:01:06  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
        W*02:01  MPWRRNMQSC SKPTCREGGR SGSAKSLRMG RRRCTAQNPK RLT
  W*03:01:01:01  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK RLT
  W*03:01:01:02  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK RLT
        W*04:01  MPWRRNMQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK RLT
        W*05:01  MPWRRNRQSC SKPTCREGGR SGSAKSLRMG RRGCSAQNPK DSHDPPPHL
```

To parse this file with Biopython, use

```
>>> from Bio import Align
>>> alignment = Align.read("W_prot.msf", "msf")
```

The parser skips all lines up to and including the line starting with ”`MSF:`”. The following lines (until the ”`//`” demarcation) are read by the parser to verify the length of each sequence. The alignment section (after the ”`//`” demarcation) is read by the parser and stored as an `Alignment` object:

```
>>> alignment  # doctest: +ELLIPSIS
<Alignment object (11 rows x 99 columns) at ...>
>>> print(alignment)
W*01:01:0         0 GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*01:01:0         0 GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*01:01:0         0 GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*01:01:0         0 GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*01:01:0         0 GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*01:01:0         0 GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*02:01           0 GLTPSNGYTAATWTRTAASSVGMNIPYDGASYLVRNQELRSWTAADKAAQMPWRRNMQSC
W*03:01:0         0 GLTPSSGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*03:01:0         0 GLTPSSGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
W*04:01           0 GLTPSNGYTAATWTRTAASSVGMNIPYDGASYLVRNQELRSWTAADKAAQMPWRRNMQSC
W*05:01           0 GLTPSSGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWRRNRQSC
<BLANKLINE>
W*01:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
```

```
W*01:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
W*01:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
W*01:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
W*01:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
W*01:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
W*02:01          60 SKPTCREGGRSGSAKSLRMGRRRCTAQNPKRLT------ 93
W*03:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKRLT------ 93
W*03:01:0        60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKRLT------ 93
W*04:01          60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKRLT------ 93
W*05:01          60 SKPTCREGGRSGSAKSLRMGRRGCSAQNPKDSHDPPPHL 99
<BLANKLINE>
```

The sequences and their names are stored in the `alignment.sequences` attribute:

```
>>> len(alignment.sequences)
11
>>> alignment.sequences[0].id
'W*01:01:01:01'
>>> alignment.sequences[0].seq
Seq('GLTPFNGYTAATWTRTAVSSVGMNIPYHGASYLVRNQELRSWTAADKAAQMPWR...PHL')
```

The alignment coordinates are stored in the `alignment.coordinates` attribute:

```
>>> alignment.coordinates
array([[ 0, 93, 99],
       [ 0, 93, 99],
       [ 0, 93, 99],
       [ 0, 93, 99],
       [ 0, 93, 99],
       [ 0, 93, 99],
       [ 0, 93, 93],
       [ 0, 93, 93],
       [ 0, 93, 93],
       [ 0, 93, 93],
       [ 0, 93, 99]])
```

Currently, Biopython does not support writing sequence alignments in the MSF format.

6.7.7 Exonerate

Exonerate is a generic program for pairwise sequence alignments [42]. The sequence alignments found by Exonerate can be output in a human-readable form, in the "cigar" (Compact Idiosyncratic Gapped Alignment Report) format, or in the "vulgar" (Verbose Useful Labelled Gapped Alignment Report) format. The user can request to include one or more of these formats in the output. The parser in `Bio.Align` can only parse alignments in the cigar or vulgar formats, and will not parse output that includes alignments in human-readable format.

The file `exn_22_m_cdna2genome_vulgar.exn` in the Biopython test suite is an example of an Exonerate output file showing the alignments in vulgar format:

```
Command line: [exonerate -m cdna2genome ../scer_cad1.fa /media/Waterloo/Downloads/genomes/scer_s288c/sc
Hostname: [blackbriar]
vulgar: gi|296143771|ref|NM_001180731.1| 0 1230 + gi|330443520|ref|NC_001136.10| 1319275 1318045 - 6146
vulgar: gi|296143771|ref|NM_001180731.1| 1230 0 - gi|330443520|ref|NC_001136.10| 1318045 1319275 + 6146
vulgar: gi|296143771|ref|NM_001180731.1| 0 516 + gi|330443688|ref|NC_001145.3| 85010 667216 + 518 M 11
-- completed exonerate analysis
```

This file includes three alignments. To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("exn_22_m_cdna2genome_vulgar.exn", "exonerate")
```

The dictionary `alignments.metadata` stores general information about these alignments, shown at the top of the output file:

```
>>> alignments.metadata  # doctest: +NORMALIZE_WHITESPACE
{'Program': 'exonerate',
 'Command line': 'exonerate -m cdna2genome ../scer_cad1.fa /media/Waterloo/Downloads/genomes/scer_s288c/
 'Hostname': 'blackbriar'}
```

Now we can iterate over the alignments. The first alignment, with alignment score 6146.0, has no gaps:

```
>>> alignment = next(alignments)
>>> alignment.score
6146.0
>>> alignment.coordinates
array([[1319275, 1319274, 1319271, 1318045],
       [      0,       1,       4,    1230]])
>>> print(alignment)  # doctest: +ELLIPSIS
gi|330443   1319275 ??????????????????????????????????????????????????????????
                  0 ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
gi|296143         0 ??????????????????????????????????????????????????????????
...
gi|330443   1318075 ????????????????????????????????? 1318045
               1200 |||||||||||||||||||||||||||||||||    1230
gi|296143      1200 ?????????????????????????????????    1230
<BLANKLINE>
```

Note that the target (the first sequence) in the printed alignment is on the reverse strand while the query (the second sequence) is on the forward strand, with the target coordinate decreasing and the query coordinate increasing. Printing this alignment in `exonerate` format using Python's built-in `format` function writes a vulgar line:

```
>>> print(format(alignment, "exonerate"))
vulgar: gi|296143771|ref|NM_001180731.1| 0 1230 + gi|330443520|ref|NC_001136.10| 1319275 1318045 - 6146
<BLANKLINE>
```

Using the `format` method allows us to request either a vulgar line (default) or a cigar line:

```
>>> print(alignment.format("exonerate", "vulgar"))
vulgar: gi|296143771|ref|NM_001180731.1| 0 1230 + gi|330443520|ref|NC_001136.10| 1319275 1318045 - 6146
<BLANKLINE>
>>> print(alignment.format("exonerate", "cigar"))
cigar: gi|296143771|ref|NM_001180731.1| 0 1230 + gi|330443520|ref|NC_001136.10| 1319275 1318045 - 6146 I
<BLANKLINE>
```

The vulgar line contains information about the alignment (in the section `M 1 1 C 3 3 M 1226`) that is missing from the cigar line `M 1 M 3 M 1226`. The vulgar line specifies that the alignment starts with a single aligned nucleotides, followed by three aligned nucleotides that form a codon (`C`), followed by 1226 aligned nucleotides. In the cigar line, we see a single aligned nucleotide, followed by three aligned nucleotides, followed by 1226 aligned nucleotides; it does not specify that the three aligned nucleotides form a codon. This information from the vulgar line is stored in the `operations` attribute:

```
>>> alignment.operations
bytearray(b'MCM')
```

See the Exonerate documentation for the definition of other operation codes.

Similarly, the `"vulgar"` or `"cigar"` argument can be used when calling `Bio.Align.write` to write a file with vulgar or cigar alignment lines.

We can print the alignment in BED and PSL format:

```
>>> print(format(alignment, "bed"))  # doctest: +NORMALIZE_WHITESPACE
gi|330443520|ref|NC_001136.10|        1318045        1319275        gi|296143771|ref|NM_001180731.1| 614
<BLANKLINE>
>>> print(format(alignment, "psl"))  # doctest: +NORMALIZE_WHITESPACE
1230        0        0        0        0        0        0        0        -        gi|296143771|ref|NM_
<BLANKLINE>
```

The SAM format parser defines its own (optional) `operations` attribute (section 6.7.13), which is not quite consistent with the `operations` attribute defined in the Exonerate format parser. As the `operations` attribute is optional, we delete it before printing the alignment in SAM format:

```
>>> del alignment.operations
>>> print(format(alignment, "sam"))  # doctest: +NORMALIZE_WHITESPACE
gi|296143771|ref|NM_001180731.1|        16        gi|330443520|ref|NC_001136.10|        1318046        1
<BLANKLINE>
```

The third alignment contains four long gaps:

```
>>> alignment = next(alignments)  # second alignment
>>> alignment = next(alignments)  # third alignment
>>> print(alignment)  # doctest: +ELLIPSIS
gi|330443     85010 ??????????-???????????????--????-?-????????----???????????
                  0 ||||||||||-|||||||||||||||--||||-|-||||||||----||||||||||||
gi|296143          0 ??????????????????????????????????????????????????????????
<BLANKLINE>
gi|330443     85061 ??????????????????????????????????????????????????????????
                 60 |||||----------------------------------------------------------
gi|296143         60 ?????----------------------------------------------------------
...
gi|330443    666990 ??????????????????????????????????????????????????????????
             582000 ------------------------------------------------------|||||||||
gi|296143        346 ------------------------------------------------------??????????
<BLANKLINE>
gi|330443    667050 ?????????-????????????????????????????????????????????????
             582060 ||--|||||-|||||||--|-||||||||||||||||||||||||||||||||||||||||
gi|296143        356 ??--?????????????--?-????????????????????????????????????
<BLANKLINE>
gi|330443    667109 ????????????????????????????????????????????????????-?????
             582120 ||||||||||||||||||||||||||||||||||||||||||||||||||||||-||||-
gi|296143        411 ????????????????????????????????????????????????????????-
<BLANKLINE>
gi|330443    667168 ?????????????????????????????????????????????????? 667216
             582180 ||-|||-|||||||||||||||||||||||||||||||||||||||||||| 582228
gi|296143        470 ??-???-?????????????????????????????????????????    516
<BLANKLINE>
>>> print(format(alignment, "exonerate"))  # doctest: +NORMALIZE_WHITESPACE
```

```
vulgar: gi|296143771|ref|NM_001180731.1| 0 516 + gi|330443688|ref|NC_001145.3|
85010 667216 + 518 M 11 11 G 1 0 M 15 15 G 2 0 M 4 4 G 1 0 M 1 1 G 1 0 M 8 8
 G 4 0 M 17 17 5 0 2 I 0 168904 3 0 2 M 4 4 G 0 1 M 8 8 G 2 0 M 3 3 G 1 0
 M 33 33 G 0 2 M 7 7 G 0 1 M 102 102 5 0 2 I 0 96820 3 0 2 M 14 14 G 0 2 M 10 10
 G 2 0 M 5 5 G 0 2 M 10 10 G 2 0 M 4 4 G 0 1 M 20 20 G 1 0 M 15 15 G 0 1 M 5 5
 G 3 0 M 4 4 5 0 2 I 0 122114 3 0 2 M 20 20 G 0 5 M 6 6 5 0 2 I 0 193835 3 0 2
 M 12 12 G 0 2 M 5 5 G 1 0 M 7 7 G 0 2 M 1 1 G 0 1 M 12 12 C 75 75 M 6 6 G 1 0
 M 4 4 G 0 1 M 2 2 G 0 1 M 3 3 G 0 1 M 41 41
<BLANKLINE>
```

6.7.8 NEXUS

The NEXUS file format [30] is used by several programs to store phylogenetic information. This is an example of a file in the NEXUS format (available as `codonposset.nex` in the `Tests/Nexus` subdirectory in the Biopython distribution):

```
#NEXUS
[MacClade 4.05 registered to Computational Biologist, University]


BEGIN DATA;
       DIMENSIONS  NTAX=2 NCHAR=22;
       FORMAT DATATYPE=DNA  MISSING=? GAP=- ;
MATRIX
[                          10        20 ]
[                          .         .  ]

Aegotheles         AAAAAGGCATTGTGGTGGGAAT   [22]
Aerodramus         ??????????TTGTGGTGGGAAT   [13]
;
END;


BEGIN CODONS;
       CODONPOSSET * CodonPositions =
              N: 1-10,
              1: 11-22\3,
              2: 12-22\3,
              3: 13-22\3;
       CODESET  * UNTITLED = Universal: all ;
END;
```

In general, files in the NEXUS format can be much more complex. `Bio.Align` relies heavily on NEXUS parser in `Bio.Nexus` (see Chapter 16) to extract `Alignment` objects from NEXUS files.

To read the alignment in this NEXUS file, use

```
>>> from Bio import Align
>>> alignment = Align.read("codonposset.nex", "nexus")
>>> print(alignment)
Aegothele         0 AAAAAGGCATTGTGGTGGGAAT 22
                  0 .........|||||||||||||| 22
Aerodramu         0 ??????????TTGTGGTGGGAAT 22
<BLANKLINE>
```

```
>>> alignment.shape
(2, 22)
```

The sequences are stored under the `sequences` attribute:

```
>>> alignment.sequences[0].id
'Aegotheles'
>>> alignment.sequences[0].seq
Seq('AAAAAGGCATTGTGGTGGGAAT')
>>> alignment.sequences[0].annotations
{'molecule_type': 'DNA'}
>>> alignment.sequences[1].id
'Aerodramus'
>>> alignment.sequences[1].seq
Seq('??????????TTGTGGTGGGAAT')
>>> alignment.sequences[1].annotations
{'molecule_type': 'DNA'}
```

To print this alignment in the NEXUS format, use

```
>>> print(format(alignment, "nexus"))
#NEXUS
begin data;
dimensions ntax=2 nchar=22;
format datatype=dna missing=? gap=-;
matrix
Aegotheles AAAAAGGCATTGTGGTGGGAAT
Aerodramus ??????????TTGTGGTGGGAAT
;
end;
<BLANKLINE>
```

Similarly, you can use `Align.write(alignment, "myfilename.nex", "nexus")` to write the alignment in the NEXUS format to the file `myfilename.nex`.

6.7.9 Tabular output from BLAST or FASTA

Alignment output in tabular output is generated by the FASTA aligner [34] run with the `-m 8CB` or `-m 8CC` argument, or by BLAST [1] run with the `-outfmt 7` argument.

The file `nucleotide_m8CC.txt` in the `Tests/Fasta` subdirectory of the Biopython source distribution is an example of an output file generated by FASTA with the `-m 8CC` argument:

```
# fasta36 -m 8CC seq/mgstm1.nt seq/gst.nlib
# FASTA 36.3.8h May, 2020
# Query: pGT875  - 657 nt
# Database: seq/gst.nlib
# Fields: query id, subject id, % identity, alignment length, mismatches, gap opens, q. start, q. end, s
# 12 hits found
pGT875        pGT875        100.00        657        0        0        1        657        38        694
pGT875        RABGLTR        79.10        646        135        0        1        646        34        6
pGT875        BTGST        59.56        413        167        21        176        594        228
pGT875        RABGSTB        66.93        127        42        8        159        289        157
pGT875        OCDHPR        91.30        23        2        1        266        289        2303        2
...
# FASTA processed 1 queries
```

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("nucleotide_m8CC.txt", "tabular")
```

Information shown in the file header is stored in the `metadata` attribute of `alignments`:

```
>>> alignments.metadata  # doctest: +NORMALIZE_WHITESPACE
{'Command line': 'fasta36 -m 8CC seq/mgstm1.nt seq/gst.nlib',
 'Program': 'FASTA',
 'Version': '36.3.8h May, 2020',
 'Database': 'seq/gst.nlib'}
```

Extract a specific alignment by iterating over the `alignments`. As an example, let's go to the fourth alignment:

```
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> print(alignment)
RABGSTB         156 ????????????????????????????????--????????????????????????
                  0 |||||||||||||||--||||||||||||||||--||||||||||-||||||||||||
pGT875          158 ??????????????????--????????????????????????????????-???????????
<BLANKLINE>
RABGSTB         214 ??????????????????????????????????????????????????????????????-?
                 60 |||||||||||||||||||||||||||||||||||||||||||||||||||||-||||||||||-|
pGT875          215 ??????????????????????????????????????????????????-?????????????
<BLANKLINE>
RABGSTB         273 ??????-???????? 287
                120 ||||||-|||||||| 135
pGT875          274 ???????????????? 289
<BLANKLINE>
>>> alignment.coordinates  # doctest: +NORMALIZE_WHITESPACE
array([[156, 171, 173, 190, 190, 201, 202, 260, 261, 272, 272, 279, 279, 287],
       [158, 173, 173, 190, 192, 203, 203, 261, 261, 272, 273, 280, 281, 289]])
>>> alignment.aligned
array([[[156, 171],
        [173, 190],
        [190, 201],
        [202, 260],
        [261, 272],
        [272, 279],
        [279, 287]],
<BLANKLINE>
       [[158, 173],
        [173, 190],
        [192, 203],
        [203, 261],
        [261, 272],
        [273, 280],
        [281, 289]]])
```

The sequence information of the target and query sequences is stored in the `target` and `query` attributes (as well as under `alignment.sequences`):

```
>>> alignment.target
SeqRecord(seq=Seq(None, length=287), id='RABGSTB', name='<unknown name>', description='<unknown descrip
>>> alignment.query
SeqRecord(seq=Seq(None, length=657), id='pGT875', name='<unknown name>', description='<unknown descripti
```

Information of the alignment is stored under the `annotations` attribute of the `alignment`:

```
>>> alignment.annotations  # doctest: +NORMALIZE_WHITESPACE
{'% identity': 66.93,
 'mismatches': 42,
 'gap opens': 8,
 'evalue': 3.2e-07,
 'bit score': 45.0}
```

BLAST in particular offers many options to customize tabular output by including or excluding specific columns; see the BLAST documentation for details. This information is stored in the dictionaries `alignment.annotations`, `alignment.target.annotations`, or `alignment.query.annotations`, as appropriate.

6.7.10 HH-suite output files

Alignment files in the `hhr` format are generated by `hhsearch` or `hhblits` in HH-suite [44]. As an example, see the file `2uvo_hhblits.hhr` in Biopython's test suite:

```
Query         2UVO:A|PDBID|CHAIN|SEQUENCE
Match_columns 171
No_of_seqs    1560 out of 4005
Neff          8.3
Searched_HMMs 34
Date          Fri Feb 15 16:34:13 2019
Command       hhblits -i 2uvoAh.fasta -d /pdb70

 No Hit                             Prob E-value P-value  Score    SS Cols Query HMM  Template HMM
  1 2uvo_A Agglutinin isolectin 1; 100.0 3.7E-34 4.8E-38  210.3   0.0  171    1-171     1-171 (171)
  2 2wga   ; lectin (agglutinin);   99.9 1.1E-30 1.4E-34  190.4   0.0  162    2-169     2-163 (164)
  3 1ulk_A Lectin-C; chitin-bindin  99.8 5.2E-24 6.6E-28  148.2   0.0  120    1-124     2-121 (126)
...
 31 4z8i_A BBTPGRP3, peptidoglycan  79.6    0.12 1.5E-05   36.1   0.0   37    1-37      9-54  (236)
 32 1wga   ; lectin (agglutinin);   40.4     2.6 0.00029   25.9   0.0  106   54-163    11-116 (164)

No 1
>2uvo_A Agglutinin isolectin 1; carbohydrate-binding protein, hevein domain, chitin-binding, GERM agglu
Probab=99.95  E-value=3.7e-34  Score=210.31  Aligned_cols=171  Identities=100%  Similarity=2.050  Sum_p

Q 2UVO:A|PDBID|C    1 ERCGEQGSNMECPNNLCCSQYGYCGMGGDYCGKGCQNGACWTSKRCGSQAGGATCTNNQCCSQYGYCGFGAEYCGAGCQG
Q Consensus         1 ~~cg~~~~~~c~~~~CCs~~g~CG~~~~~c~~~c~~~~c~~~~~Cg~~~~~~c~~~~CCs~~g~CG~~~~~c~~~c~~
                      ||||++.++..||++.|||+|+|||.+.+||+++||.+.|++..+|+++++.++|....|||.++||+.+.+||+.+||.
T Consensus         1 ~~cg~~~~~~c~~~~CCS~~g~Cg~~~~~Cg~gC~~~~c~~~~~cg~~~~~~c~~~~CCs~~g~Cg~~~~~c~~~c~~
T 2uvo_A           1 ERCGEQGSNMECPNNLCCSQYGYCGMGGDYCGKGCQNGACWTSKRCGSQAGGATCTNNQCCSQYGYCGFGAEYCGAGCQG
T ss_dssp            CBCBGGGTTBBCGGGCEECTTSBEEBSHHHHSTTCCBSSCSSCCBCBGGGTTBCCSTTCEECTTSBEEBSHHHHSTTCCB
T ss_pred            CCCCCCCCCcCCCCCCeeCCCCeECCCccccCCccccccccccccCcccCCcccCCccccCCCceeCCCccccCCCccc
Confidence           799999999999999999999999999999999999999999999999999999999999999999999999999999
```

```
Q 2UVO:A|PDBID|C   81 GPCRADIKCGSQAGGKLCPNNLCCSQWGFCGLGSEFCGGGCQSGACSTDKPCGKDAGGRVCTNNYCCSKWGSCGIGPGYC
Q Consensus        81 ~~~~~~~~Cg~~~~~~~c~~~~CCS~~G~CG~~~~~C~~~Cq~~~c~~~~~Cg~~~~~~~c~~~~CCS~~G~CG~~~~~C
                      +++++|+.|+...+++.||++.|||.|||||...+||+.+||+++|++|.+|++.+++++|..+.|||+++-||+...||
T Consensus        81 ~~~~~~~~cg~~~~~~c~~~~CCs~~g~CG~~~~~C~~gCq~~~c~~~~~cg~~~~~~c~~~~ccs~~g~Cg~~~~~C
T 2uvo_A           81 GPCRADIKCGSQAGGKLCPNNLCCSQWGFCGLGSEFCGGGCQSGACSTDKPCGKDAGGRVCTNNYCCSKWGSCGIGPGYC
T ss_dssp             SSCSSCCBCBGGGTTBCCGGGCEECTTSBEEBSHHHHSTTCCBSSCSSCCCCBTTTTTBCCSTTCEECTTSCEEBSHHHH
T ss_pred             ccccccccccccccCCCCCCCcccCCCCccCCCccccccCCCcCCccccccccccccccccccCCCCCCcCCCCEecCchhhc
Confidence            99999999999988999999999999999999999999999999999999999999999999999999999999999999


Q 2UVO:A|PDBID|C  161 GAGCQSGGCDG  171 (171)
Q Consensus       161 ~~gCq~~~c~~  171 (171)
                      +++||++.|||
T Consensus       161 ~~~cq~~~~~~  171 (171)
T 2uvo_A          161 GAGCQSGGCDG  171 (171)
T ss_dssp             STTCCBSSCC-
T ss_pred             ccccccCCCCC
Confidence            99999999986


No 2
...


No 32
>1wga   ; lectin (agglutinin); NMR {}
Probab=40.43  E-value=2.6  Score=25.90  Aligned_cols=106  Identities=20%  Similarity=0.652  Sum_probs=54

Q 2UVO:A|PDBID|C   54 TCTNNQCCSQYGYCGFGAEYCGAGCQGGPCRADIKCGSQAGGKLCPNNLCCSQWGFCGLGSEFCGGGCQSGACSTDKPCG
Q Consensus        54 ~c~~~~CCs~~g~CG~~~~~c~~~c~~~~~~~~~Cg~~~~~~c~~~~CCS~~G~CG~~~~~C~~~Cq~~~c~~~~~Cg
                      .|....||.....|......|...|....|.....|...  ...|....||.....|......|...|....+.....|.
T Consensus        11 ~c~~~~cc~~~~~c~~~~~c~~~c~~~~c~~~~~c~~~--~~~c~~~~cc~~~~~c~~~~~c~~~c~~~~c~~~~~c~
T 1wga             11 XCXXXXCCXXXXXCXXXXXXCXXXCXXXXCXXXXXCXXX--XXXCXXXXCCXXXXXCXXXXXXCXXXCXXXXCXXXXXCX
T ss_pred             ccccccccccccccccccccccccccccccccccccc--ccccccccccccccccccccccccccccccccccccccccc
Confidence            34455666666666666566555543333223333321  2346666777777777777666666555443322233333


Q 2UVO:A|PDBID|C  134 KDAGGRVCTNNYCCSKWGSCGIGPGYCGAG  163 (171)
Q Consensus       134 ~~~~~~~c~~~~CCS~~G~CG~~~~~C~~g  163 (171)
                      ..  ...|....||.....|......|...
T Consensus        89 ~~--~~~c~~~~cc~~~~~c~~~~~c~~~  116 (164)
T 1wga             89 XX--XXXCXXXXCCXXXXXCXXXXXCXXX  116 (164)
T ss_pred             cc--ccccccccccccccccccccccc
Confidence            22  233444555555555555544433


Done!
```

The file contains three sections:

- A header with general information about the alignments;

- A summary with one line for each of the alignments obtained;
- The alignments shown consecutively in detail.

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("2uvo_hhblits.hhr", "hhr")
```

Most of the header information is stored in the `metadata` attribute of `alignments`:

```
>>> alignments.metadata  # doctest: +NORMALIZE_WHITESPACE
{'Match_columns': 171,
 'No_of_seqs': (1560, 4005),
 'Neff': 8.3,
 'Searched_HMMs': 34,
 'Rundate': 'Fri Feb 15 16:34:13 2019',
 'Command line': 'hhblits -i 2uvoAh.fasta -d /pdb70'}
```

except the query name, which is stored as an attribute:

```
>>> alignments.query_name
'2UVO:A|PDBID|CHAIN|SEQUENCE'
```

as it will reappear in each of the alignments.

Iterate over the alignments:

```
>>> for alignment in alignments:
...     print(alignment.target.id)  # doctest: +ELLIPSIS
...
2uvo_A
2wga
1ulk_A
...
4z8i_A
1wga
```

Let's look at the first alignment in more detail:

```
>>> alignments.rewind()
>>> alignment = next(alignments)
>>> alignment  # doctest: +ELLIPSIS
<Alignment object (2 rows x 171 columns) at ...>
>>> print(alignment)
2uvo_A              0 ERCGEQGSNMECPNNLCCSQYGYCGMGGDYCGKGCQNGACWTSKRCGSQAGGATCTNNQC
                    0 ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
2UVO:A|PD           0 ERCGEQGSNMECPNNLCCSQYGYCGMGGDYCGKGCQNGACWTSKRCGSQAGGATCTNNQC
<BLANKLINE>
2uvo_A             60 CSQYGYCGFGAEYCGAGCQGGPCRADIKCGSQAGGKLCPNNLCCSQWGFCGLGSEFCGGG
                   60 ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
2UVO:A|PD          60 CSQYGYCGFGAEYCGAGCQGGPCRADIKCGSQAGGKLCPNNLCCSQWGFCGLGSEFCGGG
<BLANKLINE>
2uvo_A            120 CQSGACSTDKPCGKDAGGRVCTNNYCCSKWGSCGIGPGYCGAGCQSGGCDG 171
                  120 ||||||||||||||||||||||||||||||||||||||||||||||||||| 171
2UVO:A|PD         120 CQSGACSTDKPCGKDAGGRVCTNNYCCSKWGSCGIGPGYCGAGCQSGGCDG 171
<BLANKLINE>
```

The target and query sequences are stored in `alignment.sequences`. As these are pairwise alignments, we can also access them through `alignment.target` and `alignment.query`:

```
>>> alignment.target is alignment.sequences[0]
True
>>> alignment.query is alignment.sequences[1]
True
```

The ID of the query is set from the `alignments.query_name` (note that the query ID printed in the alignment in the `hhr` file is abbreviated):

```
>>> alignment.query.id
'2UVO:A|PDBID|CHAIN|SEQUENCE'
```

The ID of the target is taken from the sequence alignment block (the line starting with `T 2uvo_A`):

```
>>> alignment.target.id
'2uvo_A'
```

The sequence contents of the target and query are filled in from the information available in this alignment:

```
>>> alignment.target.seq
Seq('ERCGEQGSNMECPNNLCCSQYGYCGMGGDYCGKGCQNGACWTSKRCGSQAGGAT...CDG')
>>> alignment.query.seq
Seq('ERCGEQGSNMECPNNLCCSQYGYCGMGGDYCGKGCQNGACWTSKRCGSQAGGAT...CDG')
```

The sequence contents will be incomplete (a partially defined sequence; see Section 3.12) if the alignment does not extend over the full sequence.

The second line of this alignment block, starting with ">", shows the name and description of the Hidden Markov Model from which the target sequence was taken. These are stored under the keys `"hmm_name"` and `"hmm_description"` in the `alignment.target.annotations` dictionary:

```
>>> alignment.target.annotations  # doctest: +NORMALIZE_WHITESPACE
{'hmm_name': '2uvo_A',
 'hmm_description': 'Agglutinin isolectin 1; carbohydrate-binding protein, hevein domain, chitin-bindin
```

The dictionary `alignment.target.letter_annotations` stores the target alignent consensus sequence, the secondary structure as predicted by PSIPRED, and the target secondary structure as determined by DSSP:

```
>>> alignment.target.letter_annotations  # doctest: +NORMALIZE_WHITESPACE
{'Consensus': '~~cg~~~~~~c~~~~CCS~~g~Cg~~~~~Cg~gC~~~~c~~~~~cg~~~~~~c~~~~CCs~~g~Cg~~~~~c~~~c~~~~~~~~~
 'ss_pred': 'CCCCCCCCCcCCCCCCeeCCCCeECCCccccccCCccccccccccccccCcccCCcccCCccccCCCceeCCCccccCCCccccccccccccc
 'ss_dssp': 'CBCBGGGTTBBCGGGCEECTTSBEEBSHHHHSTTCCBSSCSSCCBCBGGGTTBCCSTTCEECTTSBEEBSHHHHSTTCCBSSCSSCCBCB
```

In this example, for the query sequence only the consensus sequence is available:

```
>>> alignment.query.letter_annotations
{'Consensus': '~~cg~~~~~~c~~~~CCs~~g~CG~~~~~c~~~c~~~~c~~~~~Cg~~~~~~c~~~~CCs~~g~CG~~~~~c~~~c~~~~~~~~~
```

The `alignment.annotations` dictionary stores information about the alignment shown on the third line of the alignment block:

```
>>> alignment.annotations  # doctest: +NORMALIZE_WHITESPACE
{'Probab': 99.95,
 'E-value': 3.7e-34,
 'Score': 210.31,
 'Identities': 100.0,
 'Similarity': 2.05,
 'Sum_probs': 166.9}
```

Confidence values for the pairwise alignment are stored under the `"Confidence"` key in the `alignment.column_annotations` dictionary. This dictionary also stores the score for each column, shown between the query and the target section of each alignment block:

```
>>> alignment.column_annotations  # doctest: +NORMALIZE_WHITESPACE
{'column score': '||||++.++..||++.|||+|+|||.+.+||+++||.+.|++..+|+++++.++|....|||.++||+.+.+||+.+||.+++++
 'Confidence': '79999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
```

6.7.11 A2M

A2M files are alignment files created by `align2model` or `hmmscore` in the SAM Sequence Alignment and Modeling Software System [26, 21]. An A2M file contains one multiple alignment. The A2M file format is similar to aligned FASTA (see section 6.7.1). However, to distinguish insertions from deletions, A2M uses both dashes and periods to represent gaps, and both upper and lower case characters in the aligned sequences. Matches are represented by upper case letters and deletions by dashes in alignment columns containing matches or deletions only. Insertions are represented by lower case letters, with gaps aligned to the insertion shown as periods. Header lines start with ">" followed by the name of the sequence, and optionally a description.

The file `probcons.a2m` in Biopython's test suite is an example of an A2M file (see section 6.7.1 for the same alignment in aligned FASTA format):

```
>plas_horvu
D.VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG.VD.VSKISQEEYLTAPGETFSVTLTV...PGTYGFYCEPHAGAGMVGKVT
V
>plas_chlre
-.VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG.VN.ADAISRDDYLNAPGETYSVKLTA...AGEYGYYCEPHQGAGMVGKII
V
>plas_anava
-.VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKsADlAKSLSHKQLLMSPGQSTSTTFPAdapAGEYTFYCEPHRGAGMVGKIT
V
>plas_proho
VqIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG.ES.APALSNTKLRIAPGSFYSVTLGT...PGTYSFYCTPHRGAGMVGTIT
V
>azup_achcy
VhMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG.AE.A-------FKSKINENYKVTFTA...PGVYGVKCTPHYGMGMVGVVE
V
```

To parse this alignment, use

```
>>> from Bio import Align
>>> alignment = Align.read("probcons.a2m", "a2m")
>>> alignment  # doctest: +ELLIPSIS
<Alignment object (5 rows x 101 columns) at ...>
>>> print(alignment)
plas_horv         0 D-VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG-VD-VSKISQE
plas_chlr         0 --VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG-VN-ADAISRD
plas_anav         0 --VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKSADLAKSLSHK
plas_proh         0 VQIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG-ES-APALSNT
azup_achc         0 VHMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG-AE-A------
<BLANKLINE>
plas_horv        57 EYLTAPGETFSVTLTV---PGTYGFYCEPHAGAGMVGKVTV 95
plas_chlr        56 DYLNAPGETYSVKLTA---AGEYGYYCEPHQGAGMVGKIIV 94
plas_anav        58 QLLMSPGQSTSTTFPADAPAGEYTFYCEPHRGAGMVGKITV 99
```

```
plas_proh        56 KLRIAPGSFYSVTLGT---PGTYSFYCTPHRGAGMVGTITV 94
azup_achc        51 -FKSKINENYKVTFTA---PGVYGVKCTPHYGMGMVGVVEV 88
<BLANKLINE>
```

The parser analyzes the pattern of dashes, periods, and lower and upper case letters in the A2M file to determine if a column is an match/mismatch/deletion (”D”) or an insertion (”I”). This information is stored under the `match` key of the `alignment.column_annotations` dictionary:

```
>>> alignment.column_annotations
{'state': 'DIDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDIDDIDDDDDDDDDDDDDDDDDDDDDDDIIIDDDDDDDDDDDDDDD
```

As the state information is stored in the `alignment`, we can print the alignment in the A2M format:

```
>>> print(format(alignment, "a2m"))
>plas_horvu
D.VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSG.VD.VSKISQEEYLTAPGETFSVTLTV...PGTYGFYCEPHAGAGMVGKVTV
>plas_chlre
-.VKLGADSGALEFVPKTLTIKSGETVNFVNNAGFPHNIVFDEDAIPSG.VN.ADAISRDDYLNAPGETYSVKLTA...AGEYGYYCEPHQGAGMVGKIIV
>plas_anava
-.VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDAALNPAKsADlAKSLSHKQLLMSPGQSTSTTFPAdapAGEYTFYCEPHRGAGMVGKITV
>plas_proho
VqIKMGTDKYAPLYEPKALSISAGDTVEFVMNKVGPHNVIFDK--VPAG.ES.APALSNTKLRIAPGSFYSVTLGT...PGTYSFYCTPHRGAGMVGTITV
>azup_achcy
VhMLNKGKDGAMVFEPASLKVAPGDTVTFIPTDK-GHNVETIKGMIPDG.AE.A-------FKSKINENYKVTFTA...PGVYGVKCTPHYGMGMVGVVEV
<BLANKLINE>
```

Similarly, the alignment can be written in the A2M format to an output file using `Align.write` (see section 6.6.2).

6.7.12 Mauve eXtended Multi-FastA (xmfa) format

Mauve [8] is a software package for constructing multiple genome alignments. These alignments are stored in the eXtended Multi-FastA (xmfa) format. Depending on how exactly `progressiveMauve` (the aligner program in Mauve) was called, the xmfa format is slightly different.

If `progressiveMauve` is called with a single sequence input file, as in

```
progressiveMauve combined.fasta  --output=combined.xmfa ...
```

where `combined.fasta` contains the genome sequences:

```
>equCab1
GAAAAGGAAAGTACGGCCCGGCCACTCCGGGTGTGTGCTAGGAGGGCTTA
>mm9
GAAGAGGAAAAGTAGATCCCTGGCGTCCGGAGCTGGGACGT
>canFam2
CAAGCCCTGCGCGCTCAGCCGGAGTGTCCCGGGCCCTGCTTTCCTTTTC
```

then the output file `combined.xmfa` is as follows:

```
#FormatVersion Mauve1
#Sequence1File        combined.fa
#Sequence1Entry        1
#Sequence1Format        FastA
#Sequence2File        combined.fa
#Sequence2Entry        2
```

```
#Sequence2Format        FastA
#Sequence3File         combined.fa
#Sequence3Entry         3
#Sequence3Format        FastA
#BackboneFile         combined.xmfa.bbcols
> 1:2-49 - combined.fa
AAGCCCTCCTAGCACACACCCGGAGTGG-CCGGGCCGTACTTTCCTTTT
> 2:0-0 + combined.fa
-----------------------------------------------
> 3:2-48 + combined.fa
AAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGCTTTCCTTTT
=
> 1:1-1 + combined.fa
G
=
> 1:50-50 + combined.fa
A
=
> 2:1-41 + combined.fa
GAAGAGGAAAAGTAGATCCCTGGCGTCCGGAGCTGGGACGT
=
> 3:1-1 + combined.fa
C
=
> 3:49-49 + combined.fa
C
=
```

with numbers (1, 2, 3) referring to the input genome sequences for horse (`equCab1`), mouse (`mm9`), and dog (`canFam2`), respectively. This xmfa file consists of six alignment blocks, separated by `=` characters. Use `Align.parse` to extract these alignments:

```
>>> from Bio import Align
>>> alignments = Align.parse("combined.xmfa", "mauve")
```

The file header data are stored in the `metadata` attribute:

```
>>> alignments.metadata  # doctest: +NORMALIZE_WHITESPACE
{'FormatVersion': 'Mauve1',
 'BackboneFile': 'combined.xmfa.bbcols',
 'File': 'combined.fa'}
```

The `identifiers` attribute stores the sequence identifiers for the three sequences, which in this case is the three numbers:

```
>>> alignments.identifiers
['0', '1', '2']
```

These identifiers are used in the individual alignments:

```
>>> for alignment in alignments:
...     print([record.id for record in alignment.sequences])
...     print(alignment)
...     print("******")
...
```

```
['0', '1', '2']
0                49 AAGCCCTCCTAGCACACACCCGGAGTGG-CCGGGCCGTACTTTCCTTTT  1
1                 0 ----------------------------------------------  0
2                 1 AAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGCTTTCCTTTT 48
<BLANKLINE>
******
['0']
0                 0 G 1
<BLANKLINE>
******
['0']
0                49 A 50
<BLANKLINE>
******
['1']
1                 0 GAAGAGGAAAAGTAGATCCCTGGCGTCCGGAGCTGGGACGT 41
<BLANKLINE>
******
['2']
2                 0 C 1
<BLANKLINE>
******
['2']
2                48 C 49
<BLANKLINE>
******
```

Note that only the first block is a real alignment; the other blocks contain only a single sequence. By including these blocks, the xmfa file contains the full sequence that was provided in the `combined.fa` input file.

If `progressiveMauve` is called with a separate input file for each genome, as in

```
progressiveMauve equCab1.fa canFam2.fa mm9.fa --output=separate.xmfa ...
```

where each Fasta file contains the genome sequence for one species only, then the output file `separate.xmfa` is as follows:

```
#FormatVersion Mauve1
#Sequence1File        equCab1.fa
#Sequence1Format        FastA
#Sequence2File        canFam2.fa
#Sequence2Format        FastA
#Sequence3File        mm9.fa
#Sequence3Format        FastA
#BackboneFile        separate.xmfa.bbcols
> 1:1-50 - equCab1.fa
TAAGCCCTCCTAGCACACACCCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC
> 2:1-49 + canFam2.fa
CAAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC
> 3:1-19 - mm9.fa
---------------------------------GGATCTACTTTTCCTCTTC
=
> 3:20-41 + mm9.fa
```

```
CTGGCGTCCGGAGCTGGGACGT
=
```

The identifiers `equCab1` for horse, `mm9` for mouse, and `canFam2` for dog are now shown explicitly in the output file. This xmfa file consists of two alignment blocks, separated by `=` characters. Use `Align.parse` to extract these alignments:

```
>>> from Bio import Align
>>> alignments = Align.parse("separate.xmfa", "mauve")
```

The file header data now does not include the input file name:

```
>>> alignments.metadata  # doctest: +NORMALIZE_WHITESPACE
{'FormatVersion': 'Mauve1',
 'BackboneFile': 'separate.xmfa.bbcols'}
```

The `identifiers` attribute stores the sequence identifiers for the three sequences:

```
>>> alignments.identifiers
['equCab1.fa', 'canFam2.fa', 'mm9.fa']
```

These identifiers are used in the individual alignments:

```
>>> for alignment in alignments:
...     print([record.id for record in alignment.sequences])
...     print(alignment)
...     print("******")
...
['equCab1.fa', 'canFam2.fa', 'mm9.fa']
equCab1.f        50 TAAGCCCTCCTAGCACACACCCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC  0
canFam2.f         0 CAAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC 49
mm9.fa           19 --------------------------------GGATCTACTTTTCCTCTTC  0
<BLANKLINE>
******
['mm9.fa']
mm9.fa           19 CTGGCGTCCGGAGCTGGGACGT 41
<BLANKLINE>
******
```

To output the alignments in Mauve format, use `Align.write`:

```
>>> from io import StringIO
>>> stream = StringIO()
>>> alignments = Align.parse("separate.xmfa", "mauve")
>>> Align.write(alignments, stream, "mauve")
2
>>> print(stream.getvalue())  # doctest: +NORMALIZE_WHITESPACE
#FormatVersion Mauve1
#Sequence1File        equCab1.fa
#Sequence1Format        FastA
#Sequence2File        canFam2.fa
#Sequence2Format        FastA
#Sequence3File        mm9.fa
#Sequence3Format        FastA
#BackboneFile        separate.xmfa.bbcols
```

```
> 1:1-50 - equCab1.fa
TAAGCCCTCCTAGCACACACCCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC
> 2:1-49 + canFam2.fa
CAAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC
> 3:1-19 - mm9.fa
---------------------------------GGATCTACTTTTCCTCTTC
=
> 3:20-41 + mm9.fa
CTGGCGTCCGGAGCTGGGACGT
=
<BLANKLINE>
```

Here, the writer makes use of the information stored in `alignments.metadata` and `alignments.identifiers` to create this format. If your `alignments` object does not have these attributes, you can provide them as keyword arguments to `Align.write`:

```
>>> stream = StringIO()
>>> alignments = Align.parse("separate.xmfa", "mauve")
>>> metadata = alignments.metadata
>>> identifiers = alignments.identifiers
>>> alignments = list(alignments)  # this drops the attributes
>>> alignments.metadata  # doctest: +ELLIPSIS
Traceback (most recent call last):
 ...
AttributeError: 'list' object has no attribute 'metadata'
>>> alignments.identifiers  # doctest: +ELLIPSIS
Traceback (most recent call last):
 ...
AttributeError: 'list' object has no attribute 'identifiers'
>>> Align.write(alignments, stream, "mauve", metadata=metadata, identifiers=identifiers)
2
>>> print(stream.getvalue())  # doctest: +NORMALIZE_WHITESPACE
#FormatVersion Mauve1
#Sequence1File        equCab1.fa
#Sequence1Format        FastA
#Sequence2File        canFam2.fa
#Sequence2Format        FastA
#Sequence3File        mm9.fa
#Sequence3Format        FastA
#BackboneFile        separate.xmfa.bbcols
> 1:1-50 - equCab1.fa
TAAGCCCTCCTAGCACACACCCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC
> 2:1-49 + canFam2.fa
CAAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC
> 3:1-19 - mm9.fa
---------------------------------GGATCTACTTTTCCTCTTC
=
> 3:20-41 + mm9.fa
CTGGCGTCCGGAGCTGGGACGT
=
<BLANKLINE>
```

Python does not allow you to add these attributes to the `alignments` object directly, as in this example it

was converted to a plain list. However, you can construct an `Alignments` object and add attributes to it (see Section 6.5):

```
>>> alignments = Align.Alignments(alignments)
>>> alignments.metadata = metadata
>>> alignments.identifiers = identifiers
>>> stream = StringIO()
>>> Align.write(alignments, stream, "mauve", metadata=metadata, identifiers=identifiers)
2
>>> print(stream.getvalue())  # doctest: +NORMALIZE_WHITESPACE
#FormatVersion Mauve1
#Sequence1File        equCab1.fa
#Sequence1Format        FastA
#Sequence2File        canFam2.fa
#Sequence2Format        FastA
#Sequence3File        mm9.fa
#Sequence3Format        FastA
#BackboneFile        separate.xmfa.bbcols
> 1:1-50 - equCab1.fa
TAAGCCCTCCTAGCACACACCCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC
> 2:1-49 + canFam2.fa
CAAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC
> 3:1-19 - mm9.fa
---------------------------------GGATCTACTTTTCCTCTTC
=
> 3:20-41 + mm9.fa
CTGGCGTCCGGAGCTGGGACGT
=
<BLANKLINE>
```

When printing a single alignment in `Mauve` format, use keyword arguments to provide the metadata and identifiers:

```
>>> alignment = alignments[0]
>>> print(alignment.format("mauve", metadata=metadata, identifiers=identifiers))
> 1:1-50 - equCab1.fa
TAAGCCCTCCTAGCACACACCCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC
> 2:1-49 + canFam2.fa
CAAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC
> 3:1-19 - mm9.fa
---------------------------------GGATCTACTTTTCCTCTTC
=
<BLANKLINE>
```

6.7.13 Sequence Alignment/Map (SAM)

Files in the Sequence Alignment/Map (SAM) format [29] store pairwise sequence alignments, usually of next-generation sequencing data against a reference genome. The file `ex1.sam` in Biopython's test suite is an example of a minimal file in the SAM format. Its first few lines are as follows:

```
EAS56_57:6:190:289:82   69      chr1    100     0       *       =       100     0       CTCAAGGTTGTTGCA
EAS56_57:6:190:289:82   137     chr1    100     73      35M     =       100     0       AGGGGTGCAGAGCCG
EAS51_64:3:190:727:308  99      chr1    103     99      35M     =       263     195     GGTGCAGAGCCGAGT
...
```

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("ex1.sam", "sam")
>>> alignment = next(alignments)
```

The `flag` of the first line is 69. According to the SAM/BAM file format specification, lines for which the flag contains the bitwise flag 4 are unmapped. As 69 has the bit corresponding to this position set to True, this sequence is unmapped and was not aligned to the genome (in spite of the first line showing `chr1`). The target of this alignment (or the first item in `alignment.sequences`) is therefore `None`:

```
>>> alignment.flag
69
>>> bin(69)
'0b1000101'
>>> bin(4)
'0b100'
>>> if alignment.flag & 4:
...     print("unmapped")
... else:
...     print("mapped")
...
unmapped
>>> alignment.sequences
[None, SeqRecord(seq=Seq('CTCAAGGTTGTTGCAAGGGGGTCTATGTGAACAAA'), id='EAS56_57:6:190:289:82', name='<unkn
>>> alignment.target is None
True
```

The second line represents an alignment to chromosome 1:

```
>>> alignment = next(alignments)
>>> if alignment.flag & 4:
...     print("unmapped")
... else:
...     print("mapped")
...
mapped
>>> alignment.target
SeqRecord(seq=None, id='chr1', name='<unknown name>', description='', dbxrefs=[])
```

As this SAM file does not store the genome sequence information for each alignment, we cannot print the alignment. However, we can print the alignment information in SAM format or any other format (such as BED, see section 6.7.14) that does not require the target sequence information:

```
>>> format(alignment, "sam")
'EAS56_57:6:190:289:82\t137\tchr1\t100\t73\t35M\t=\t100\t0\tAGGGGTGCAGAGCCGAGTCACGGGGTTGCCAGCAC\t<<<<<<
>>> format(alignment, "bed")
'chr1\t99\t134\tEAS56_57:6:190:289:82\t0\t+\t99\t134\t0\t1\t35,\t0,\n'
```

However, we cannot print the alignment in PSL format (see section 6.7.16) as that would require knowing the size of the target sequence chr1:

```
>>> format(alignment, "psl")  # doctest: +ELLIPSIS
Traceback (most recent call last):
  ...
TypeError: ...
```

If you know the size of the target sequences, you can set them by hand:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> target = SeqRecord(Seq(None, length=1575), id="chr1")
>>> alignment.target = target
>>> format(alignment, "psl")  # doctest: +ELLIPSIS
'35\t0\t0\t0\t0\t0\t0\t0\t+\tEAS56_57:6:190:289:82\t35\t0\t35\tchr1\t1575\t99\t134\t1\t35,\t0,\t99,\n'
```

The file `ex1_header.sam` in Biopython's test suite contains the same alignments, but now also includes a header. Its first few lines are as follows:

```
@HD\tVN:1.3\tSO:coordinate
@SQ\tSN:chr1\tLN:1575
@SQ\tSN:chr2\tLN:1584
EAS56_57:6:190:289:82   69      chr1    100     0       *       =       100     0       CTCAAGGTTGTTGCA
...
```

The header stores general information about the alignments, including the size of the target chromosomes. The target information is stored in the `targets` attribute of the `alignments` object:

```
>>> from Bio import Align
>>> alignments = Align.parse("ex1_header.sam", "sam")
>>> len(alignments.targets)
2
>>> alignments.targets[0]
SeqRecord(seq=Seq(None, length=1575), id='chr1', name='<unknown name>', description='', dbxrefs=[])
>>> alignments.targets[1]
SeqRecord(seq=Seq(None, length=1584), id='chr2', name='<unknown name>', description='', dbxrefs=[])
```

Other information provided in the header is stored in the `metadata` attribute:

```
>>> alignments.metadata
{'HD': {'VN': '1.3', 'SO': 'coordinate'}}
```

With the target information, we can now also print the alignment in PSL format:

```
>>> alignment = next(alignments)  # the unmapped sequence; skip it
>>> alignment = next(alignments)
>>> format(alignment, "psl")
'35\t0\t0\t0\t0\t0\t0\t0\t+\tEAS56_57:6:190:289:82\t35\t0\t35\tchr1\t1575\t99\t134\t1\t35,\t0,\t99,\n'
```

We can now also print the alignment in human-readable form, but note that the target sequence contents is not available from this file:

```
>>> print(alignment)
chr1             99 ????????????????????????????????????? 134
                   0 .....................................  35
EAS56_57:          0 AGGGGTGCAGAGCCGAGTCACGGGGTTGCCAGCAC  35
<BLANKLINE>
```

Alignments in the file `sam1.sam` in the Biopython test suite contain an additional `MD` tag that shows how the query sequence differs from the target sequence:

```
@SQ     SN:1    LN:239940
@PG     ID:bwa  PN:bwa  VN:0.6.2-r126
HWI-1KL120:88:D0LRBACXX:1:1101:1780:2146        77      *       0       0       *       *       0
...
HWI-1KL120:88:D0LRBACXX:1:1101:2852:2134        137     1       136186  25      101M    =       136186
```

The parser reconstructs the local genome sequence from the `MD` tag, allowing us to see the target sequence explicitly when printing the alignment:

```
>>> from Bio import Align
>>> alignments = Align.parse("sam1.sam", "sam")
>>> for alignment in alignments:
...     if not alignment.flag & 4:  # Skip the unmapped lines
...         break
...
>>> alignment  # doctest: +ELLIPSIS
<Alignment object (2 rows x 101 columns) at ...>
>>> print(alignment)
1            136185 TCACGGTGGCCTGTTGAGGCAGGGGGTCACGCTGACCTCTGTCCGCGTGGGAGGGGCCGG
                 0 |||||||||||||||||||||||||.|||||||||||||||.||.|||||||||||||||||
HWI-1KL12        0 TCACGGTGGCCTGTTGAGGCAGGGGCTCACGCTGACCTCTCTCGGCGTGGGAGGGGCCGG
<BLANKLINE>
1            136245 TGTGAGGCAAGGGCTCACACTGACCTCTCTCAGCGTGGGAG 136286
                60 |||||||||||||||||.||||||||||.|||||||||    101
HWI-1KL12       60 TGTGAGGCAAGGGCTCACGCTGACCTCTCTCGGCGTGGGAG    101
<BLANKLINE>
```

SAM files may include additional information to distinguish simple sequence insertions and deletions from skipped regions of the genome (e.g. introns), hard and soft clipping, and padded sequence regions. As this information cannot be stored in the `coordinates` attribute of an `Alignment` object, and is stored in a dedicated `operations` attribute instead. Let's use the third alignment in this SAM file as an example:

```
>>> from Bio import Align
>>> alignments = Align.parse("dna_rna.sam", "sam")
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> print(format(alignment, "SAM"))  # doctest: +NORMALIZE_WHITESPACE
NR_111921.1       0       chr3       48663768       0       46M1827N82M3376N76M12H       *
<BLANKLINE>
>>> alignment.coordinates
array([[48663767, 48663813, 48665640, 48665722, 48669098, 48669174],
       [       0,       46,       46,      128,      128,      204]])
>>> alignment.operations
bytearray(b'MNMNM')
>>> alignment.query.annotations["hard_clip_right"]
12
```

In this alignment, the cigar string `63M1062N75M468N43M` defines 46 aligned nucleotides, an intron of 1827 nucleotides, 82 aligned nucleotides, an intron of 3376 nucleotides, 76 aligned nucleotides, and 12 hard-clipped nucleotides. These operations are shown in the `operations` attribute, except for hard-clipping, which is stored in `alignment.query.annotations["hard_clip_right"]` (or `alignment.query.annotations["hard_clip_left"]`, if applicable) instead.

To write a SAM file with alignments created from scratch, use an `Alignments` (plural) object (see Section 6.5) to store the alignments as well as the metadata and targets:

```
>>> from io import StringIO
>>> import numpy as np

>>> from Bio import Align
```

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

>>> alignments = Align.Alignments()

>>> seq1 = Seq(None, length=10000)
>>> target1 = SeqRecord(seq1, id="chr1")
>>> seq2 = Seq(None, length=15000)
>>> target2 = SeqRecord(seq2, id="chr2")
>>> alignments.targets = [target1, target2]
>>> alignments.metadata = {"HD": {"VN": "1.3", "SO": "coordinate"}}

>>> seqA = Seq(None, length=20)
>>> queryA = SeqRecord(seqA, id="readA")
>>> sequences = [target1, queryA]
>>> coordinates = np.array([[4300, 4320], [0, 20]])
>>> alignment = Align.Alignment(sequences, coordinates)
>>> alignments.append(alignment)

>>> seqB = Seq(None, length=25)
>>> queryB = SeqRecord(seqB, id="readB")
>>> sequences = [target1, queryB]
>>> coordinates = np.array([[5900, 5925], [25, 0]])
>>> alignment = Align.Alignment(sequences, coordinates)
>>> alignments.append(alignment)

>>> seqC = Seq(None, length=40)
>>> queryC = SeqRecord(seqC, id="readC")
>>> sequences = [target2, queryC]
>>> coordinates = np.array([[12300, 12318], [0, 18]])
>>> alignment = Align.Alignment(sequences, coordinates)
>>> alignments.append(alignment)

>>> stream = StringIO()
>>> Align.write(alignments, stream, "sam")
3
>>> print(stream.getvalue())  # doctest: +NORMALIZE_WHITESPACE
@HD        VN:1.3        SO:coordinate
@SQ        SN:chr1        LN:10000
@SQ        SN:chr2        LN:15000
readA        0        chr1        4301        255        20M        *        0        0        *
readB        16        chr1        5901        255        25M        *        0        0        *
readC        0        chr2        12301        255        18M22S        *        0        0        *
<BLANKLINE>
```

6.7.14 Browser Extensible Data (BED)

BED (Browser Extensible Data) files are typically used to store the alignments of gene transcripts to the genome. See the description from UCSC for a full explanation of the BED format.

BED files have three required fields and nine optional fields. The file `bed12.bed` in subdirectory `Tests/Blat` is an example of a BED file with 12 fields:

```
chr22        1000        5000        mRNA1        960        +        1200        4900        255,0,0
chr22        2000        6000        mRNA2        900        -        2300        5960        0,255,0
```

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("bed12.bed", "bed")
>>> len(alignments)
2
>>> for alignment in alignments:
...     print(alignment.coordinates)
...
[[1000 1567 4512 5000]
 [   0  567  567 1055]]
[[2000 2433 5601 6000]
 [ 832  399  399    0]]
```

Note that the first sequence ("`mRNA1`") was mapped to the forward strand, while the second sequence ("`mRNA2`") was mapped to the reverse strand.

As a BED file does not store the length of each chromosome, the length of the target sequence is set to its maximum:

```
>>> alignment.target
SeqRecord(seq=Seq(None, length=9223372036854775807), id='chr22', name='<unknown name>', description='',
```

The length of the query sequence can be inferred from its alignment information:

```
>>> alignment.query
SeqRecord(seq=Seq(None, length=832), id='mRNA2', name='<unknown name>', description='', dbxrefs=[])
```

The alignment score (field 5) and information stored in fields 7-9 (referred to as `thickStart`, `thickEnd`, and `itemRgb` in the BED format specification) are stored as attributes on the `alignment` object:

```
>>> alignment.score
900.0
>>> alignment.thickStart
2300
>>> alignment.thickEnd
5960
>>> alignment.itemRgb
'0,255,0'
```

To print an alignment in the BED format, you can use Python's built-in `format` function:

```
>>> print(format(alignment, "bed"))  # doctest: +NORMALIZE_WHITESPACE
chr22        2000        6000        mRNA2        900        -        2300        5960        0,255,0
<BLANKLINE>
```

or you can use the `format` method of the `alignment` object. This allows you to specify the number of fields to be written as the `bedN` keyword argument:

```
>>> print(alignment.format("bed"))  # doctest: +NORMALIZE_WHITESPACE
chr22        2000        6000        mRNA2        900        -        2300        5960        0,255,0
<BLANKLINE>
>>> print(alignment.format("bed", 3))  # doctest: +NORMALIZE_WHITESPACE
chr22        2000        6000
```

```
<BLANKLINE>
>>> print(alignment.format("bed", 6))  # doctest: +NORMALIZE_WHITESPACE
chr22        2000        6000        mRNA2        900        -
<BLANKLINE>
```

The same keyword argument can be used with `Align.write`:

```
>>> alignments.rewind()
>>> Align.write(alignments, "mybed3file.bed", "bed", bedN=3)
2
>>> alignments.rewind()
>>> Align.write(alignments, "mybed6file.bed", "bed", bedN=6)
2
>>> alignments.rewind()
>>> Align.write(alignments, "mybed12file.bed", "bed")
2
```

6.7.15 bigBed

The bigBed file format is an indexed binary version of a BED file 6.7.14. To create a bigBed file, you can either use the `bedToBigBed` program from UCSC (). or you can use Biopython for it by calling the `Bio.Align.write` function with `fmt="bigbed"`. While the two methods should result in identical bigBed files, using `bedToBigBed` is much faster and may be more reliable, as it is the gold standard. As bigBed files come with a built-in index, it allows you to quickly search a specific genomic region.

As an example, let's parse the bigBed file `dna_rna.bb`, available in the `Tests/Blat` subdirectory in the Biopython distribution:

```
>>> from Bio import Align
>>> alignments = Align.parse("dna_rna.bb", "bigbed")
>>> len(alignments)
4
>>> print(alignments.declaration)  # doctest: +NORMALIZE_WHITESPACE
table bed
"Browser Extensible Data"
(
   string          chrom;          "Reference sequence chromosome or scaffold"
   uint            chromStart;     "Start position in chromosome"
   uint            chromEnd;       "End position in chromosome"
   string          name;           "Name of item."
   uint            score;          "Score (0-1000)"
   char[1]         strand;         "+ or - for strand"
   uint            thickStart;     "Start of where display should be thick (start codon)"
   uint            thickEnd;       "End of where display should be thick (stop codon)"
   uint            reserved;       "Used as itemRgb as of 2004-11-22"
   int             blockCount;     "Number of blocks"
   int[blockCount] blockSizes;     "Comma separated list of block sizes"
   int[blockCount] chromStarts;    "Start positions relative to chromStart"
)
<BLANKLINE>
```

The `declaration` contains the specification of the columns, in AutoSql format, that was used to create the bigBed file. Target sequences (typically, the chromosomes against which the sequences were aligned) are stored in the `targets` attribute. In the bigBed format, only the identifier and the size of each target is stored. In this example, there is only a single chromosome:

```
>>> alignments.targets
[SeqRecord(seq=Seq(None, length=198295559), id='chr3', name='<unknown name>', description='<unknown des
```

Let's look at the individual alignments. The alignment information is stored in the same way as for a BED file (see section 6.7.14):

```
>>> alignment = next(alignments)
>>> alignment.target.id
'chr3'
>>> alignment.query.id
'NR_046654.1'
>>> alignment.coordinates
array([[42530895, 42530958, 42532020, 42532095, 42532563, 42532606],
       [     181,      118,      118,       43,       43,        0]])
>>> alignment.thickStart
42530895
>>> alignment.thickEnd
42532606
>>> print(alignment)  # doctest: +ELLIPSIS
chr3       42530895 ??????????????????????????????????????????????????????????????
                 0 ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
NR_046654      181 ??????????????????????????????????????????????????????????????
<BLANKLINE>
chr3       42530955 ??????????????????????????????????????????????????????????????
                60 |||-----------------------------------------------------------
NR_046654      121 ???-----------------------------------------------------------
...
chr3       42532515 ??????????????????????????????????????????????????????????????
              1620 ------------------------------------------------------||||||||||||
NR_046654       43 ------------------------------------------------------??????????
<BLANKLINE>
chr3       42532575 ????????????????????????????????? 42532606
              1680 |||||||||||||||||||||||||||||||||     1711
NR_046654       31 ?????????????????????????????????        0
<BLANKLINE>
```

The default bigBed format does not store the sequence contents of the target and query. If these are available elsewhere (for example, a Fasta file), you can set `alignment.target.seq` and `alignment.query.seq` to show the sequence contents when printing the alignment, or to write the alignment in formats that require the sequence contents (such as Clustal, see section 6.7.2). The test script `test_Align_bigbed.py` in the `Tests` subdirectory in the Biopython distribution gives some examples on how to do that.

Now let's see how to search for a sequence region. These are the sequences stored in the bigBed file, printed in BED format (see section 6.7.14):

```
>>> alignments.rewind()
>>> for alignment in alignments:
...     print(format(alignment, "bed"))  # doctest: +NORMALIZE_WHITESPACE
...
chr3        42530895        42532606        NR_046654.1        1000        -        42530895        4253
<BLANKLINE>
chr3        42530895        42532606        NR_046654.1_modified        978        -        42530895
<BLANKLINE>
chr3        48663767        48669174        NR_111921.1        1000        +        48663767        486
```

```
<BLANKLINE>
chr3        48663767        48669174        NR_111921.1_modified        972        +        48663767
<BLANKLINE>
```

Use the `search` method on the `alignments` object to find regions on chr3 between positions 48000000 and 49000000. This method returns an iterator:

```
>>> selected_alignments = alignments.search("chr3", 48000000, 49000000)
>>> for alignment in selected_alignments:
...     print(alignment.query.id)
...
NR_111921.1
NR_111921.1_modified
```

The chromosome name may be `None` to include all chromosomes, and the start and end positions may be `None` to start searching from position 0 or to continue searching until the end of the chromosome, respectively.

Writing alignments in the bigBed format is as easy as calling `Bio.Align.write`:

```
>>> Align.write(alignments, "output.bb", "bigbed")
```

You can specify the number of BED fields to be included in the bigBed file. For example, to write a BED6 file, use

```
>>> Align.write(alignments, "output.bb", "bigbed", bedN=6)
```

Same as for `bedToBigBed`, you can include additional columns in the bigBed output. Suppose the file `bedExample2.as` (available in the `Tests/Blat` subdirectory of the Biopython distribution) stores the declaration of the included BED fields in AutoSql format. We can read this declaration as follows:

```
>>> from Bio.Align import bigbed
>>> with open("bedExample2.as") as stream:
...     autosql_data = stream.read()
...
>>> declaration = bigbed.AutoSQLTable.from_string(autosql_data)
>>> type(declaration)
<class 'Bio.Align.bigbed.AutoSQLTable'>
>>> print(declaration)
table hg18KGchr7
"UCSC Genes for chr7 with color plus GeneSymbol and SwissProtID"
(
   string  chrom;         "Reference sequence chromosome or scaffold"
   uint    chromStart;    "Start position of feature on chromosome"
   uint    chromEnd;      "End position of feature on chromosome"
   string  name;          "Name of gene"
   uint    score;         "Score"
   char[1] strand;        "+ or - for strand"
   uint    thickStart;    "Coding region start"
   uint    thickEnd;      "Coding region end"
   uint    reserved;      "Green on + strand, Red on - strand"
   string  geneSymbol;    "Gene Symbol"
   string  spID;          "SWISS-PROT protein Accession number"
)
<BLANKLINE>
```

Now we can write a bigBed file with the 9 BED fields plus the additional fields `geneSymbol` and `spID` by calling

```
>>> Align.write(
...     alignments,
...     "output.bb",
...     "bigbed",
...     bedN=9,
...     declaration=declaration,
...     extraIndex=["name", "geneSymbol"],
... )
```

Here, we also requested to include additional indices on the `name` and `geneSymbol` in the bigBed file. `Align.write` expects to find the keys `geneSymbol` and `spID` in the `alignment.annotations` dictionary. Please refer to the test script `test_Align_bigbed.py` in the `Tests` subdirectory in the Biopython distribution for more examples of writing alignment files in the bigBed format.

6.7.16 Pattern Space Layout (PSL)

PSL (Pattern Space Layout) files are are generated by the BLAST-Like Alignment Tool BLAT [24]. Like BED files (see section 6.7.14), PSL files are typically used to store alignments of transcripts to genomes. This is an example of a short BLAT file (available as `dna_rna.psl` in the `Tests/Blat` subdirectory of the Biopython distribution), with the standard PSL header consisting of 5 lines:

```
psLayout version 3

match       mis-        rep.        N's       Q gap       Q gap       T gap       T gap       s
            match       match                 count       bases       count       bases
-----------------------------------------------------------------------------------------------
165         0           39          0         0           0           2           5203        +       NR_111921.1
175         0           6           0         0           0           2           1530        -       NR_046654.1
162         2           39          0         1           2           3           5204        +       NR_111921.1_modi
172         1           6           0         1           3           3           1532        -       NR_046654.1_modif
```

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("dna_rna.psl", "psl")
>>> alignments.metadata
{'psLayout version': '3'}
```

Iterate over the alignments to get one `Alignment` object for each line:

```
>>> for alignment in alignments:
...     print(alignment.target.id, alignment.query.id)
...
chr3 NR_046654.1
chr3 NR_046654.1_modified
chr3 NR_111921.1
chr3 NR_111921.1_modified
```

Let's look at the last alignment in more detail. The first four columns in the PSL file show the number of matches, the number of mismatches, the number of nucleotides aligned to repeat regions, and the number of nucleotides aligned to N (unknown) characters. These values are stored as attributes to the `Alignment` object:

```
>>> alignment.matches
162
```

```
>>> alignment.misMatches
2
>>> alignment.repMatches
39
>>> alignment.nCount
0
```

As the sequence data of the target and query are not stored explicitly in the PSL file, the sequence content of `alignment.target` and `alignment.query` is undefined. However, their sequence lengths are known:

```
>>> alignment.target  # doctest: +ELLIPSIS
SeqRecord(seq=Seq(None, length=198295559), id='chr3', ...)
>>> alignment.query  # doctest: +ELLIPSIS
SeqRecord(seq=Seq(None, length=220), id='NR_111921.1_modified', ...)
```

We can print the alignment in BED or PSL format:

```
>>> print(format(alignment, "bed"))  # doctest: +NORMALIZE_WHITESPACE
chr3        48663767        48669174        NR_111921.1_modified        0        +        48663767
<BLANKLINE>
>>> print(format(alignment, "psl"))  # doctest: +NORMALIZE_WHITESPACE
162        2        39        0        1        2        3        5204        +        NR_111921.1_modi
<BLANKLINE>
```

Here, the number of matches, mismatches, repeat region matches, and matches to unknown nucleotides were taken from the corresponding attributes of the `Alignment` object. If these attributes are not available, for example if the alignment did not come from a PSL file, then these numbers are calculated using the sequence contents, if available. Repeat regions in the target sequence are indicated by masking the sequence as lower-case or upper-case characters, as defined by the following values for the `mask` keyword argument:

- `False` (default): Do not count matches to masked sequences separately;
- `"lower"`: Count and report matches to lower-case characters as matches to repeat regions;
- `"upper"`: Count and report matches to upper-case characters as matches to repeat regions;

The character used for unknown nucleotides is defined by the `wildcard` argument. For consistency with BLAT, the wildcard character is `"N"` by default. Use `wildcard=None` if you don't want to count matches to any unknown nucleotides separately.

```
>>> import numpy
>>> from Bio import Align
>>> query = "GGTGGGGG"
>>> target = "AAAAAAAggggGGNGAAAAA"
>>> coordinates = numpy.array([[0, 7, 15, 20], [0, 0, 8, 8]])
>>> alignment = Align.Alignment([target, query], coordinates)
>>> print(alignment)
target            0 AAAAAAAggggGGNGAAAAA 20
                  0 -------....||.|----- 20
query             0 -------GGTGGGGG-----  8
<BLANKLINE>
>>> line = alignment.format("psl")
>>> print(line)  # doctest: +NORMALIZE_WHITESPACE
6   1   0   1   0   0   0   0   +   query   8   0   8   target   20   7   15   1   8,   0,   7,
>>> line = alignment.format("psl", mask="lower")
```

```
>>> print(line)  # doctest: +NORMALIZE_WHITESPACE
3   1   3   1   0   0   0   0   +   query   8   0   8   target   20   7   15   1   8,   0,   7,
>>> line = alignment.format("psl", mask="lower", wildcard=None)
>>> print(line)  # doctest: +NORMALIZE_WHITESPACE
3   2   3   0   0   0   0   0   +   query   8   0   8   target   20   7   15   1   8,   0,   7,
```

The same arguments can be used when writing alignments to an output file in PSL format using `Bio.Align.write`. This function has an additional keyword `header` (`True` by default) specifying if the PSL header should be written.

In addition to the `format` method, you can use Python's built-in `format` function:

```
>>> print(format(alignment, "psl"))  # doctest: +NORMALIZE_WHITESPACE
6   1   0   1   0   0   0   0   +   query   8   0   8   target   20   7   15   1   8,   0,   7,
```

allowing `Alignment` objects to be used in formatted (f-) strings in Python:

```
>>> line = f"The alignment in PSL format is '{alignment:psl}'."
>>> print(line)  # doctest: +NORMALIZE_WHITESPACE
The alignment in PSL format is '6   1   0   1   0   0   0   0   +   query   8   0   8   target   20   7
'
```

Note that optional keyword arguments cannot be used with the `format` function or with formatted strings.

6.7.17 bigPsl

A bigPsl file is a bigBed file with a BED12+13 format consisting of the 12 predefined BED fields and 13 custom fields defined in the AutoSql file bigPsl.as provided by UCSC, creating an indexed binary version of a PSL file (see section 6.7.16). To create a bigPsl file, you can either use the `pslToBigPsl` and `bedToBigBed` programs from UCSC. or you can use Biopython by calling the `Bio.Align.write` function with `fmt="bigpsl"`. While the two methods should result in identical bigPsl files, the UCSC tools are much faster and may be more reliable, as it is the gold standard. As bigPsl files are bigBed files, they come with a built-in index, allowing you to quickly search a specific genomic region.

As an example, let's parse the bigBed file `dna_rna.psl.bb`, available in the `Tests/Blat` subdirectory in the Biopython distribution. This file is the bigPsl equivalent of the bigBed file `dna_rna.bb` (see section 6.7.15) and of the PSL file `dna_rna.psl` (see section 6.7.16).

```
>>> from Bio import Align
>>> alignments = Align.parse("dna_rna.psl.bb", "bigpsl")
>>> len(alignments)
4
>>> print(alignments.declaration)  # doctest: +NORMALIZE_WHITESPACE
table bigPsl
"bigPsl pairwise alignment"
(
   string          chrom;          "Reference sequence chromosome or scaffold"
   uint            chromStart;     "Start position in chromosome"
   uint            chromEnd;       "End position in chromosome"
   string          name;           "Name or ID of item, ideally both human readable and unique"
   uint            score;          "Score (0-1000)"
   char[1]         strand;         "+ or - indicates whether the query aligns to the + or - strand on t
   uint            thickStart;     "Start of where display should be thick (start codon)"
   uint            thickEnd;       "End of where display should be thick (stop codon)"
   uint            reserved;       "RGB value (use R,G,B string in input file)"
   int             blockCount;     "Number of blocks"
```

```
   int[blockCount] blockSizes;      "Comma separated list of block sizes"
   int[blockCount] chromStarts;     "Start positions relative to chromStart"
   uint            oChromStart;     "Start position in other chromosome"
   uint            oChromEnd;       "End position in other chromosome"
   char[1]         oStrand;         "+ or -, - means that psl was reversed into BED-compatible coordina
   uint            oChromSize;      "Size of other chromosome."
   int[blockCount] oChromStarts;    "Start positions relative to oChromStart or from oChromStart+oChrom
   lstring         oSequence;       "Sequence on other chrom (or edit list, or empty)"
   string          oCDS;            "CDS in NCBI format"
   uint            chromSize;       "Size of target chromosome"
   uint            match;           "Number of bases matched."
   uint            misMatch;        "Number of bases that don't match"
   uint            repMatch;        "Number of bases that match but are part of repeats"
   uint            nCount;          "Number of 'N' bases"
   uint            seqType;         "0=empty, 1=nucleotide, 2=amino_acid"
)
<BLANKLINE>
```

The declaration contains the specification of the columns as defined by the `bigPsl.as` AutoSql file from UCSC. Target sequences (typically, the chromosomes against which the sequences were aligned) are stored in the `targets` attribute. In the bigBed format, only the identifier and the size of each target is stored. In this example, there is only a single chromosome:

```
>>> alignments.targets
[SeqRecord(seq=Seq(None, length=198295559), id='chr3', name='<unknown name>', description='<unknown des
```

Iterating over the alignments gives one Alignment object for each line:

```
>>> for alignment in alignments:
...     print(alignment.target.id, alignment.query.id)
...
chr3 NR_046654.1
chr3 NR_046654.1_modified
chr3 NR_111921.1
chr3 NR_111921.1_modified
```

Let's look at the individual alignments. The alignment information is stored in the same way as for the corresponding PSL file (see section 6.7.16):

```
>>> alignment.coordinates
array([[48663767, 48663795, 48663796, 48663813, 48665640, 48665716,
        48665716, 48665722, 48669098, 48669174],
       [       3,       31,       31,       48,       48,      124,
             126,      132,      132,      208]])
>>> alignment.thickStart
48663767
>>> alignment.thickEnd
48669174
>>> alignment.matches
162
>>> alignment.misMatches
2
>>> alignment.repMatches
39
```

```
>>> alignment.nCount
0
```

We can print the alignment in BED or PSL format:

```
>>> print(format(alignment, "bed"))  # doctest: +NORMALIZE_WHITESPACE
chr3        48663767        48669174        NR_111921.1_modified        1000        +        48663767
<BLANKLINE>
>>> print(format(alignment, "psl"))  # doctest: +NORMALIZE_WHITESPACE
162        2        39        0        1        2        3        5204        +        NR_111921.1_modi
<BLANKLINE>
```

As a bigPsl file is a special case of a bigBed file, you can use the `search` method on the alignments object to find alignments to specific genomic regions. For example, we can look for regions on chr3 between positions 48000000 and 49000000:

```
>>> selected_alignments = alignments.search("chr3", 48000000, 49000000)
>>> for alignment in selected_alignments:
...     print(alignment.query.id)
...
NR_111921.1
NR_111921.1_modified
```

The chromosome name may be `None` to include all chromosomes, and the start and end positions may be `None` to start searching from position 0 or to continue searching until the end of the chromosome, respectively.

To write a bigPsl file with Biopython, use `Bio.Align.write(alignments, "myfilename.bb", fmt="bigpsl")`, where `myfilename.bb` is the name of the output bigPsl file. Alternatively, you can use a (binary) stream for output. Additional options are

- `compress`: If `True` (default), compress data using zlib; if `False`, do not compress data.
- `extraIndex`: List of strings with the names of extra columns to be indexed.
- `cds`: If `True`, look for a query feature of type CDS and write it in NCBI style in the PSL file (default: `False`).
- `fa`: If `True`, include the query sequence in the PSL file (default: `False`).
- `mask`: Specify if repeat regions in the target sequence are masked and should be reported in the `repMatches` field instead of in the `matches` field. Acceptable values are
  - `None`: no masking (default);
  - `"lower"`: masking by lower-case characters;
  - `"upper"`: masking by upper-case characters.
- `wildcard`: Report alignments to the wildcard character (representing unknown nucleotides) in the target or query sequence in the `nCount` field instead of in the `matches`, `misMatches`, or `repMatches` fields. Default value is `"N"`.

See section 6.7.16 for an explanation on how the number of matches, mismatches, repeat region matches, and matches to unknown nucleotides are obtained.

6.7.18 Multiple Alignment Format (MAF)

MAF (Multiple Alignment Format) files store a series of multiple sequence alignments in a human-readable format. MAF files are typically used to store alignment s of genomes to each other. The file `ucsc_test.maf` in the `Tests/MAF` subdirectory of the Biopython distribution is an example of a simple MAF file:

```
track name=euArc visibility=pack mafDot=off frames="multiz28wayFrames" speciesOrder="hg16 panTro1 baboo
##maf version=1 scoring=tba.v8
# tba.v8 (((human chimp) baboon) (mouse rat))
# multiz.v7
# maf_project.v5 _tba_right.maf3 mouse _tba_C
# single_cov2.v4 single_cov2 /dev/stdin

a score=23262.0
s hg16.chr7    27578828 38 + 158545518 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s panTro1.chr6 28741140 38 + 161576975 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s baboon         116834 38 +   4622798 AAA-GGGAATGTTAACCAAATGA---GTTGTCTCTTATGGTG
s mm4.chr6     53215344 38 + 151104725 -AATGGGAATGTTAAGCAAACGA---ATTGTCTCTCAGTGTG
s rn3.chr4     81344243 40 + 187371129 -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCAATGTG

a score=5062.0
s hg16.chr7    27699739 6 + 158545518 TAAAGA
s panTro1.chr6 28862317 6 + 161576975 TAAAGA
s baboon         241163 6 +   4622798 TAAAGA
s mm4.chr6     53303881 6 + 151104725 TAAAGA
s rn3.chr4     81444246 6 + 187371129 taagga

a score=6636.0
s hg16.chr7    27707221 13 + 158545518 gcagctgaaaaca
s panTro1.chr6 28869787 13 + 161576975 gcagctgaaaaca
s baboon         249182 13 +   4622798 gcagctgaaaaca
s mm4.chr6     53310102 13 + 151104725 ACAGCTGAAAATA
```

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("ucsc_test.maf", "maf")
```

Information shown in the file header (the track line and subsequent lines starting with "`#`")) is stored in the `metadata` attribute of the `alignments` object:

```
>>> alignments.metadata  # doctest: +NORMALIZE_WHITESPACE
{'name': 'euArc',
 'visibility': 'pack',
 'mafDot': 'off',
 'frames': 'multiz28wayFrames',
 'speciesOrder': ['hg16', 'panTro1', 'baboon', 'mm4', 'rn3'],
 'description': 'A sample alignment',
 'MAF Version': '1',
 'Scoring': 'tba.v8',
 'Comments': ['tba.v8 (((human chimp) baboon) (mouse rat))',
              'multiz.v7',
              'maf_project.v5 _tba_right.maf3 mouse _tba_C',
              'single_cov2.v4 single_cov2 /dev/stdin']}
```

By iterating over the `alignments` we obtain one `Alignment` object for each alignment block in the MAF file:

```
>>> alignment = next(alignments)
>>> alignment.score
23262.0
>>> {seq.id: len(seq) for seq in alignment.sequences}  # doctest: +NORMALIZE_WHITESPACE
{'hg16.chr7': 158545518,
 'panTro1.chr6': 161576975,
 'baboon': 4622798,
 'mm4.chr6': 151104725,
 'rn3.chr4': 187371129}
>>> alignment.coordinates  # doctest: +NORMALIZE_WHITESPACE
array([[27578828, 27578829, 27578831, 27578831, 27578850, 27578850, 27578866],
       [28741140, 28741141, 28741143, 28741143, 28741162, 28741162, 28741178],
       [  116834,   116835,   116837,   116837,   116856,   116856, 116872],
       [53215344, 53215344, 53215346, 53215347, 53215366, 53215366, 53215382],
       [81344243, 81344243, 81344245, 81344245, 81344264, 81344267, 81344283]])
>>> print(alignment)
hg16.chr7  27578828 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG 27578866
panTro1.c  28741140 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG 28741178
baboon       116834 AAA-GGGAATGTTAACCAAATGA---GTTGTCTCTTATGGTG   116872
mm4.chr6   53215344 -AATGGGAATGTTAAGCAAACGA---ATTGTCTCTCAGTGTG 53215382
rn3.chr4   81344243 -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCAATGTG 81344283
<BLANKLINE>
>>> print(format(alignment, "phylip"))
5 42
hg16.chr7 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
panTro1.chAAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
baboon    AAA-GGGAATGTTAACCAAATGA---GTTGTCTCTTATGGTG
mm4.chr6  -AATGGGAATGTTAAGCAAACGA---ATTGTCTCTCAGTGTG
rn3.chr4  -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCAATGTG
<BLANKLINE>
```

In addition to the "`a`" (alignment block) and "`s`" (sequence) lines, MAF files may contain "`i`" lines with information about the genome sequence before and after this block, "`e`" lines with information about empty parts of the alignment, and "`q`" lines showing the quality of each aligned base. This is an example of an alignment block including such lines:

```
a score=19159.000000
s mm9.chr10                          3014644 45 + 129993255 CCTGTACC---CTTTGGTGAGAATTTTTGTTTCAGTGTTAAAAG
s hg18.chr6                         15870786 46 - 170899992 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAAC-
i hg18.chr6                         I 9085 C 0
s panTro2.chr6                      16389355 46 - 173908612 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAAC-
q panTro2.chr6                                              99999999999999999999999-999999999999999999-
i panTro2.chr6                      I 9106 C 0
s calJac1.Contig6394                    6182 46 +    133105 CCTATACCTTTCTTTCATGAGAA-TTTTGTTTGAATCCTAAAC-
i calJac1.Contig6394                N 0 C 0
s loxAfr1.scaffold_75566                1167 34 -     10574 ------------TTTGGTTAGAA-TTATGCTTTAATTCAAAAC-
q loxAfr1.scaffold_75566                                    ------------99999699899-999999999999869998-
i loxAfr1.scaffold_75566            N 0 C 0
e tupBel1.scaffold_114895.1-498454   167376 4145 -    498454 I
e echTel1.scaffold_288249             87661 7564 +    100002 I
e otoGar1.scaffold_334.1-359464      181217 2931 -    359464 I
```

```
e ponAbe2.chr6                    16161448 8044 - 174210431 I
```

This is the 10th alignment block in the file `ucsc_mm9_chr10.maf` (available in the `Tests/MAF` subdirectory of the Biopython distribution):

```
>>> from Bio import Align
>>> alignments = Align.parse("ucsc_mm9_chr10.maf", "maf")
>>> for i in range(10):
...     alignment = next(alignments)
...
>>> alignment.score
19159.0
>>> print(alignment)
mm9.chr10   3014644 CCTGTACC---CTTTGGTGAGAATTTTTGTTTCAGTGTTAAAAGTTTG   3014689
hg18.chr6 155029206 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAAC-TTTT 155029160
panTro2.c 157519257 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAAC-TTTT 157519211
calJac1.C      6182 CCTATACCTTTCTTTCATGAGAA-TTTTGTTTGAATCCTAAAC-TTTT      6228
loxAfr1.s      9407 ------------TTTGGTTAGAA-TTATGCTTTAATTCAAAAC-TTCC      9373
<BLANKLINE>
```

The "`i`" lines show the relationship between the sequence in the current alignment block to the ones in the preceding and subsequent alignment block. This information is stored in the `annotations` attribute of the corresponding sequence:

```
>>> alignment.sequences[0].annotations
{}
>>> alignment.sequences[1].annotations
{'leftStatus': 'I', 'leftCount': 9085, 'rightStatus': 'C', 'rightCount': 0}
```

showing that there are 9085 bases inserted ("`I`") between this block and the preceding one, while the block is contiguous ("`C`") with the subsequent one. See the UCSC documentation for the full description of these fields and status characters.

The "`q`" lines show the sequence quality, which is stored under the "`quality`" dictionary key of the`annotations` attribute of the corresponding sequence:

```
>>> alignment.sequences[2].annotations["quality"]
'99999999999999999999999999999999999999999999999'
>>> alignment.sequences[4].annotations["quality"]
'99999698989999999999999998699989997'
```

The "`e`" lines show information about species with a contiguous sequence before and after this alignment bloack, but with no aligning nucleotides in this alignment block. This is stored under the "`empty`" key of the `alignment.annotations` dictionary:

```
>>> alignment.annotations["empty"]  # doctest: +NORMALIZE_WHITESPACE
[(SeqRecord(seq=Seq(None, length=498454), id='tupBel1.scaffold_114895.1-498454', name='', description='
 (SeqRecord(seq=Seq(None, length=100002), id='echTel1.scaffold_288249', name='', description='', dbxrefs
 (SeqRecord(seq=Seq(None, length=359464), id='otoGar1.scaffold_334.1-359464', name='', description='', d
 (SeqRecord(seq=Seq(None, length=174210431), id='ponAbe2.chr6', name='', description='', dbxrefs=[]), (
```

This shows for example that there were non-aligning bases inserted ("`I`") from position 158040939 to 158048983 on the opposite strand of the `ponAbe2.chr6` genomic sequence. Again, see the UCSC documentation for the full definition of "`e`" lines.

To print an alignment in MAF format, you can use Python's built-in `format` function:

```
>>> print(format(alignment, "MAF"))
a score=19159.000000
s mm9.chr10                          3014644   45 + 129993255 CCTGTACC---CTTTGGTGAGAATTTTTGTTTCAGTGTTAAA
s hg18.chr6                         15870786   46 - 170899992 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAA
i hg18.chr6                         I 9085 C 0
s panTro2.chr6                      16389355   46 - 173908612 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAA
q panTro2.chr6                                                 99999999999999999999999-99999999999999999
i panTro2.chr6                      I 9106 C 0
s calJac1.Contig6394                    6182   46 +    133105 CCTATACCTTTCTTTCATGAGAA-TTTTGTTTGAATCCTAAA
i calJac1.Contig6394                N 0 C 0
s loxAfr1.scaffold_75566                1167   34 -     10574 ------------TTTGGTTAGAA-TTATGCTTTAATTCAAAA
q loxAfr1.scaffold_75566                                       ------------99999699899-99999999999986999
i loxAfr1.scaffold_75566            N 0 C 0
e tupBel1.scaffold_114895.1-498454   167376 4145 -    498454 I
e echTel1.scaffold_288249             87661 7564 +    100002 I
e otoGar1.scaffold_334.1-359464      181217 2931 -    359464 I
e ponAbe2.chr6                      16161448 8044 - 174210431 I
<BLANKLINE>
<BLANKLINE>
```

To write a complete MAF file, use `Bio.Align.write(alignments, "myfilename.maf", fmt="maf")`, where `myfilename.maf` is the name of the output MAF file. Alternatively, you can use a (text) stream for output. File header information will be taken from the `metadata` attribute of the `alignments` object. If you are creating the alignments from scratch, you can use the `Alignments` (plural) class to create a list-like `alignments` object (see Section 6.5) and give it a `metadata` attribute.

6.7.19 bigMaf

A bigMaf file is a bigBed file with a BED3+1 format consisting of the 3 required BED fields plus a custom field that stores a MAF alignment block as a string, crearing an indexed binary version of a MAF file (see section 6.7.19). The associated AutoSql file bigMaf.as is provided by UCSC. To create a bigMaf file, you can either use the `mafToBigMaf` and `bedToBigBed` programs from UCSC. or you can use Biopython by calling the Bio.Align.write function with `fmt="bigmaf"`. While the two methods should result in identical bigMaf files, the UCSC tools are much faster and may be more reliable, as it is the gold standard. As bigMaf files are bigBed files, they come with a built-in index, allowing you to quickly search a specific region of the reference genome.

The file `ucsc_test.bb` in the `Tests/MAF` subdirectory of the Biopython distribution is an example of a bigMaf file. This file is equivalent to the MAF file `ucsc_test.maf` (see section 6.7.18). To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("ucsc_test.bb", "bigmaf")
>>> len(alignments)
3
>>> print(alignments.declaration)  # doctest: +NORMALIZE_WHITESPACE
table bedMaf
"Bed3 with MAF block"
(
   string  chrom;         "Reference sequence chromosome or scaffold"
   uint    chromStart;    "Start position in chromosome"
   uint    chromEnd;      "End position in chromosome"
   lstring mafBlock;      "MAF block"
```

```
)
<BLANKLINE>
```

The declaration contains the specification of the columns as defined by the bigMaf.as AutoSql file from UCSC.

The bigMaf file does not store the header information found in the MAF file, but it does define a reference genome. The corresponding `SeqRecord` is stored in the `targets` attribute of the `alignments` object:

```
>>> alignments.reference
'hg16'
>>> alignments.targets  # doctest: +ELLIPSIS
[SeqRecord(seq=Seq(None, length=158545518), id='hg16.chr7', ...)]
```

By iterating over the `alignments` we obtain one `Alignment` object for each alignment block in the bigMaf file:

```
>>> alignment = next(alignments)
>>> alignment.score
23262.0
>>> {seq.id: len(seq) for seq in alignment.sequences}  # doctest: +NORMALIZE_WHITESPACE
{'hg16.chr7': 158545518,
 'panTro1.chr6': 161576975,
 'baboon': 4622798,
 'mm4.chr6': 151104725,
 'rn3.chr4': 187371129}
>>> alignment.coordinates  # doctest: +NORMALIZE_WHITESPACE
array([[27578828, 27578829, 27578831, 27578831, 27578850, 27578850, 27578866],
       [28741140, 28741141, 28741143, 28741143, 28741162, 28741162, 28741178],
       [  116834,   116835,   116837,   116837,   116856,   116856, 116872],
       [53215344, 53215344, 53215346, 53215347, 53215366, 53215366, 53215382],
       [81344243, 81344243, 81344245, 81344245, 81344264, 81344267, 81344283]])
>>> print(alignment)
hg16.chr7  27578828 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG 27578866
panTro1.c  28741140 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG 28741178
baboon       116834 AAA-GGGAATGTTAACCAAATGA---GTTGTCTCTTATGGTG   116872
mm4.chr6   53215344 -AATGGGAATGTTAAGCAAACGA---ATTGTCTCTCAGTGTG 53215382
rn3.chr4   81344243 -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCAATGTG 81344283
<BLANKLINE>
>>> print(format(alignment, "phylip"))
5 42
hg16.chr7 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
panTro1.chAAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
baboon    AAA-GGGAATGTTAACCAAATGA---GTTGTCTCTTATGGTG
mm4.chr6  -AATGGGAATGTTAAGCAAACGA---ATTGTCTCTCAGTGTG
rn3.chr4  -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCAATGTG
<BLANKLINE>
```

Information in the "`i`", "`e`", and "`q`" lines is stored in the same way as in the corresponding MAF file (see section 6.7.18):

```
>>> from Bio import Align
>>> alignments = Align.parse("ucsc_mm9_chr10.bb", "bigmaf")
>>> for i in range(10):
...     alignment = next(alignments)
```

```
...
>>> alignment.score
19159.0
>>> print(alignment)
mm9.chr10   3014644 CCTGTACC---CTTTGGTGAGAATTTTTGTTTCAGTGTTAAAAGTTTG   3014689
hg18.chr6 155029206 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAAC-TTTT 155029160
panTro2.c 157519257 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAAC-TTTT 157519211
calJac1.C      6182 CCTATACCTTTCTTTCATGAGAA-TTTTGTTTGAATCCTAAAC-TTTT      6228
loxAfr1.s      9407 ------------TTTGGTTAGAA-TTATGCTTTAATTCAAAAC-TTCC      9373
<BLANKLINE>
>>> print(format(alignment, "MAF"))
a score=19159.000000
s mm9.chr10                         3014644   45 + 129993255 CCTGTACC---CTTTGGTGAGAATTTTTGTTTCAGTGTTAAA
s hg18.chr6                        15870786   46 - 170899992 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAA
i hg18.chr6                        I 9085 C 0
s panTro2.chr6                     16389355   46 - 173908612 CCTATACCTTTCTTTTATGAGAA-TTTTGTTTTAATCCTAAA
q panTro2.chr6                                                9999999999999999999999-99999999999999999
i panTro2.chr6                     I 9106 C 0
s calJac1.Contig6394                   6182   46 +    133105 CCTATACCTTTCTTTCATGAGAA-TTTTGTTTGAATCCTAAA
i calJac1.Contig6394               N 0 C 0
s loxAfr1.scaffold_75566               1167   34 -     10574 ------------TTTGGTTAGAA-TTATGCTTTAATTCAAAA
q loxAfr1.scaffold_75566                                      ------------99999699899-99999999999986999
i loxAfr1.scaffold_75566           N 0 C 0
e tupBel1.scaffold_114895.1-498454   167376 4145 -    498454 I
e echTel1.scaffold_288249             87661 7564 +    100002 I
e otoGar1.scaffold_334.1-359464      181217 2931 -    359464 I
e ponAbe2.chr6                     16161448 8044 - 174210431 I
<BLANKLINE>
<BLANKLINE>
>>> alignment.sequences[1].annotations
{'leftStatus': 'I', 'leftCount': 9085, 'rightStatus': 'C', 'rightCount': 0}
>>> alignment.sequences[2].annotations["quality"]
'99999999999999999999999999999999999999999999999'
>>> alignment.sequences[4].annotations["quality"]
'99999698999999999999998699989997'
>>> alignment.annotations["empty"]  # doctest: +NORMALIZE_WHITESPACE
[(SeqRecord(seq=Seq(None, length=498454), id='tupBel1.scaffold_114895.1-498454', name='', description='
 (SeqRecord(seq=Seq(None, length=100002), id='echTel1.scaffold_288249', name='', description='', dbxref
 (SeqRecord(seq=Seq(None, length=359464), id='otoGar1.scaffold_334.1-359464', name='', description='', 
 (SeqRecord(seq=Seq(None, length=174210431), id='ponAbe2.chr6', name='', description='', dbxrefs=[]), (
```

To write a complete bigMaf file, use `Bio.Align.write(alignments, "myfilename.bb", fmt="bigMaf")`, where `myfilename.bb` is the name of the output bigMaf file. Alternatively, you can use a (binary) stream for output. If you are creating the alignments from scratch, you can use the `Alignments` (plural) class to create a list-like `alignments` object (see Section 6.5) and give it a `targets` attribute. The latter must be a list of `SeqRecord` objects for the chromosomes for the reference species in the order in which they appear in the alignments. Alternatively, you can use the `targets` keyword argument when calling `Bio.Align.write`. The `id` of each `SeqRecord` must be of the form `reference.chromosome`, where `reference` refers to the reference species. `Bio.Align.write` has the additional keyword argument `compress` (`True` by default) specifying whether the data should be compressed using zlib.

As a bigMaf file is a special case of a bigBed file, you can use the `search` method on the `alignments` object to find alignments to specific regions of the reference species. For example, we can look for regions

on chr10 between positions 3018000 and 3019000 on chromosome 10:

```
>>> selected_alignments = alignments.search("mm9.chr10", 3018000, 3019000)
>>> for alignment in selected_alignments:
...     start, end = alignment.coordinates[0, 0], alignment.coordinates[0, -1]
...     print(start, end)
...
3017743 3018161
3018161 3018230
3018230 3018359
3018359 3018482
3018482 3018644
3018644 3018822
3018822 3018932
3018932 3019271
```

The chromosome name may be `None` to include all chromosomes, and the start and end positions may be `None` to start searching from position 0 or to continue searching until the end of the chromosome, respectively. Note that we can search on genomic position for the reference species only.

6.7.20 UCSC chain file format

Chain files describe a pairwise alignment between two nucleotide sequences, allowing gaps in both sequences. Only the length of each aligned subsequences and the gap lengths are stored in a chain file; the sequences themselves are not stored. Chain files are typically used to store alignments between two genome assembly versions, allowing alignments to one genome assembly version to be lifted over to the other genome assembly. This is an example of a chain file (available as `psl_34_001.chain` in the `Tests/Blat` subdirectory of the Biopython distribution):

```
chain 16 chr4 191154276 + 61646095 61646111 hg18_dna 33 + 11 27 1
16
chain 33 chr1 249250621 + 10271783 10271816 hg18_dna 33 + 0 33 2
33
chain 17 chr2 243199373 + 53575980 53575997 hg18_dna 33 - 8 25 3
17
chain 35 chr9 141213431 + 85737865 85737906 hg19_dna 50 + 9 50 4
41
chain 41 chr8 146364022 + 95160479 95160520 hg19_dna 50 + 8 49 5
41
chain 30 chr22 51304566 + 42144400 42144436 hg19_dna 50 + 11 47 6
36
chain 41 chr2 243199373 + 183925984 183926028 hg19_dna 50 + 1 49 7
6       0       4
38
chain 31 chr19 59128983 + 35483340 35483510 hg19_dna 50 + 10 46 8
25      134     0
11
chain 39 chr18 78077248 + 23891310 23891349 hg19_dna 50 + 10 49 9
39
...
```

This file was generated by running UCSC's `pslToChain` program on the PSL file `psl_34_001.psl`. According to the chain file format specification, there should be a blank line after each chain block, but some tools (including `pslToChain`) apparently do not follow this rule.

To parse this file, use

```
>>> from Bio import Align
>>> alignments = Align.parse("psl_34_001.chain", "chain")
```

Iterate over alignments to get one `Alignment` object for each chain:

```
>>> for alignment in alignments:
...     print(alignment.target.id, alignment.query.id)  # doctest: +ELLIPSIS
...
chr4 hg18_dna
chr1 hg18_dna
chr2 hg18_dna
chr9 hg19_dna
chr8 hg19_dna
chr22 hg19_dna
chr2 hg19_dna
...
chr1 hg19_dna
```

Rewind the alignments, and iterate from the start until we reach the seventh alignment:

```
>>> alignments.rewind()
>>> for i in range(7):
...     alignment = next(alignments)
...
```

Check the alignment score and chain ID (the first and last number, respectively, in the header line of each chain block) to confirm that we got the seventh alignment:

```
>>> alignment.score
41.0
>>> alignment.annotations["id"]
'7'
```

We can print the alignment in the chain file format. The alignment coordinates are consistent with the information in the chain block, with an aligned section of 6 nucleotides, a gap of 4 nucleotides, and an aligned section of 38 nucleotides:

```
>>> print(format(alignment, "chain"))  # doctest: +NORMALIZE_WHITESPACE
chain 41 chr2 243199373 + 183925984 183926028 hg19_dna 50 + 1 49 7
6       0       4
38
<BLANKLINE>
<BLANKLINE>
>>> alignment.coordinates
array([[183925984, 183925990, 183925990, 183926028],
       [       1,         7,        11,        49]])
>>> print(alignment)
chr2      183925984 ??????----??????????????????????????????????????? 183926028
                  0 ||||||----|||||||||||||||||||||||||||||||||||||||        48
hg19_dna          1 ???????????????????????????????????????????????        49
<BLANKLINE>
```

We can also print the alignment in a few other alignment fite formats:

```
>>> print(format(alignment, "BED"))  # doctest: +NORMALIZE_WHITESPACE
chr2        183925984        183926028        hg19_dna        41        +        183925984        18392
<BLANKLINE>
>>> print(format(alignment, "PSL"))  # doctest: +NORMALIZE_WHITESPACE
44        0        0        0        1        4        0        0        +        hg19_dna        50
<BLANKLINE>
>>> print(format(alignment, "exonerate"))
vulgar: hg19_dna 1 49 + chr2 183925984 183926028 + 41 M 6 6 G 4 0 M 38 38
<BLANKLINE>
>>> print(alignment.format("exonerate", "cigar"))
cigar: hg19_dna 1 49 + chr2 183925984 183926028 + 41 M 6 I 4 M 38
<BLANKLINE>
>>> print(format(alignment, "sam"))  # doctest: +NORMALIZE_WHITESPACE
hg19_dna        0        chr2        183925985        255        1S6M4I38M1S        *        0        0
<BLANKLINE>
```

Chapter 7

Pairwise sequence alignment

Pairwise sequence alignment is the process of aligning two sequences to each other by optimizing the similarity score between them. The `Bio.Align` module contains the `PairwiseAligner` class for global and local alignments using the Needleman-Wunsch, Smith-Waterman, Gotoh (three-state), and Waterman-Smith-Beyer global and local pairwise alignment algorithms, with numerous options to change the alignment parameters. We refer to Durbin *et al.* [11] for in-depth information on sequence alignment algorithms.

7.1 Basic usage

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

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

The `PairwiseAligner` object `aligner` (see Section 7.2) stores the alignment parameters to be used for the pairwise alignments. These attributes can be set in the constructor of the object:

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

or after the object is made:

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

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

```
>>> target = "GAACT"
>>> query = "GAT"
>>> score = aligner.score(target, query)
>>> score
3.0
```

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

```
>>> alignments = aligner.align(target, query)
>>> alignment = alignments[0]
>>> alignment  # doctest: +ELLIPSIS
<Alignment object (2 rows x 5 columns) at ...>
```

Iterate over the `Alignment` objects and print them to see the alignments:

```
>>> for alignment in alignments:
...     print(alignment)
...
target            0 GAACT 5
                  0 ||--| 5
query             0 GA--T 3
<BLANKLINE>
target            0 GAACT 5
                  0 |-|-| 5
query             0 G-A-T 3
<BLANKLINE>
```

Each alignment stores the alignment score:

```
>>> alignment.score
3.0
```

as well as pointers to the sequences that were aligned:

```
>>> alignment.target
'GAACT'
>>> alignment.query
'GAT'
```

Internally, the alignment is stored in terms of the sequence coordinates:

```
>>> alignment = alignments[0]
>>> alignment.coordinates
array([[0, 2, 4, 5],
       [0, 2, 2, 3]])
```

Here, the two rows refer to the target and query sequence. These coordinates show that the alignment consists of the following three blocks:

- `target[0:2]` aligned to `query[0:2]`;
- `target[2:4]` aligned to a gap, since `query[2:2]` is an empty string;
- `target[4:5]` aligned to `query[2:3]`.

The number of aligned sequences is always 2 for a pairwise alignment:

```
>>> len(alignment)
2
```

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

```
>>> alignment.length
5
```

The `aligned` property, which returns the start and end indices of aligned subsequences, returns two tuples of length 2 for the first alignment:

```
>>> alignment.aligned
array([[[0, 2],
        [4, 5]],
<BLANKLINE>
       [[0, 2],
        [2, 3]]])
```

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

```
>>> alignment = alignments[1]
>>> print(alignment)
target            0 GAACT 5
                  0 |-|-| 5
query             0 G-A-T 3
<BLANKLINE>
>>> alignment.aligned
array([[[0, 1],
        [2, 3],
        [4, 5]],
<BLANKLINE>
       [[0, 1],
        [1, 2],
        [2, 3]]])
```

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

```
>>> aligner.mode = "global"
>>> aligner.mismatch_score = -10
>>> alignments = aligner.align("AAACAAA", "AAAGAAA")
>>> len(alignments)
2
>>> print(alignments[0])
target            0 AAAC-AAA 7
                  0 |||--||| 8
query             0 AAA-GAAA 7
<BLANKLINE>
>>> alignments[0].aligned
array([[[0, 3],
        [4, 7]],
<BLANKLINE>
       [[0, 3],
        [4, 7]]])
>>> print(alignments[1])
target            0 AAA-CAAA 7
                  0 |||--||| 8
query             0 AAAG-AAA 7
<BLANKLINE>
>>> alignments[1].aligned
array([[[0, 3],
        [4, 7]],
<BLANKLINE>
       [[0, 3],
        [4, 7]]])
```

The `map` method can be applied on a pairwise alignment `alignment1` to find the pairwise alignment of the query of `alignment2` to the target of `alignment1`, where the target of `alignment2` and the query of `alignment1` are identical. A typical example is where `alignment1` is the pairwise alignment between a chromosome and a transcript, `alignment2` is the pairwise alignment between the transcript and a sequence (e.g., an RNA-seq read), and we want to find the alignment of the sequence to the chromosome:

```
>>> aligner.mode = "local"
>>> aligner.open_gap_score = -1
>>> aligner.extend_gap_score = 0
>>> chromosome = "AAAAAAAACCCCCCCCAAAAAAAAAAAAGGGGGGGAAAAAAAA"
>>> transcript = "CCCCCCCGGGGGG"
>>> alignments1 = aligner.align(chromosome, transcript)
>>> len(alignments1)
1
>>> alignment1 = alignments1[0]
>>> print(alignment1)
target            8 CCCCCCCAAAAAAAAAAAGGGGGG 32
                  0 |||||||-----------|||||| 24
query             0 CCCCCCC-----------GGGGGG 13
<BLANKLINE>
>>> sequence = "CCCCGGGG"
>>> alignments2 = aligner.align(transcript, sequence)
>>> len(alignments2)
1
>>> alignment2 = alignments2[0]
>>> print(alignment2)
target            3 CCCCGGGG 11
                  0 ||||||||  8
query             0 CCCCGGGG  8
<BLANKLINE>
>>> mapped_alignment = alignment1.map(alignment2)
>>> print(mapped_alignment)
target           11 CCCCAAAAAAAAAAGGGG 30
                  0 ||||-----------|||| 19
query             0 CCCC-----------GGGG  8
<BLANKLINE>
>>> format(mapped_alignment, "psl")
'8\t0\t0\t0\t0\t0\t1\t11\t+\tquery\t8\t0\t8\ttarget\t40\t11\t30\t2\t4,4,\t0,4,\t11,26,\n'
```

Mapping the alignment does not depend on the sequence contents. If we delete the sequence contents, the same alignment is found in PSL format (though we obviously lose the ability to print the sequence alignment):

```
>>> from Bio.Seq import Seq
>>> alignment1.target = Seq(None, len(alignment1.target))
>>> alignment1.query = Seq(None, len(alignment1.query))
>>> alignment2.target = Seq(None, len(alignment2.target))
>>> alignment2.query = Seq(None, len(alignment2.query))
>>> mapped_alignment = alignment1.map(alignment2)
>>> format(mapped_alignment, "psl")
'8\t0\t0\t0\t0\t0\t1\t11\t+\tquery\t8\t0\t8\ttarget\t40\t11\t30\t2\t4,4,\t0,4,\t11,26,\n'
```

By default, a global pairwise alignment is performed, which finds the optimal alignment over the whole

length of `target` and `query`. Instead, a local alignment will find the subsequence of `target` and `query` with the highest alignment score. Local alignments can be generated by setting `aligner.mode` to `"local"`:

```
>>> aligner.mode = "local"
>>> target = "AGAACTC"
>>> query = "GAACT"
>>> score = aligner.score(target, query)
>>> score
5.0
>>> alignments = aligner.align(target, query)
>>> for alignment in alignments:
...     print(alignment)
...
target            1 GAACT 6
                  0 ||||| 5
query             0 GAACT 5
<BLANKLINE>
```

Note that there is some ambiguity in the definition of the best local alignments if segments with a score 0 can be added to the alignment. We follow the suggestion by Waterman & Eggert [50] and disallow such extensions.

7.2 The pairwise aligner object

The `PairwiseAligner` object stores all alignment parameters to be used for the pairwise alignments. To see an overview of the values for all parameters, use

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner(match_score=1.0, mode="local")
>>> print(aligner)
Pairwise sequence aligner with parameters
  wildcard: None
  match_score: 1.000000
  mismatch_score: 0.000000
  target_internal_open_gap_score: 0.000000
  target_internal_extend_gap_score: 0.000000
  target_left_open_gap_score: 0.000000
  target_left_extend_gap_score: 0.000000
  target_right_open_gap_score: 0.000000
  target_right_extend_gap_score: 0.000000
  query_internal_open_gap_score: 0.000000
  query_internal_extend_gap_score: 0.000000
  query_left_open_gap_score: 0.000000
  query_left_extend_gap_score: 0.000000
  query_right_open_gap_score: 0.000000
  query_right_extend_gap_score: 0.000000
  mode: local
<BLANKLINE>
```

See Sections 7.3, 7.4, and 7.5 below for the definition of these parameters. The attribute `mode` (described above in Section 7.1) can be set equal to `"global"` or `"local"` to specify global or local pairwise alignment, respectively.

Depending on the gap scoring parameters (see Sections 7.4 and 7.5) and mode, a `PairwiseAligner` object automatically chooses the appropriate algorithm to use for pairwise sequence alignment. To verify the selected algorithm, use

```
>>> aligner.algorithm
'Smith-Waterman'
```

This attribute is read-only.

A `PairwiseAligner` object also stores the precision ϵ to be used during alignment. The value of ϵ is stored in the attribute `aligner.epsilon`, and by default is equal to 10^{-6}:

```
>>> aligner.epsilon
1e-06
```

Two scores will be considered equal to each other for the purpose of the alignment if the absolute difference between them is less than ϵ.

7.3 Substitution scores

Substitution scores define the value to be added to the total score when two letters (nucleotides or amino acids) are aligned to each other. The substitution scores to be used by the `PairwiseAligner` can be specified in two ways:

- By specifying a match score for identical letters, and a mismatch scores for mismatched letters. Nucleotide sequence alignments are typically based on match and mismatch scores. For example, by default BLAST [1] uses a match score of $+1$ and a mismatch score of -2 for nucleotide alignments by `megablast`, with a gap penalty of 2.5 (see section 7.4 for more information on gap scores). Match and mismatch scores can be specified by setting the `match` and `mismatch` attributes of the `PairwiseAligner` object:

  ```
  >>> from Bio import Align
  >>> aligner = Align.PairwiseAligner()
  >>> aligner.match_score
  1.0
  >>> aligner.mismatch_score
  0.0
  >>> score = aligner.score("ACGT", "ACAT")
  >>> print(score)
  3.0
  >>> aligner.match_score = 1.0
  >>> aligner.mismatch_score = -2.0
  >>> aligner.gap_score = -2.5
  >>> score = aligner.score("ACGT", "ACAT")
  >>> print(score)
  1.0
  ```

  When using match and mismatch scores, you can specify a wildcard character (`None` by default) for unknown letters. These will get a zero score in alignments, irrespective of the value of the match or mismatch score:

  ```
  >>> aligner.wildcard = "?"
  >>> score = aligner.score("ACGT", "AC?T")
  >>> print(score)
  3.0
  ```

- Alternatively, you can use the `substitution_matrix` attribute of the `PairwiseAligner` object to specify a substitution matrix. This allows you to apply different scores for different pairs of matched and mismatched letters. This is typically used for amino acid sequence alignments. For example, by default BLAST [1] uses the BLOSUM62 substitution matrix for protein alignments by `blastp`. This substitution matrix is available from Biopython:

```
>>> from Bio.Align import substitution_matrices
>>> substitution_matrices.load()  # doctest: +ELLIPSIS
['BENNER22', 'BENNER6', 'BENNER74', 'BLASTN', 'BLASTP', 'BLOSUM45', 'BLOSUM50', 'BLOSUM62', ..., 'T
>>> matrix = substitution_matrices.load("BLOSUM62")
>>> print(matrix)  # doctest: +ELLIPSIS
#  Matrix made by matblas from blosum62.iij
...
     A    R    N    D    C    Q ...
A  4.0 -1.0 -2.0 -2.0  0.0 -1.0 ...
R -1.0  5.0  0.0 -2.0 -3.0  1.0 ...
N -2.0  0.0  6.0  1.0 -3.0  0.0 ...
D -2.0 -2.0  1.0  6.0 -3.0  0.0 ...
C  0.0 -3.0 -3.0 -3.0  9.0 -3.0 ...
Q -1.0  1.0  0.0  0.0 -3.0  5.0 ...
...
>>> aligner.substitution_matrix = matrix
>>> score = aligner.score("ACDQ", "ACDQ")
>>> score
24.0
>>> score = aligner.score("ACDQ", "ACNQ")
>>> score
19.0
```

  When using a substitution matrix, `X` is *not* interpreted as an unknown character. Instead, the score provided by the substitution matrix will be used:

```
>>> matrix["D", "X"]
-1.0
>>> score = aligner.score("ACDQ", "ACXQ")
>>> score
17.0
```

By default, `aligner.substitution_matrix` is `None`. The attributes `aligner.match_score` and `aligner.mismatch_score` are ignored if `aligner.substitution_matrix` is not `None`. Setting `aligner.match_score` or `aligner.mismatch_score` to valid values will reset `aligner.substitution_matrix` to `None`.

7.4 Affine gap scores

Affine gap scores are defined by a score to open a gap, and a score to extend an existing gap:

gap score = open gap score + $(n-1) \times$ extend gap score,

where n is the length of the gap. Biopython's pairwise sequence aligner allows fine-grained control over the gap scoring scheme by specifying the following twelve attributes of a `PairwiseAligner` object:

These attributes allow for different gap scores for internal gaps and on either end of the sequence, as shown in this example:

For convenience, `PairwiseAligner` objects have additional attributes that refer to a number of these values collectively, as shown (hierarchically) in Table 7.1.

Opening scores	Extending scores
query_left_open_gap_score	query_left_extend_gap_score
query_internal_open_gap_score	query_internal_extend_gap_score
query_right_open_gap_score	query_right_extend_gap_score
target_left_open_gap_score	target_left_extend_gap_score
target_internal_open_gap_score	target_internal_extend_gap_score
target_right_open_gap_score	target_right_extend_gap_score

target	query	score
A	-	query left open gap score
C	-	query left extend gap score
C	-	query left extend gap score
G	G	match score
G	T	mismatch score
G	-	query internal open gap score
A	-	query internal extend gap score
A	-	query internal extend gap score
T	T	match score
A	A	match score
G	-	query internal open gap score
C	C	match score
-	C	target internal open gap score
-	C	target internal extend gap score
C	C	match score
T	G	mismatch score
C	C	match score
-	C	target internal open gap score
A	A	match score
-	T	target right open gap score
-	A	target right extend gap score
-	A	target right extend gap score

7.5 General gap scores

For even more fine-grained control over the gap scores, you can specify a gap scoring function. For example, the gap scoring function below disallows a gap after two nucleotides in the query sequence:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> def my_gap_score_function(start, length):
...     if start == 2:
...         return -1000
...     else:
...         return -1 * length
...
>>> aligner.query_gap_score = my_gap_score_function
>>> alignments = aligner.align("AACTT", "AATT")
>>> for alignment in alignments:
...     print(alignment)
...
target            0 AACTT 5
```

Table 7.1: Meta-attributes of PairwiseAligner objects.

Meta-attribute	**Attributes it maps to**
gap_score	target_gap_score, query_gap_score
open_gap_score	target_open_gap_score, query_open_gap_score
extend_gap_score	target_extend_gap_score, query_extend_gap_score
internal_gap_score	target_internal_gap_score, query_internal_gap_score
internal_open_gap_score	target_internal_open_gap_score, query_internal_open_gap_score
internal_extend_gap_score	target_internal_extend_gap_score, query_internal_extend_gap_score
end_gap_score	target_end_gap_score, query_end_gap_score
end_open_gap_score	target_end_open_gap_score, query_end_open_gap_score
end_extend_gap_score	target_end_extend_gap_score, query_end_extend_gap_score
left_gap_score	target_left_gap_score, query_left_gap_score
right_gap_score	target_right_gap_score, query_right_gap_score
left_open_gap_score	target_left_open_gap_score, query_left_open_gap_score
left_extend_gap_score	target_left_extend_gap_score, query_left_extend_gap_score
right_open_gap_score	target_right_open_gap_score, query_right_open_gap_score
right_extend_gap_score	target_right_extend_gap_score, query_right_extend_gap_score
target_open_gap_score	target_internal_open_gap_score, target_left_open_gap_score, target_right_open_gap_score
target_extend_gap_score	target_internal_extend_gap_score, target_left_extend_gap_score, target_right_extend_gap_score
target_gap_score	target_open_gap_score, target_extend_gap_score
query_open_gap_score	query_internal_open_gap_score, query_left_open_gap_score, query_right_open_gap_score
query_extend_gap_score	query_internal_extend_gap_score, query_left_extend_gap_score, query_right_extend_gap_score
query_gap_score	query_open_gap_score, query_extend_gap_score
target_internal_gap_score	target_internal_open_gap_score, target_internal_extend_gap_score
target_end_gap_score	target_end_open_gap_score, target_end_extend_gap_score
target_end_open_gap_score	target_left_open_gap_score, target_right_open_gap_score
target_end_extend_gap_score	target_left_extend_gap_score, target_right_extend_gap_score
target_left_gap_score	target_left_open_gap_score, target_left_extend_gap_score
target_right_gap_score	target_right_open_gap_score, target_right_extend_gap_score
query_end_gap_score	query_end_open_gap_score, query_end_extend_gap_score
query_end_open_gap_score	query_left_open_gap_score, query_right_open_gap_score
query_end_extend_gap_score	query_left_extend_gap_score, query_right_extend_gap_score
query_internal_gap_score	query_internal_open_gap_score, query_internal_extend_gap_score
query_left_gap_score	query_left_open_gap_score, query_left_extend_gap_score
query_right_gap_score	query_right_open_gap_score, query_right_extend_gap_score

```
                0 -|.|| 5
query           0 -AATT 4
<BLANKLINE>
target          0 AACTT 5
                0 |-.|| 5
query           0 A-ATT 4
<BLANKLINE>
target          0 AACTT 5
                0 ||.-| 5
query           0 AAT-T 4
<BLANKLINE>
target          0 AACTT 5
                0 ||.|- 5
query           0 AATT- 4
<BLANKLINE>
```

7.6 Using a pre-defined substitution matrix and gap scores

By default, a `PairwiseAligner` object is initialized with a match score of +1.0, a mismatch score of 0.0, and all gap scores equal to 0.0, While this has the benefit of being a simple scoring scheme, in general it does not give the best performance. Instead, you can use the argument `scoring` to select a predefined scoring scheme when initializing a `PairwiseAligner` object. Currently, the provided scoring schemes are `blastn` and `megablast`, which are suitable for nucleotide alignments, and `blastp`, which is suitable for protein alignments. Selecting these scoring schemes will initialize the `PairwiseAligner` object to the default scoring parameters used by BLASTN, MegaBLAST, and BLASTP, respectively.

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner(scoring="blastn")
>>> print(aligner)  # doctest:+ELLIPSIS
Pairwise sequence aligner with parameters
  substitution_matrix: <Array object at ...>
  target_internal_open_gap_score: -7.000000
  target_internal_extend_gap_score: -2.000000
  target_left_open_gap_score: -7.000000
  target_left_extend_gap_score: -2.000000
  target_right_open_gap_score: -7.000000
  target_right_extend_gap_score: -2.000000
  query_internal_open_gap_score: -7.000000
  query_internal_extend_gap_score: -2.000000
  query_left_open_gap_score: -7.000000
  query_left_extend_gap_score: -2.000000
  query_right_open_gap_score: -7.000000
  query_right_extend_gap_score: -2.000000
  mode: global
<BLANKLINE>
>>> print(aligner.substitution_matrix[:, :])
     A    T    G    C    S    W    R    Y    K    M    B    V    H    D    N
A  2.0 -3.0 -3.0 -3.0 -3.0 -1.0 -1.0 -3.0 -3.0 -1.0 -3.0 -1.0 -1.0 -1.0 -2.0
T -3.0  2.0 -3.0 -3.0 -3.0 -1.0 -3.0 -1.0 -1.0 -3.0 -1.0 -3.0 -1.0 -1.0 -2.0
G -3.0 -3.0  2.0 -3.0 -1.0 -3.0 -1.0 -3.0 -1.0 -3.0 -1.0 -1.0 -3.0 -1.0 -2.0
C -3.0 -3.0 -3.0  2.0 -1.0 -3.0 -3.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -3.0 -2.0
```

```
S -3.0 -3.0 -1.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
W -1.0 -1.0 -3.0 -3.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
R -1.0 -3.0 -1.0 -3.0 -1.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
Y -3.0 -1.0 -3.0 -1.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
K -3.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -2.0
M -1.0 -3.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
B -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
V -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
H -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
D -1.0 -1.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -2.0
N -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0
<BLANKLINE>
```

7.7 Iterating over alignments

The `alignments` returned by `aligner.align` are a kind of immutable iterable objects (similar to `range`). While they appear similar to a `tuple` or `list` of `Alignment` objects, they are different in the sense that each `Alignment` object is created dynamically when it is needed. This approach was chosen because the number of alignments can be extremely large, in particular for poor alignments (see Section 7.10 for an example).

You can perform the following operations on `alignments`:

- `len(alignments)` returns the number of alignments stored. This function returns quickly, even if the number of alignments is huge. If the number of alignments is extremely large (typically, larger than 9,223,372,036,854,775,807, which is the largest integer that can be stored as a `long int` on 64 bit machines), `len(alignments)` will raise an `OverflowError`. A large number of alignments suggests that the alignment quality is low.

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> alignments = aligner.align("AAA", "AA")
>>> len(alignments)
3
```

- You can extract a specific alignment by index:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> alignments = aligner.align("AAA", "AA")
>>> print(alignments[2])
target            0 AAA 3
                  0 -|| 3
query             0 -AA 2
<BLANKLINE>
>>> print(alignments[0])
target            0 AAA 3
                  0 ||- 3
query             0 AA- 2
<BLANKLINE>
```

- You can iterate over alignments, for example as in

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

By calling `alignments.rewind`, you can rewind the `alignments` iterator to the first alignment and iterate over the alignments from the beginning:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> alignments = aligner.align("AAA", "AA")
>>> for alignment in alignments:
...     print(alignment)
...
target            0 AAA 3
                  0 ||- 3
query             0 AA- 2
<BLANKLINE>
target            0 AAA 3
                  0 |-| 3
query             0 A-A 2
<BLANKLINE>
target            0 AAA 3
                  0 -|| 3
query             0 -AA 2
<BLANKLINE>
>>> alignments.rewind()
>>> for alignment in alignments:
...     print(alignment)
...
target            0 AAA 3
                  0 ||- 3
query             0 AA- 2
<BLANKLINE>
target            0 AAA 3
                  0 |-| 3
query             0 A-A 2
<BLANKLINE>
target            0 AAA 3
                  0 -|| 3
query             0 -AA 2
<BLANKLINE>
```

You can also convert the `alignments` iterator into a `list` or `tuple`:

```
>>> alignments = list(alignments)
```

It is wise to check the number of alignments by calling `len(alignments)` before attempting to call `list(alignments)` to save all alignments as a list.

- The alignment score (which has the same value for each alignment in `alignments`) is stored as an attribute. This allows you to check the alignment score before proceeding to extract individual alignments:

```
>>> print(alignments.score)
2.0
```

7.8 Aligning to the reverse strand

By default, the pairwise aligner aligns the forward strand of the query to the forward strand of the target. To calculate the alignment score for `query` to the reverse strand of `target`, use `strand="-"`:

```
>>> from Bio import Align
>>> from Bio.Seq import reverse_complement
>>> target = "AAAACCC"
>>> query = "AACC"
>>> aligner = Align.PairwiseAligner()
>>> aligner.mismatch_score = -1
>>> aligner.internal_gap_score = -1
>>> aligner.score(target, query)  # strand is "+" by default
4.0
>>> aligner.score(target, reverse_complement(query), strand="-")
4.0
>>> aligner.score(target, query, strand="-")
0.0
>>> aligner.score(target, reverse_complement(query))
0.0
```

The alignments against the reverse strand can be obtained by specifying `strand="-"` when calling `aligner.align`:

```
>>> alignments = aligner.align(target, query)
>>> len(alignments)
1
>>> print(alignments[0])
target            0 AAAACCC 7
                  0 --||||- 7
query             0 --AACC- 4
<BLANKLINE>
>>> print(alignments[0].format("bed"))  # doctest: +NORMALIZE_WHITESPACE
target   2   6   query   4   +   2   6   0   1   4,   0,
<BLANKLINE>
>>> alignments = aligner.align(target, reverse_complement(query), strand="-")
>>> len(alignments)
1
>>> print(alignments[0])
target            0 AAAACCC 7
                  0 --||||- 7
query             4 --AACC- 0
<BLANKLINE>
>>> print(alignments[0].format("bed"))  # doctest: +NORMALIZE_WHITESPACE
target   2   6   query   4   -   2   6   0   1   4,   0,
<BLANKLINE>
>>> alignments = aligner.align(target, query, strand="-")
>>> len(alignments)
2
>>> print(alignments[0])
target            0 AAAACCC----  7
                  0 ----------- 11
```

```
query             4 -------GGTT  0
<BLANKLINE>
>>> print(alignments[1])
target            0 ----AAAACCC  7
                  0 ---------- 11
query             4 GGTT-------  0
<BLANKLINE>
```

Note that the score for aligning `query` to the reverse strand of `target` may be different from the score for aligning the reverse complement of `query` to the forward strand of `target` if the left and right gap scores are different:

```
>>> aligner.left_gap_score = -0.5
>>> aligner.right_gap_score = -0.2
>>> aligner.score(target, query)
2.8
>>> alignments = aligner.align(target, query)
>>> len(alignments)
1
>>> print(alignments[0])
target            0 AAAACCC 7
                  0 --||||- 7
query             0 --AACC- 4
<BLANKLINE>
>>> aligner.score(target, reverse_complement(query), strand="-")
3.1
>>> alignments = aligner.align(target, reverse_complement(query), strand="-")
>>> len(alignments)
1
>>> print(alignments[0])
target            0 AAAACCC 7
                  0 --||||- 7
query             4 --AACC- 0
<BLANKLINE>
```

7.9 Substitution matrices

Substitution matrices [11] provide the scoring terms for classifying how likely two different residues are to substitute for each other. This is essential in doing sequence comparisons. Biopython provides a ton of common substitution matrices, including the famous PAM and BLOSUM series of matrices, and also provides functionality for creating your own substitution matrices.

7.9.1 Array objects

You can think of substitutions matrices as 2D arrays in which the indices are letters (nucleotides or amino acids) rather than integers. The `Array` class in `Bio.Align.substitution_matrices` is a subclass of numpy arrays that supports indexing both by integers and by specific strings. An `Array` instance can either be a one-dimensional array or a square two-dimensional arrays. A one-dimensional `Array` object can for example be used to store the nucleotide frequency of a DNA sequence, while a two-dimensional `Array` object can be used to represent a scoring matrix for sequence alignments.

To create a one-dimensional `Array`, only the alphabet of allowed letters needs to be specified:

```
>>> from Bio.Align.substitution_matrices import Array
>>> counts = Array("ACGT")
>>> print(counts)
A 0.0
C 0.0
G 0.0
T 0.0
<BLANKLINE>
```

The allowed letters are stored in the `alphabet` property:

```
>>> counts.alphabet
'ACGT'
```

This property is read-only; modifying the underlying `_alphabet` attribute may lead to unexpected results. Elements can be accessed both by letter and by integer index:

```
>>> counts["C"] = -3
>>> counts[2] = 7
>>> print(counts)
A  0.0
C -3.0
G  7.0
T  0.0
<BLANKLINE>
>>> counts[1]
-3.0
```

Using a letter that is not in the alphabet, or an index that is out of bounds, will cause a `IndexError`:

```
>>> counts["U"]
Traceback (most recent call last):
    ...
IndexError: 'U'
>>> counts["X"] = 6
Traceback (most recent call last):
    ...
IndexError: 'X'
>>> counts[7]
Traceback (most recent call last):
    ...
IndexError: index 7 is out of bounds for axis 0 with size 4
```

A two-dimensional `Array` can be created by specifying `dims=2`:

```
>>> from Bio.Align.substitution_matrices import Array
>>> counts = Array("ACGT", dims=2)
>>> print(counts)
    A   C   G   T
A 0.0 0.0 0.0 0.0
C 0.0 0.0 0.0 0.0
G 0.0 0.0 0.0 0.0
T 0.0 0.0 0.0 0.0
<BLANKLINE>
```

Again, both letters and integers can be used for indexing, and specifying a letter that is not in the alphabet will cause an `IndexError`:

```
>>> counts["A", "C"] = 12.0
>>> counts[2, 1] = 5.0
>>> counts[3, "T"] = -2
>>> print(counts)
    A    C   G    T
A 0.0 12.0 0.0  0.0
C 0.0  0.0 0.0  0.0
G 0.0  5.0 0.0  0.0
T 0.0  0.0 0.0 -2.0
<BLANKLINE>
>>> counts["X", 1]
Traceback (most recent call last):
    ...
IndexError: 'X'
>>> counts["A", 5]
Traceback (most recent call last):
    ...
IndexError: index 5 is out of bounds for axis 1 with size 4
```

Selecting a row or column from the two-dimensional array will return a one-dimensional `Array`:

```
>>> counts = Array("ACGT", dims=2)
>>> counts["A", "C"] = 12.0
>>> counts[2, 1] = 5.0
>>> counts[3, "T"] = -2

>>> counts["G"]
Array([0., 5., 0., 0.],
      alphabet='ACGT')
>>> counts[:, "C"]
Array([12.,  0.,  5.,  0.],
      alphabet='ACGT')
```

`Array` objects can thus be used as an array and as a dictionary. They can be converted to plain numpy arrays or plain dictionary objects:

```
>>> import numpy as np
>>> x = Array("ACGT")
>>> x["C"] = 5

>>> x
Array([0., 5., 0., 0.],
      alphabet='ACGT')
>>> a = np.array(x)  # create a plain numpy array
>>> a
array([0., 5., 0., 0.])
>>> d = dict(x)  # create a plain dictionary
>>> d
{'A': 0.0, 'C': 5.0, 'G': 0.0, 'T': 0.0}
```

While the alphabet of an `Array` is usually a string, you may also use a tuple of (immutable) objects. This is used for example for a codon substitution matrix, where the keys are not individual nucleotides or amino acids but instead three-nucleotide codons.

While the `alphabet` property of an `Array` is immutable, you can create a new `Array` object by selecting the letters you are interested in from the alphabet. For example,

```
>>> a = Array("ABCD", dims=2, data=np.arange(16).reshape(4, 4))
>>> print(a)
     A    B    C    D
A  0.0  1.0  2.0  3.0
B  4.0  5.0  6.0  7.0
C  8.0  9.0 10.0 11.0
D 12.0 13.0 14.0 15.0
<BLANKLINE>
>>> b = a.select("CAD")
>>> print(b)
     C    A    D
C 10.0  8.0 11.0
A  2.0  0.0  3.0
D 14.0 12.0 15.0
<BLANKLINE>
```

Note that this also allows you to reorder the alphabet.

Data for letters that are not found in the alphabet are set to zero:

```
>>> c = a.select("DEC")
>>> print(c)
     D   E    C
D 15.0 0.0 14.0
E  0.0 0.0  0.0
C 11.0 0.0 10.0
<BLANKLINE>
```

As the `Array` class is a subclass of numpy array, it can be used as such. A `ValueError` is triggered if the `Array` objects appearing in a mathematical operation have different alphabets, for example

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

7.9.2 Calculating a substitution matrix from a pairwise sequence alignment

As `Array` is a subclass of a numpy array, you can apply mathematical operations on an `Array` object in much the same way. Here, we illustrate this by calculating a scoring matrix from the alignment of the 16S ribosomal RNA gene sequences of *Escherichia coli* and *Bacillus subtilis*. First, we create a `PairwiseAligner` object (see Chapter 7) and initialize it with the default scores used by `blastn`:

```
>>> from Bio.Align import PairwiseAligner
>>> aligner = PairwiseAligner(scoring="blastn")
>>> aligner.mode = "local"
```

Next, we read in the 16S ribosomal RNA gene sequence of *Escherichia coli* and *Bacillus subtilis* (provided in `Tests/Align/ecoli.fa` and `Tests/Align/bsubtilis.fa`), and align them to each other:

```
>>> from Bio import SeqIO
>>> sequence1 = SeqIO.read("ecoli.fa", "fasta")
>>> sequence2 = SeqIO.read("bsubtilis.fa", "fasta")
>>> alignments = aligner.align(sequence1, sequence2)
```

The number of alignments generated is very large:

```
>>> len(alignments)
1990656
```

However, as they only differ trivially from each other, we arbitrarily choose the first alignment, and count the number of each substitution:

```
>>> alignment = alignments[0]
>>> substitutions = alignment.substitutions
>>> print(substitutions)
      A     C     G     T
A 307.0  19.0  34.0  19.0
C  15.0 280.0  25.0  29.0
G  34.0  24.0 401.0  20.0
T  24.0  36.0  20.0 228.0
<BLANKLINE>
```

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

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

The background probability is the probability of finding an A, C, G, or T nucleotide in each sequence separately. This can be calculated as the sum of each row or column:

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

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

```
>>> expected_frequencies = background[:, None].dot(background[None, :])
>>> print(format(expected_frequencies, "%.4f"))
       A      C      G      T
A 0.0627 0.0585 0.0793 0.0499
C 0.0585 0.0546 0.0740 0.0466
G 0.0793 0.0740 0.1002 0.0631
T 0.0499 0.0466 0.0631 0.0397
<BLANKLINE>
```

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

```
>>> oddsratios = observed_frequencies / expected_frequencies
>>> import numpy as np
>>> scoring_matrix = np.log2(oddsratios)
>>> print(scoring_matrix)
     A    C    G    T
A  1.7 -2.4 -1.8 -1.8
C -2.4  1.8 -2.2 -1.1
G -1.8 -2.2  1.4 -2.3
T -1.8 -1.1 -2.3  1.9
<BLANKLINE>
```

The matrix can be used to set the substitution matrix for the pairwise aligner (see Chapter 7):

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

7.9.3 Calculating a substitution matrix from a multiple sequence alignment

In this example, we'll first read a protein sequence alignment from the Clustalw file protein.aln (also available online here)

```
>>> from Bio import Align
>>> filename = "protein.aln"
>>> alignment = Align.read(filename, "clustal")
```

Section 6.7.2 contains more information on doing this.

The `substitutions` property of the alignment stores the number of times different residues substitute for each other:

```
>>> substitutions = alignment.substitutions
```

To make the example more readable, we'll select only amino acids with polar charged side chains:

```
>>> substitutions = substitutions.select("DEHKR")
>>> print(substitutions)
       D      E      H      K      R
D 2360.0  270.0   15.0    1.0   48.0
E  241.0 3305.0   15.0   45.0    2.0
H    0.0   18.0 1235.0    8.0    0.0
K    0.0    9.0   24.0 3218.0  130.0
R    2.0    2.0   17.0  103.0 2079.0
<BLANKLINE>
```

Rows and columns for other amino acids were removed from the matrix.

Next, we normalize the matrix and make it symmetric.

```
>>> observed_frequencies = substitutions / substitutions.sum()
>>> observed_frequencies = (observed_frequencies + observed_frequencies.transpose()) / 2.0
>>> print(format(observed_frequencies, "%.4f"))
       D      E      H      K      R
D 0.1795 0.0194 0.0006 0.0000 0.0019
E 0.0194 0.2514 0.0013 0.0021 0.0002
H 0.0006 0.0013 0.0939 0.0012 0.0006
K 0.0000 0.0021 0.0012 0.2448 0.0089
R 0.0019 0.0002 0.0006 0.0089 0.1581
<BLANKLINE>
```

Summing over rows or columns gives the relative frequency of occurrence of each residue:

```
>>> background = observed_frequencies.sum(0)
>>> print(format(background, "%.4f"))
D 0.2015
E 0.2743
H 0.0976
K 0.2569
R 0.1697
<BLANKLINE>
>>> background.sum()
1.0
```

The expected frequency of residue pairs is then

```
>>> expected_frequencies = background[:, None].dot(background[None, :])
>>> print(format(expected_frequencies, "%.4f"))
       D      E      H      K      R
D 0.0406 0.0553 0.0197 0.0518 0.0342
E 0.0553 0.0752 0.0268 0.0705 0.0465
H 0.0197 0.0268 0.0095 0.0251 0.0166
K 0.0518 0.0705 0.0251 0.0660 0.0436
R 0.0342 0.0465 0.0166 0.0436 0.0288
<BLANKLINE>
```

Here, `background[:, None]` creates a 2D array consisting of a single column with the values of `expected_frequencies`, and `rxpected_frequencies[None, :]` a 2D array with these values as a single row. Taking their dot product (inner product) creates a matrix of expected frequencies where each entry consists of two `expected_frequencies` values multiplied with each other. For example, `expected_frequencies['D', 'E']` is equal to `residue_frequencies['D']`

We can now calculate the log-odds matrix by dividing the observed frequencies by the expected frequencies and taking the logarithm:

```
>>> import numpy as np
>>> scoring_matrix = np.log2(observed_frequencies / expected_frequencies)
>>> print(scoring_matrix)
      D    E    H     K    R
D   2.1 -1.5 -5.1 -10.4 -4.2
E  -1.5  1.7 -4.4  -5.1 -8.3
H  -5.1 -4.4  3.3  -4.4 -4.7
K -10.4 -5.1 -4.4   1.9 -2.3
R  -4.2 -8.3 -4.7  -2.3  2.5
<BLANKLINE>
```

This matrix can be used as the substitution matrix when performing alignments. For example,

```
>>> from Bio.Align import PairwiseAligner
>>> aligner = PairwiseAligner()
>>> aligner.substitution_matrix = scoring_matrix
>>> aligner.gap_score = -3.0
>>> alignments = aligner.align("DEHEK", "DHHKK")
>>> print(alignments[0])
target            0 DEHEK 5
                  0 |.|.| 5
query             0 DHHKK 5
<BLANKLINE>
>>> print("%.2f" % alignments.score)
-2.18
>>> score = (
...     scoring_matrix["D", "D"]
...     + scoring_matrix["E", "H"]
...     + scoring_matrix["H", "H"]
...     + scoring_matrix["E", "K"]
...     + scoring_matrix["K", "K"]
... )
>>> print("%.2f" % score)
-2.18
```

(see Chapter 7 for details).

7.9.4 Reading `Array` objects from file

`Bio.Align.substitution_matrices` includes a parser to read one- and two-dimensional `Array` objects from file. One-dimensional arrays are represented by a simple two-column format, with the first column containing the key and the second column the corresponding value. For example, the file `hg38.chrom.sizes` (obtained from UCSC), available in the `Tests/Align` subdirectory of the Biopython distribution, contains the size in nucleotides of each chromosome in human genome assembly hg38:

```
chr1    248956422
chr2    242193529
chr3    198295559
chr4    190214555
...
chrUn_KI270385v1    990
chrUn_KI270423v1    981
chrUn_KI270392v1    971
chrUn_KI270394v1    970
```

To parse this file, use

```
>>> from Bio.Align import substitution_matrices
>>> with open("hg38.chrom.sizes") as handle:
...     table = substitution_matrices.read(handle)
...
>>> print(table)  # doctest: +ELLIPSIS
chr1 248956422.0
chr2 242193529.0
chr3 198295559.0
```

```
chr4 190214555.0
...
chrUn_KI270423v1       981.0
chrUn_KI270392v1       971.0
chrUn_KI270394v1       970.0
<BLANKLINE>
```

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

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

For two-dimensional arrays, we follow the file format of substitution matrices provided by NCBI. For example, the BLOSUM62 matrix, which is the default substitution matrix for NCBI's protein-protein BLAST [1] program `blastp`, is stored as follows:

```
#  Matrix made by matblas from blosum62.iij
#  * column uses minimum score
#  BLOSUM Clustered Scoring Matrix in 1/2 Bit Units
#  Blocks Database = /data/blocks_5.0/blocks.dat
#  Cluster Percentage: >= 62
#  Entropy =   0.6979, Expected =  -0.5209
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1  0 -4
R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1  0 -1 -4
N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  3  0 -1 -4
D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3  4  1 -1 -4
C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1 -3 -3 -2 -4
Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2  0  3 -1 -4
E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3 -1 -2 -1 -4
H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3  0  0 -1 -4
...
```

This file is included in the Biopython distribution under `Bio/Align/substitution_matrices/data`. To parse this file, use

```
>>> from Bio.Align import substitution_matrices
>>> with open("BLOSUM62") as handle:
...     matrix = substitution_matrices.read(handle)
...
>>> print(matrix.alphabet)
```

```
ARNDCQEGHILKMFPSTWYVBZX*
>>> print(matrix["A", "D"])
-2.0
```

The header lines starting with `#` are stored in the attribute `header`:

```
>>> matrix.header[0]
'Matrix made by matblas from blosum62.iij'
```

We can now use this matrix as the substitution matrix on an aligner object:

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

To save an Array object, create a string first:

```
>>> text = str(matrix)
>>> print(text)  # doctest: +ELLIPSIS
#  Matrix made by matblas from blosum62.iij
#  * column uses minimum score
#  BLOSUM Clustered Scoring Matrix in 1/2 Bit Units
#  Blocks Database = /data/blocks_5.0/blocks.dat
#  Cluster Percentage: >= 62
#  Entropy =   0.6979, Expected =  -0.5209
   A    R    N    D    C    Q    E    G    H    I    L    K    M    F    P    S ...
A  4.0 -1.0 -2.0 -2.0  0.0 -1.0 -1.0  0.0 -2.0 -1.0 -1.0 -1.0 -1.0 -2.0 -1.0  1.0 ...
R -1.0  5.0  0.0 -2.0 -3.0  1.0  0.0 -2.0  0.0 -3.0 -2.0  2.0 -1.0 -3.0 -2.0 -1.0 ...
N -2.0  0.0  6.0  1.0 -3.0  0.0  0.0  0.0  1.0 -3.0 -3.0  0.0 -2.0 -3.0 -2.0  1.0 ...
D -2.0 -2.0  1.0  6.0 -3.0  0.0  2.0 -1.0 -1.0 -3.0 -4.0 -1.0 -3.0 -3.0 -1.0  0.0 ...
C  0.0 -3.0 -3.0 -3.0  9.0 -3.0 -4.0 -3.0 -3.0 -1.0 -1.0 -3.0 -1.0 -2.0 -3.0 -1.0 ...
...
```

and write the `text` to a file.

7.9.5 Loading predefined substitution matrices

Biopython contains a large set of substitution matrices defined in the literature, including BLOSUM (Blocks Substitution Matrix) [19] and PAM (Point Accepted Mutation) matrices [9]. These matrices are available as flat files in the `Bio/Align/substitution_matrices/data` directory, and can be loaded into Python using the `load` function in the `substitution_matrices` submodule. For example, the BLOSUM62 matrix can be loaded by running

```
>>> from Bio.Align import substitution_matrices
>>> m = substitution_matrices.load("BLOSUM62")
```

This substitution matrix has an alphabet consisting of the 20 amino acids used in the genetic code, the three ambiguous amino acids B (asparagine or aspartic acid), Z (glutamine or glutamic acid), and X (representing any amino acid), and the stop codon represented by an asterisk:

```
>>> m.alphabet
'ARNDCQEGHILKMFPSTWYVBZX*'
```

To get a full list of available substitution matrices, use `load` without an argument:

```
>>> substitution_matrices.load()  # doctest: +ELLIPSIS
['BENNER22', 'BENNER6', 'BENNER74', 'BLASTN', 'BLASTP', 'BLOSUM45', 'BLOSUM50', ..., 'TRANS']
```

Note that the substitution matrix provided by Schneider *et al.* [40] uses an alphabet consisting of three-nucleotide codons:

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

7.10 Examples

Suppose you want to do a global pairwise alignment between the same two hemoglobin sequences from above (`HBA_HUMAN`, `HBB_HUMAN`) stored in `alpha.faa` and `beta.faa`:

```
>>> from Bio import Align
>>> from Bio import SeqIO
>>> seq1 = SeqIO.read("alpha.faa", "fasta")
>>> seq2 = SeqIO.read("beta.faa", "fasta")
>>> aligner = Align.PairwiseAligner()
>>> score = aligner.score(seq1.seq, seq2.seq)
>>> print(score)
72.0
```

showing an alignment score of 72.0. To see the individual alignments, do

```
>>> alignments = aligner.align(seq1.seq, seq2.seq)
```

In this example, the total number of optimal alignments is huge (more than 4×10^{37}), and calling `len(alignments)` will raise an `OverflowError`:

```
>>> len(alignments)
Traceback (most recent call last):
...
OverflowError: number of optimal alignments is larger than 9223372036854775807
```

Let's have a look at the first alignment:

```
>>> alignment = alignments[0]
```

The alignment object stores the alignment score, as well as the alignment itself:

```
>>> print(alignment.score)
72.0
>>> print(alignment)
target            0 MV-LS-PAD--KTN--VK-AA-WGKV-----GAHAGEYGAEALE-RMFLSF----P-TTK
                  0 ||-|--|----|----|--|--||||-----|---||--|--|--|--|------|-|--
query             0 MVHL-TP--EEK--SAV-TA-LWGKVNVDEVG---GE--A--L-GR--L--LVVYPWT--
<BLANKLINE>
target           41 TY--FPHF----DLSHGS---AQVK-G------HGKKV--A--DA-LTNAVAHV-DDMPN
                 60 ----|--|----|||------|-|--|------|||||--|--|--|--|--|--|---|
query            39 --QRF--FESFGDLS---TPDA-V-MGNPKVKAHGKKVLGAFSD-GL--A--H-LD---N
<BLANKLINE>
```

```
target           79 ALS----A-LSD-LHAH--KLR-VDPV-NFK-LLSHC---LLVT--LAAHLPA----EFT
                120 -|-----|-||--||----||--|||--||--||------|-|---||-|-------|||
query            81 -L-KGTFATLS-ELH--CDKL-HVDP-ENF-RLL---GNVL-V-CVLA-H---HFGKEFT
<BLANKLINE>
target          119 PA-VH-ASLDKFLAS---VSTV------LTS--KYR- 142
                180 |--|--|------|----|--|------|----||-- 217
query           124 P-PV-QA------A-YQKV--VAGVANAL--AHKY-H 147
<BLANKLINE>
```

Better alignments are usually obtained by penalizing gaps: higher costs for opening a gap and lower costs for extending an existing gap. For amino acid sequences match scores are usually encoded in matrices like `PAM` or `BLOSUM`. Thus, a more meaningful alignment for our example can be obtained by using the BLOSUM62 matrix, together with a gap open penalty of 10 and a gap extension penalty of 0.5:

```
>>> from Bio import Align
>>> from Bio import SeqIO
>>> from Bio.Align import substitution_matrices
>>> seq1 = SeqIO.read("alpha.faa", "fasta")
>>> seq2 = SeqIO.read("beta.faa", "fasta")
>>> aligner = Align.PairwiseAligner()
>>> aligner.open_gap_score = -10
>>> aligner.extend_gap_score = -0.5
>>> aligner.substitution_matrix = substitution_matrices.load("BLOSUM62")
>>> score = aligner.score(seq1.seq, seq2.seq)
>>> print(score)
292.5
>>> alignments = aligner.align(seq1.seq, seq2.seq)
>>> len(alignments)
2
>>> print(alignments[0].score)
292.5
>>> print(alignments[0])
target            0 MV-LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF-DLS-----HGS
                  0 ||-|.|..|..|.|.||||--...|.|.|||.|.....|.|...|..|-|||-----.|.
query             0 MVHLTPEEKSAVTALWGKV--NVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGN
<BLANKLINE>
target           53 AQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAH
                 60 ..||.|||||..|.....||.|........||.||..||.|||.||.||...|...||.|
query            58 PKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLAHH
<BLANKLINE>
target          113 LPAEFTPAVHASLDKFLASVSTVLTSKYR 142
                120 ...||||.|.|...|..|.|...|..||. 149
query           118 FGKEFTPPVQAAYQKVVAGVANALAHKYH 147
<BLANKLINE>
```

This alignment has the same score that we obtained earlier with EMBOSS needle using the same sequences and the same parameters.

To perform a local alignment, set `aligner.mode` to `'local'`:

```
>>> aligner.mode = "local"
>>> aligner.open_gap_score = -10
>>> aligner.extend_gap_score = -1
>>> alignments = aligner.align("LSPADKTNVKAA", "PEEKSAV")
```

```
>>> print(len(alignments))
1
>>> alignment = alignments[0]
>>> print(alignment)
target            2 PADKTNV 9
                  0 |..|..| 7
query             0 PEEKSAV 7
<BLANKLINE>
>>> print(alignment.score)
16.0
```

7.11 Generalized pairwise alignments

In most cases, `PairwiseAligner` is used to perform alignments of sequences (strings or `Seq` objects) consisting of single-letter nucleotides or amino acids. More generally, `PairwiseAligner` can also be applied to lists or tuples of arbitrary objects. This section will describe some examples of such generalized pairwise alignments.

7.11.1 Generalized pairwise alignments using a substitution matrix and alphabet

Schneider *et al.* [40] created a substitution matrix for aligning three-nucleotide codons (see below in section 7.9 for more information). This substitution matrix is associated with an alphabet consisting of all three-letter codons:

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

We can use this matrix to align codon sequences to each other:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> aligner.substitution_matrix = m
>>> aligner.gap_score = -1.0
>>> s1 = ("AAT", "CTG", "TTT", "TTT")
>>> s2 = ("AAT", "TTA", "TTT")
>>> alignments = aligner.align(s1, s2)
>>> len(alignments)
2
>>> print(alignments[0])
AAT CTG TTT TTT
|||  ...  |||  ---
AAT TTA TTT  ---
<BLANKLINE>
>>> print(alignments[1])
AAT CTG TTT TTT
|||  ...  ---  |||
AAT TTA ---  TTT
<BLANKLINE>
```

Note that aligning `TTT` to `TTA`, as in this example:

```
AAT CTG TTT TTT
|||  ---  ...  |||
AAT  ---  TTA  TTT
```

would get a much lower score:

```
>>> print(m["CTG", "TTA"])
7.6
>>> print(m["TTT", "TTA"])
-0.3
```

presumably because `CTG` and `TTA` both code for leucine, while `TTT` codes for phenylalanine. The three-letter codon substitution matrix also reveals a preference among codons representing the same amino acid. For example, `TTA` has a preference for `CTG` preferred compared to `CTC`, though all three code for leucine:

```
>>> s1 = ("AAT", "CTG", "CTC", "TTT")
>>> s2 = ("AAT", "TTA", "TTT")
>>> alignments = aligner.align(s1, s2)
>>> len(alignments)
1
>>> print(alignments[0])
AAT CTG CTC TTT
|||  ...  ---  |||
AAT  TTA  ---  TTT
<BLANKLINE>
>>> print(m["CTC", "TTA"])
6.5
```

7.11.2 Generalized pairwise alignments using match/mismatch scores and an alphabet

Using the three-letter amino acid symbols, the sequences above translate to

```
>>> s1 = ("Asn", "Leu", "Leu", "Phe")
>>> s2 = ("Asn", "Leu", "Phe")
```

We can align these sequences directly to each other by using a three-letter amino acid alphabet:

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> aligner.alphabet = ['Ala', 'Arg', 'Asn', 'Asp', 'Cys',
...                     'Gln', 'Glu', 'Gly', 'His', 'Ile',
...                     'Leu', 'Lys', 'Met', 'Phe', 'Pro',
...                     'Ser', 'Thr', 'Trp', 'Tyr', 'Val']  # fmt: skip
...
```

We use +6/-1 match and mismatch scores as an approximation of the BLOSUM62 matrix, and align these sequences to each other:

```
>>> aligner.match = +6
>>> aligner.mismatch = -1
>>> alignments = aligner.align(s1, s2)
>>> print(len(alignments))
2
```

```
>>> print(alignments[0])
Asn Leu Leu Phe
|||  |||  ---  |||
Asn Leu  ---  Phe
<BLANKLINE>
>>> print(alignments[1])
Asn Leu Leu Phe
|||  ---  |||  |||
Asn  ---  Leu Phe
<BLANKLINE>
>>> print(alignments.score)
18.0
```

7.11.3 Generalized pairwise alignments using match/mismatch scores and integer sequences

Internally, the first step when performing an alignment is to replace the two sequences by integer arrays consisting of the indices of each letter in each sequence in the alphabet associated with the aligner. This step can be bypassed by passing integer arrays directly:

```
>>> import numpy as np
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> s1 = np.array([2, 10, 10, 13], np.int32)
>>> s2 = np.array([2, 10, 13], np.int32)
>>> aligner.match = +6
>>> aligner.mismatch = -1
>>> alignments = aligner.align(s1, s2)
>>> print(len(alignments))
2
>>> print(alignments[0])
2 10 10 13
|  ||  --  ||
2 10  --  13
<BLANKLINE>
>>> print(alignments[1])
2 10 10 13
|  --  ||  ||
2  --  10 13
<BLANKLINE>
>>> print(alignments.score)
18.0
```

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

Unknown letters can again be included by defining a wildcard character, and using the corresponding Unicode code point number as the index:

```
>>> aligner.wildcard = "?"
>>> ord(aligner.wildcard)
63
>>> s2 = np.array([2, 63, 13], np.int32)
>>> aligner.gap_score = -3
```

```
>>> alignments = aligner.align(s1, s2)
>>> print(len(alignments))
2
>>> print(alignments[0])
2 10 10 13
| .. -- ||
2 63 -- 13
<BLANKLINE>
>>> print(alignments[1])
2 10 10 13
| -- .. ||
2 -- 63 13
<BLANKLINE>
>>> print(alignments.score)
9.0
```

7.11.4 Generalized pairwise alignments using a substitution matrix and integer sequences

Integer sequences can also be aligned using a substitution matrix, in this case a numpy square array without an alphabet associated with it. In this case, all index values must be non-negative, and smaller than the size of the substitution matrix:

```
>>> from Bio import Align
>>> import numpy as np
>>> aligner = Align.PairwiseAligner()
>>> m = np.eye(5)
>>> m[0, 1:] = m[1:, 0] = -2
>>> m[2, 2] = 3
>>> print(m)
[[ 1. -2. -2. -2. -2.]
 [-2.  1.  0.  0.  0.]
 [-2.  0.  3.  0.  0.]
 [-2.  0.  0.  1.  0.]
 [-2.  0.  0.  0.  1.]]
>>> aligner.substitution_matrix = m
>>> aligner.gap_score = -1
>>> s1 = np.array([0, 2, 3, 4], np.int32)
>>> s2 = np.array([0, 3, 2, 1], np.int32)
>>> alignments = aligner.align(s1, s2)
>>> print(len(alignments))
2
>>> print(alignments[0])
0 - 2 3 4
| - | . -
0 3 2 1 -
<BLANKLINE>
>>> print(alignments[1])
0 - 2 3 4
| - | - .
0 3 2 - 1
<BLANKLINE>
```

Table 7.2: Meta-attributes of CodonAligner objects.

Meta-attribute	Attributes it maps to
frameshift_minus_score	frameshift_minus_two_score, frameshift_minus_one_score
frameshift_plus_score	frameshift_plus_two_score, frameshift_plus_one_score
frameshift_two_score	frameshift_minus_two_score, frameshift_plus_two_score
frameshift_one_score	frameshift_minus_one_score, frameshift_plus_one_score
frameshift_score	frameshift_minus_two_score, frameshift_minus_one_score, frameshift_plus_one_score, frameshift_plus_two_score

```
>>> print(alignments.score)
2.0
```

7.12 Codon alignments

The `CodonAligner` class in the `Bio.Align` module implements a specialized aligner for aligning a nucleotide sequence to the amino acid sequence it encodes. Such alignments are non-trivial if frameshifts occur during translation.

7.12.1 Aligning a nucleotide sequence to an amino acid sequence

To align a nucleotide sequence to an amino acid sequence, first create a `CodonAligner` object:

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

The `CodonAligner` object `aligner` stores the alignment parameters to be used for the alignments:

```
>>> print(aligner)
Codon aligner with parameters
  wildcard: 'X'
  match_score: 1.0
  mismatch_score: 0.0
  frameshift_minus_two_score: -3.0
  frameshift_minus_one_score: -3.0
  frameshift_plus_one_score: -3.0
  frameshift_plus_two_score: -3.0
<BLANKLINE>
```

The `wildcard`, `match_score`, and `mismatch_score` parameters are defined in the same was as for the `PairwiseAligner` class described above (see Section 7.2). The values specified by the `frameshift_minus_two_score`, `frameshift_minus_one_score`, `frameshift_plus_one_score`, and `frameshift_plus_two_score` parameters are added to the alignment score whenever a -2, -1, +1, or +2 frame shift, respectively, occurs in the alignment. By default, the frame shift scores are set to -3.0. Similar to the `PairwiseAligner` class (Table 7.1), the `CodonAligner` class defines additional attributes that refer to a number of these values collectively, as shown in Table 7.2. Now let's consider two nucleotide sequences and the amino acid sequences they encode:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> nuc1 = Seq("TCAGGGACTGCGAGAACCAAGCTACTGCTGCTGCTGGCTGCGCTCTGCGCCGCAGGTGGGGCGCTGGAG")
>>> rna1 = SeqRecord(nuc1, id="rna1")
>>> nuc2 = Seq("TCAGGGACTTCGAGAACCAAGCGCTCCTGCTGCTGGCTGCGCTCGGCGCCGCAGGTGGAGCACTGGAG")
```

```
>>> rna2 = SeqRecord(nuc2, id="rna2")
>>> aa1 = Seq("SGTARTKLLLLLAALCAAGGALE")
>>> aa2 = Seq("SGTSRTKRLLLLAALGAAGGALE")
>>> pro1 = SeqRecord(aa1, id="pro1")
>>> pro2 = SeqRecord(aa2, id="pro2")
```

While the two protein sequences both consist of 23 amino acids, the first nucleotide sequence consists of $3 \times 23 = 69$ nucleotides while the second nucleotide sequence tonsists of only 68 nucleotides:

```
>>> len(pro1)
23
>>> len(pro2)
23
>>> len(rna1)
69
>>> len(rna2)
68
```

This is due to a -1 frame shift event during translation of the second nucleotide sequence. Use `CodonAligner.align` to align `rna1` to `pro1`, and `rna2` to `pro2`, returning an iterator of `Alignment` objects:

```
>>> alignments1 = aligner.align(pro1, rna1)
>>> len(alignments1)
1
>>> alignment1 = next(alignments1)
>>> print(alignment1)
pro1             0 S  G  T  A  R  T  K  L  L  L  L  L  A  A  L  C  A  A  G  G
rna1             0 TCAGGGACTGCGAGAACCAAGCTACTGCTGCTGCTGGCTGCGCTCTGCGCCGCAGGTGGG
<BLANKLINE>
pro1            20 A  L  E   23
rna1            60 GCGCTGGAG 69
<BLANKLINE>
>>> alignment1.coordinates
array([[ 0, 23],
       [ 0, 69]])
>>> alignment1[0]
'SGTARTKLLLLLAALCAAGGALE'
>>> alignment1[1]
'TCAGGGACTGCGAGAACCAAGCTACTGCTGCTGCTGGCTGCGCTCTGCGCCGCAGGTGGGGCGCTGGAG'
>>> alignments2 = aligner.align(pro2, rna2)
>>> len(alignments2)
1
>>> alignment2 = next(alignments2)
>>> print(alignment2)
pro2             0 S  G  T  S  R  T  K  R   8
rna2             0 TCAGGGACTTCGAGAACCAAGCGC 24
<BLANKLINE>
pro2             8 L  L  L  L  A  A  L  G  A  A  G  G  A  L  E   23
rna2            23 CTCCTGCTGCTGGCTGCGCTCGGCGCCGCAGGTGGAGCACTGGAG 68
<BLANKLINE>
>>> alignment2[0]
'SGTSRTKRLLLLAALGAAGGALE'
>>> alignment2[1]
```

```
'TCAGGGACTTCGAGAACCAAGCGCCTCCTGCTGCTGGCTGCGCTCGGCGCCGCAGGTGGAGCACTGGAG'
>>> alignment2.coordinates
array([[ 0,  8,  8, 23],
       [ 0, 24, 23, 68]])
```

While `alignment1` is a continuous alignment of the 69 nucleotides to the 23 amino acids, in `alignment2` we find a -1 frame shift after 24 nucleotides. As `alignment2[1]` contains the nucleotide sequence after applying the -1 frame shift, it is one nucleotide longer than `nuc2` and can be translated directly, resulting in the amino acid sequence `aa2`:

```
>>> from Bio.Seq import translate
>>> len(nuc2)
68
>>> len(alignment2[1])
69
>>> translate(alignment2[1])
'SGTSRTKRLLLLAALGAAGGALE'
>>> _ == aa2
True
```

The alignment score is stored as an attribute on the `alignments1` and `alignments2` iterators, and on the individual alignments `alignment1` and `alignment2`:

```
>>> alignments1.score
23.0
>>> alignment1.score
23.0
>>> alignments2.score
20.0
>>> alignment2.score
20.0
```

where the score of the `rna1-pro1` alignment is equal to the number of aligned amino acids, and the score of the `rna2-pro2` alignment is 3 less due to the penalty for the frame shift. To calculate the alignment score without calculating the alignment itself, the `score` method can be used:

```
>>> score = aligner.score(pro1, rna1)
>>> print(score)
23.0
>>> score = aligner.score(pro2, rna2)
>>> print(score)
20.0
```

7.12.2 Generating a multiple sequence alignment of codon sequences

Suppose we have a third related amino acid sequence and its associated nucleotide sequence:

```
>>> aa3 = Seq("MGTALLLLAALCAAGGALE")
>>> pro3 = SeqRecord(aa3, id="pro3")
>>> nuc3 = Seq("ATGGGAACCGCGCTGCTTTTGCTACTGGCCGCGCTCTGCGCCGCAGGTGGGGCCCTGGAG")
>>> rna3 = SeqRecord(nuc3, id="rna3")
>>> nuc3.translate() == aa3
True
```

As above, we use the `CodonAligner` to align the nucleotide sequence to the amino acid sequence:

```
>>> alignments3 = aligner.align(pro3, rna3)
>>> len(alignments3)
1
>>> alignment3 = next(alignments3)
>>> print(alignment3)
pro3             0 M  G  T  A  L  L  L  L  L  A  A  L  C  A  A  G  G  A  L  E
rna3             0 ATGGGAACCGCGCTGCTTTTGCTACTGGCCGCGCTCTGCGCCGCAGGTGGGGCCCTGGAG
<BLANKLINE>
pro3            20
rna3            60
<BLANKLINE>
```

The three amino acid sequences can be aligned to each other, for example using ClustalW. Here, we create the alignment by hand:

```
>>> import numpy as np
>>> from Bio.Align import Alignment
>>> sequences = [pro1, pro2, pro3]
>>> protein_alignment = Alignment(
...     sequences, coordinates=np.array([[0, 4, 7, 23], [0, 4, 7, 23], [0, 4, 4, 20]])
... )
>>> print(protein_alignment)
pro1             0 SGTARTKLLLLLAALCAAGGALE 23
pro2             0 SGTSRTKRLLLLAALGAAGGALE 23
pro3             0 MGTA---LLLLLAALCAAGGALE 20
<BLANKLINE>
```

Now we can use the `mapall` method on the protein alignment, with the nucleotide-to-protein pairwise alignments as the argument, to obtain the corresponding codon alignment:

```
>>> codon_alignment = protein_alignment.mapall([alignment1, alignment2, alignment3])
>>> print(codon_alignment)
rna1             0 TCAGGGACTGCGAGAACCAAGCTA 24
rna2             0 TCAGGGACTTCGAGAACCAAGCGC 24
rna3             0 ATGGGAACCGCG---------CTG 15
<BLANKLINE>
rna1            24 CTGCTGCTGCTGGCTGCGCTCTGCGCCGCAGGTGGGGCGCTGGAG 69
rna2            23 CTCCTGCTGCTGGCTGCGCTCGGCGCCGCAGGTGGAGCACTGGAG 68
rna3            15 CTTTTGCTACTGGCCGCGCTCTGCGCCGCAGGTGGGGCCCTGGAG 60
<BLANKLINE>
```

7.12.3 Analyzing a codon alignment

Calculating the number of nonsynonymous and synonymous substitutions per site

The most important application of a codon alignment is to estimate the number of nonsynonymous substitutions per site (dN) and synonymous substitutions per site (dS). These can be calculated by the `calculate_dn_ds` function in `Bio.Align.analysis`. This function takes a pairwise codon alignment and input, as well as the optional arguments `method` specifying the calculation method, `codon_table` (defaulting to the Standard Code), the ratio `k` of the transition and transversion rates, and `cfreq` to specify the equilibrium codon frequency. Biopython currently supports three counting based methods (`NG86`, `LWL85`, `YN00`) as well as the maximum likelihood method (`ML`) to estimate dN and dS:

- `NG86`: Nei and Gojobori (1986) [33] (default). With this method, you can also specify the ratio of the transition and transversion rates via the argument `k`, defaulting to `1.0`.

- `LWL85`: Li *et al.* (1985) [28].
- `YN00`: Yang and Nielsen (2000) [51].
- `ML`: Goldman and Yang (1994) [13]. With this method, you can also specify the equilibrium codon frequency via the `cfreq` argument, with the following options:
  - `F1x4`: count the nucleotide frequency in the provided codon sequences, and use it to calculate the background codon frequency;
  - `F3x4`: (default) count the nucleotide frequency separately for the first, second, and third position in the provided codons, and use it to calculate the background codon frequency;
  - `F61`: count the frequency of codons from the provided codon sequences, with a pseudocount of 0.1.

The `calculate_dN_dS` method can be applied to a pairwise codon alignment. In general, the different calculation methods will result in slightly different estimates for dN and dS:

```
>>> from Bio.Align import analysis
>>> pairwise_codon_alignment = codon_alignment[:2]
>>> print(pairwise_codon_alignment)
rna1              0 TCAGGGACTGCGAGAACCAAGCTA 24
                  0 |||||||||.||||||||||||..
rna2              0 TCAGGGACTTCGAGAACCAAGCGC 24
<BLANKLINE>
rna1             24 CTGCTGCTGCTGGCTGCGCTCTGCGCCGCAGGTGGGGCGCTGGAG 69
                 24 ||.||||||||||||||||.||||||||||||.||.|||||| 69
rna2             23 CTCCTGCTGCTGGCTGCGCTCGGCGCCGCAGGTGGAGCACTGGAG 68
<BLANKLINE>
>>> dN, dS = analysis.calculate_dn_ds(pairwise_codon_alignment, method="NG86")
>>> print(dN, dS)  # doctest: +ELLIPSIS
0.067715... 0.201197...
>>> dN, dS = analysis.calculate_dn_ds(pairwise_codon_alignment, method="LWL85")
>>> print(dN, dS)  # doctest: +ELLIPSIS
0.068728... 0.207551...

>>> dN, dS = analysis.calculate_dn_ds(pairwise_codon_alignment, method="YN00")
>>> print(dN, dS)  # doctest: +ELLIPSIS
0.081468... 0.127706...
>>> dN, dS = analysis.calculate_dn_ds(pairwise_codon_alignment, method="ML")
>>> print(dN, dS)  # doctest: +ELLIPSIS
0.069475... 0.205754...
```

For a multiple alignment of codon sequences, you can calculate a matrix of dN and dS values:

```
>>> dN, dS = analysis.calculate_dn_ds_matrix(codon_alignment, method="NG86")
>>> print(dN)
rna1    0.000000
rna2    0.067715    0.000000
rna3    0.060204    0.145469    0.000000
    rna1    rna2    rna3
>>> print(dS)
rna1    0.000000
rna2    0.201198    0.000000
rna3    0.664268    0.798957    0.000000
    rna1    rna2    rna3
```

The objects `dN` and `dS` returned by `calculate_dn_ds_matrix` are instances of the `DistanceMatrix` class in `Bio.Phylo.TreeConstruction`. This function only takes `codon_table` as an optional argument.

From these two sequences, you can create a dN tree and a dS tree using `Bio.Phylo.TreeConstruction`:

```
>>> from Bio.Phylo.TreeConstruction import DistanceTreeConstructor
>>> dn_constructor = DistanceTreeConstructor()
>>> ds_constructor = DistanceTreeConstructor()
>>> dn_tree = dn_constructor.upgma(dN)
>>> ds_tree = ds_constructor.upgma(dS)
>>> print(type(dn_tree))
<class 'Bio.Phylo.BaseTree.Tree'>
>>> print(dn_tree)  # doctest: +ELLIPSIS
Tree(rooted=True)
    Clade(branch_length=0, name='Inner2')
        Clade(branch_length=0.053296..., name='rna2')
        Clade(branch_length=0.023194..., name='Inner1')
            Clade(branch_length=0.0301021..., name='rna3')
            Clade(branch_length=0.0301021..., name='rna1')
>>> print(ds_tree)  # doctest: +ELLIPSIS
Tree(rooted=True)
    Clade(branch_length=0, name='Inner2')
        Clade(branch_length=0.365806..., name='rna3')
        Clade(branch_length=0.265207..., name='Inner1')
            Clade(branch_length=0.100598..., name='rna2')
            Clade(branch_length=0.100598..., name='rna1')
```

Performing the McDonald-Kreitman test

The McDonald-Kreitman test assesses the amount of adaptive evolution by comparing the within species synonymous substitutions and nonsynonymous substitutions to the between species synonymous substitutions and nonsynonymous substitutions to see if they are from the same evolutionary process. The test requires gene sequences sampled from different individuals of the same species. In the following example, we will use Adh gene from fruit fly. The data includes 11 individuals from *Drosophila melanogaster*, 4 individuals from *Drosophila simulans*, and 12 individuals from *Drosophila yakuba*. The protein alignment data and the nucleotide sequences are available in the `Tests/codonalign` directory as the files `adh.aln` and `drosophila.fasta`, respectively, in the Biopython distribution. The function `mktest` in `Bio.Align.analysis` implements the Mcdonald-Kreitman test.

```
>>> from Bio import SeqIO
>>> from Bio import Align
>>> from Bio.Align import CodonAligner
>>> from Bio.Align.analysis import mktest
>>> aligner = CodonAligner()
>>> nucleotide_records = SeqIO.index("drosophila.fasta", "fasta")
>>> for nucleotide_record in nucleotide_records.values():
...     print(nucleotide_record.description)  # doctest: +ELLIPSIS
...
gi|9097|emb|X57361.1| Drosophila simulans (individual c) ...
gi|9099|emb|X57362.1| Drosophila simulans (individual d) ...
gi|9101|emb|X57363.1| Drosophila simulans (individual e) ...
gi|9103|emb|X57364.1| Drosophila simulans (individual f) ...
gi|9217|emb|X57365.1| Drosophila yakuba (individual a) ...
gi|9219|emb|X57366.1| Drosophila yakuba (individual b) ...
```

```
gi|9221|emb|X57367.1| Drosophila yakuba (individual c) ...
gi|9223|emb|X57368.1| Drosophila yakuba (individual d) ...
gi|9225|emb|X57369.1| Drosophila yakuba (individual e) ...
gi|9227|emb|X57370.1| Drosophila yakuba (individual f) ...
gi|9229|emb|X57371.1| Drosophila yakuba (individual g) ...
gi|9231|emb|X57372.1| Drosophila yakuba (individual h) ...
gi|9233|emb|X57373.1| Drosophila yakuba (individual i) ...
gi|9235|emb|X57374.1| Drosophila yakuba (individual j) ...
gi|9237|emb|X57375.1| Drosophila yakuba (individual k) ...
gi|9239|emb|X57376.1| Drosophila yakuba (individual l) ...
gi|156879|gb|M17837.1|DROADHCK D.melanogaster (strain Ja-F) ...
gi|156863|gb|M19547.1|DROADHCC D.melanogaster (strain Af-S) ...
gi|156877|gb|M17836.1|DROADHCJ D.melanogaster (strain Af-F) ...
gi|156875|gb|M17835.1|DROADHCI D.melanogaster (strain Wa-F) ...
gi|156873|gb|M17834.1|DROADHCH D.melanogaster (strain Fr-F) ...
gi|156871|gb|M17833.1|DROADHCG D.melanogaster (strain Fl-F) ...
gi|156869|gb|M17832.1|DROADHCF D.melanogaster (strain Ja-S) ...
gi|156867|gb|M17831.1|DROADHCE D.melanogaster (strain Fl-2S) ...
gi|156865|gb|M17830.1|DROADHCD D.melanogaster (strain Fr-S) ...
gi|156861|gb|M17828.1|DROADHCB D.melanogaster (strain Fl-1S) ...
gi|156859|gb|M17827.1|DROADHCA D.melanogaster (strain Wa-S) ...
>>> protein_alignment = Align.read("adh.aln", "clustal")
>>> len(protein_alignment)
27
>>> print(protein_alignment)  # doctest: +ELLIPSIS
gi|9217|e         0 MAFTLTNKNVVFVAGLGGIGLDTSKELVKRDLKNLVILDRIENPAAIAELKAINPKVTVT
gi|9219|e         0 MAFTLTNKNVVFVAGLGGIGLDTSKELVKRDLKNLVILDRIENPAAIAELKAINPKVTVT
gi|9221|e         0 MAFTLTNKNVVFVAGLGGIGLDTSKELVKRDLKNLVILDRIENPAAIAELKAINPKVTVT
...
gi|156859         0 MSFTLTNKNVIFVAGLGGIGLDTSKELLKRDLKNLVILDRIENPAAIAELKAINPKVTVT
<BLANKLINE>
...
<BLANKLINE>
gi|9217|e       240 GTLEAIQWSKHWDSGI 256
gi|9219|e       240 GTLEAIQWSKHWDSGI 256
gi|9221|e       240 GTLEAIQWSKHWDSGI 256
...
gi|156859       240 GTLEAIQWTKHWDSGI 256
<BLANKLINE>
>>> codon_alignments = []
>>> for protein_record in protein_alignment.sequences:
...     nucleotide_record = nucleotide_records[protein_record.id]
...     alignments = aligner.align(protein_record, nucleotide_record)
...     assert len(alignments) == 1
...     codon_alignment = next(alignments)
...     codon_alignments.append(codon_alignment)
...
>>> print(codon_alignment)  # doctest: +ELLIPSIS
gi|156859         0 M  S  F  T  L  T  N  K  N  V  I  F  V  A  G  L  G  G  I  G
gi|156859         0 ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTTGCCGGTCTGGGAGGCATTGGT
<BLANKLINE>
```

```
gi|156859        20 L  D  T  S  K  E  L  L  K  R  D  L  K  N  L  V  I  L  D  R
gi|156859        60 CTGGACACCAGCAAGGAGCTGCTCAAGCGCGATCTGAAGAACCTGGTGATCCTCGACCGC
<BLANKLINE>
...
<BLANKLINE>
gi|156859       240 G  T  L  E  A  I  Q  W  T  K  H  W  D  S  G  I   256
gi|156859       720 GGCACCCTGGAGGCCATCCAGTGGACCAAGCACTGGGACTCCGGCATC 768
<BLANKLINE>
>>> nucleotide_records.close()  # Close indexed FASTA file
>>> alignment = protein_alignment.mapall(codon_alignments)
>>> print(alignment)  # doctest: +ELLIPSIS
gi|9217|e         0 ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATTGGT
gi|9219|e         0 ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATTGGT
gi|9221|e         0 ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATTGGT
...
gi|156859         0 ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTTGCCGGTCTGGGAGGCATTGGT
<BLANKLINE>
...
<BLANKLINE>
gi|9217|e       720 GGCACCCTGGAGGCCATCCAGTGGTCCAAGCACTGGGACTCCGGCATC 768
gi|9219|e       720 GGCACCCTGGAGGCCATCCAGTGGTCCAAGCACTGGGACTCCGGCATC 768
gi|9221|e       720 GGTACCCTGGAGGCCATCCAGTGGTCCAAGCACTGGGACTCCGGCATC 768
...
gi|156859       720 GGCACCCTGGAGGCCATCCAGTGGACCAAGCACTGGGACTCCGGCATC 768
<BLANKLINE>
>>> unique_species = ["Drosophila simulans", "Drosophila yakuba", "D.melanogaster"]
>>> species = []
>>> for record in alignment.sequences:
...     description = record.description
...     for s in unique_species:
...         if s in description:
...             break
...     else:
...         raise Exception(f"Failed to find species for {description}")
...     species.append(s)
...
>>> print(species)
['Drosophila yakuba', 'Drosophila yakuba', 'Drosophila yakuba', 'Drosophila yakuba', 'Drosophila yakuba
>>> pvalue = mktest(alignment, species)
>>> print(pvalue)  # doctest: +ELLIPSIS
0.00206457...
```

In addition to the multiple codon alignment, the function `mktest` takes as input the species to which each sequence in the alignment belongs to. The codon table can be provided as an optional argument `codon_table`.

Chapter 8

Multiple Sequence Alignment objects

This chapter describes the older `MultipleSeqAlignment` class and the parsers in `Bio.AlignIO` that parse the output of sequence alignment software, generating `MultipleSeqAlignment` objects. By Multiple Sequence Alignments we mean a collection of multiple sequences which have been aligned together – usually with the insertion of gap characters, and addition of leading or trailing gaps – such that all the sequence strings are the same length. Such an alignment can be regarded as a matrix of letters, where each row is held as a `SeqRecord` object internally.

We will introduce the `MultipleSeqAlignment` object which holds this kind of data, and the `Bio.AlignIO` module for reading and writing them as various file formats (following the design of the `Bio.SeqIO` module from the previous chapter). Note that both `Bio.SeqIO` and `Bio.AlignIO` can read and write sequence alignment files. The appropriate choice will depend largely on what you want to do with the data.

The final part of this chapter is about using common multiple sequence alignment tools like ClustalW and MUSCLE from Python, and parsing the results with Biopython.

8.1 Parsing or Reading Sequence Alignments

We have two functions for reading in sequence alignments, `Bio.AlignIO.read()` and `Bio.AlignIO.parse()` which following the convention introduced in `Bio.SeqIO` are for files containing one or multiple alignments respectively.

Using `Bio.AlignIO.parse()` will return an *iterator* which gives `MultipleSeqAlignment` objects. Iterators are typically used in a for loop. Examples of situations where you will have multiple different alignments include resampled alignments from the PHYLIP tool `seqboot`, or multiple pairwise alignments from the EMBOSS tools `water` or `needle`, or Bill Pearson's FASTA tools.

However, in many situations you will be dealing with files which contain only a single alignment. In this case, you should use the `Bio.AlignIO.read()` function which returns a single `MultipleSeqAlignment` object.

Both functions expect two mandatory arguments:

1. The first argument is a *handle* to read the data from, typically an open file (see Section 25.1), or a filename.

2. The second argument is a lower case string specifying the alignment format. As in `Bio.SeqIO` we don't try and guess the file format for you! See `http://biopython.org/wiki/AlignIO` for a full listing of supported formats.

There is also an optional `seq_count` argument which is discussed in Section 8.1.3 below for dealing with ambiguous file formats which may contain more than one alignment.

8.1.1 Single Alignments

As an example, consider the following annotation rich protein alignment in the PFAM or Stockholm file format:

```
# STOCKHOLM 1.0
#=GS COATB_BPIKE/30-81  AC P03620.1
#=GS COATB_BPIKE/30-81  DR PDB; 1ifl ; 1-52;
#=GS Q9T0Q8_BPIKE/1-52  AC Q9T0Q8.1
#=GS COATB_BPI22/32-83  AC P15416.1
#=GS COATB_BPM13/24-72  AC P69541.1
#=GS COATB_BPM13/24-72  DR PDB; 2cpb ; 1-49;
#=GS COATB_BPM13/24-72  DR PDB; 2cps ; 1-49;
#=GS COATB_BPZJ2/1-49   AC P03618.1
#=GS Q9T0Q9_BPFD/1-49   AC Q9T0Q9.1
#=GS Q9T0Q9_BPFD/1-49   DR PDB; 1nh4 A; 1-49;
#=GS COATB_BPIF1/22-73  AC P03619.2
#=GS COATB_BPIF1/22-73  DR PDB; 1ifk ; 1-50;
COATB_BPIKE/30-81             AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA
#=GR COATB_BPIKE/30-81  SS    -HHHHHHHHHHHHHH--HHHHHHHH--HHHHHHHHHHHHHHHHHHHHH----
Q9T0Q8_BPIKE/1-52             AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA
COATB_BPI22/32-83             DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA
COATB_BPM13/24-72             AEGDDP...AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
#=GR COATB_BPM13/24-72  SS    ---S-T...CHCHHHHCCCCTCCCTTCHHHHHHHHHHHHHHHHHHHHCTT--
COATB_BPZJ2/1-49              AEGDDP...AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA
Q9T0Q9_BPFD/1-49              AEGDDP...AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
#=GR Q9T0Q9_BPFD/1-49   SS    ------...-HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH--
COATB_BPIF1/22-73              FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA
#=GR COATB_BPIF1/22-73  SS    XX-HHHH--HHHHHH--HHHHHHH--HHHHHHHHHHHHHHHHHHHHHHH---
#=GC SS_cons                  XHHHHHHHHHHHHHHHHCHHHHHHHHCHHHHHHHHHHHHHHHHHHHHHHC--
#=GC seq_cons                 AEssss...AptAhDSLpspAT-hIu.sWshVsslVsAsluIKLFKKFsSKA
//
```

This is the seed alignment for the Phage_Coat_Gp8 (PF05371) PFAM entry, downloaded from a now out of date release of PFAM from https://pfam.xfam.org/. We can load this file as follows (assuming it has been saved to disk as "PF05371_seed.sth" in the current working directory):

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
```

This code will print out a summary of the alignment:

```
>>> print(alignment)
Alignment with 7 rows and 52 columns
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRL...SKA COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKL...SRA Q9T0Q8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRL...SKA COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA Q9T0Q9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKL...SRA COATB_BPIF1/22-73
```

You'll notice in the above output the sequences have been truncated. We could instead write our own code to format this as we please by iterating over the rows as `SeqRecord` objects:

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> print("Alignment length %i" % alignment.get_alignment_length())
Alignment length 52
>>> for record in alignment:
...     print("%s - %s" % (record.seq, record.id))
...
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA - COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA - Q9T0Q8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA - COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA - COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - Q9T0Q9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA - COATB_BPIF1/22-73
```

You could also call Python's built-in `format` function on the alignment object to show it in a particular file format – see Section 8.2.2 for details.

Did you notice in the raw file above that several of the sequences include database cross-references to the PDB and the associated known secondary structure? Try this:

```
>>> for record in alignment:
...     if record.dbxrefs:
...         print("%s %s" % (record.id, record.dbxrefs))
...
COATB_BPIKE/30-81 ['PDB; 1ifl ; 1-52;']
COATB_BPM13/24-72 ['PDB; 2cpb ; 1-49;', 'PDB; 2cps ; 1-49;']
Q9T0Q9_BPFD/1-49 ['PDB; 1nh4 A; 1-49;']
COATB_BPIF1/22-73 ['PDB; 1ifk ; 1-50;']
```

To have a look at all the sequence annotation, try this:

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

PFAM provide a nice web interface at http://pfam.xfam.org/family/PF05371 which will actually let you download this alignment in several other formats. This is what the file looks like in the FASTA file format:

```
>COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA
>Q9T0Q8_BPIKE/1-52
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA
>COATB_BPI22/32-83
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA
>COATB_BPM13/24-72
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
>COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA
>Q9T0Q9_BPFD/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
>COATB_BPIF1/22-73
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA
```

Note the website should have an option about showing gaps as periods (dots) or dashes, we've shown dashes above. Assuming you download and save this as file "PF05371_seed.faa" then you can load it with almost exactly the same code:

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.faa", "fasta")
>>> print(alignment)
```

All that has changed in this code is the filename and the format string. You'll get the same output as before, the sequences and record identifiers are the same. However, as you should expect, if you check each `SeqRecord` there is no annotation nor database cross-references because these are not included in the FASTA file format.

Note that rather than using the Sanger website, you could have used `Bio.AlignIO` to convert the original Stockholm format file into a FASTA file yourself (see below).

With any supported file format, you can load an alignment in exactly the same way just by changing the format string. For example, use "phylip" for PHYLIP files, "nexus" for NEXUS files or "emboss" for the alignments output by the EMBOSS tools. There is a full listing on the wiki page (http://biopython.org/wiki/AlignIO) and in the built-in documentation (also online):

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

8.1.2 Multiple Alignments

The previous section focused on reading files containing a single alignment. In general however, files can contain more than one alignment, and to read these files we must use the `Bio.AlignIO.parse()` function.

Suppose you have a small alignment in PHYLIP format:

```
    5    6
Alpha      AACAAC
Beta       AACCCC
Gamma      ACCAAC
Delta      CCACCA
Epsilon    CCAAAC
```

If you wanted to bootstrap a phylogenetic tree using the PHYLIP tools, one of the steps would be to create a set of many resampled alignments using the tool `bootseq`. This would give output something like this, which has been abbreviated for conciseness:

```
    5     6
Alpha      AAACCA
Beta       AAACCC
Gamma      ACCCCA
Delta      CCCAAC
Epsilon    CCCAAA
    5     6
Alpha      AAACAA
Beta       AAACCC
Gamma      ACCCAA
Delta      CCCACC
Epsilon    CCCAAA
    5     6
Alpha      AAAAAC
```

```
Beta      AAACCC
Gamma     AACAAC
Delta     CCCCCA
Epsilon   CCCAAC
...
    5     6
Alpha     AAAACC
Beta      ACCCCC
Gamma     AAAACC
Delta     CCCCAA
Epsilon   CAAACC
```

If you wanted to read this in using `Bio.AlignIO` you could use:

```
>>> from Bio import AlignIO
>>> alignments = AlignIO.parse("resampled.phy", "phylip")
>>> for alignment in alignments:
...     print(alignment)
...     print()
...
```

This would give the following output, again abbreviated for display:

```
Alignment with 5 rows and 6 columns
AAACCA Alpha
AAACCC Beta
ACCCCA Gamma
CCCAAC Delta
CCCAAA Epsilon

Alignment with 5 rows and 6 columns
AAACAA Alpha
AAACCC Beta
ACCCAA Gamma
CCCACC Delta
CCCAAA Epsilon

Alignment with 5 rows and 6 columns
AAAAAC Alpha
AAACCC Beta
AACAAC Gamma
CCCCCA Delta
CCCAAC Epsilon

...

Alignment with 5 rows and 6 columns
AAAACC Alpha
ACCCCC Beta
AAAACC Gamma
CCCCAA Delta
CAAACC Epsilon
```

As with the function `Bio.SeqIO.parse()`, using `Bio.AlignIO.parse()` returns an iterator. If you want to keep all the alignments in memory at once, which will allow you to access them in any order, then turn the iterator into a list:

```
>>> from Bio import AlignIO
>>> alignments = list(AlignIO.parse("resampled.phy", "phylip"))
>>> last_align = alignments[-1]
>>> first_align = alignments[0]
```

8.1.3 Ambiguous Alignments

Many alignment file formats can explicitly store more than one alignment, and the division between each alignment is clear. However, when a general sequence file format has been used there is no such block structure. The most common such situation is when alignments have been saved in the FASTA file format. For example consider the following:

```
>Alpha
ACTACGACTAGCTCAG--G
>Beta
ACTACCGCTAGCTCAGAAG
>Gamma
ACTACGGCTAGCACAGAAG
>Alpha
ACTACGACTAGCTCAGG--
>Beta
ACTACCGCTAGCTCAGAAG
>Gamma
ACTACGGCTAGCACAGAAG
```

This could be a single alignment containing six sequences (with repeated identifiers). Or, judging from the identifiers, this is probably two different alignments each with three sequences, which happen to all have the same length.

What about this next example?

```
>Alpha
ACTACGACTAGCTCAG--G
>Beta
ACTACCGCTAGCTCAGAAG
>Alpha
ACTACGACTAGCTCAGG--
>Gamma
ACTACGGCTAGCACAGAAG
>Alpha
ACTACGACTAGCTCAGG--
>Delta
ACTACGGCTAGCACAGAAG
```

Again, this could be a single alignment with six sequences. However this time based on the identifiers we might guess this is three pairwise alignments which by chance have all got the same lengths.

This final example is similar:

```
>Alpha
ACTACGACTAGCTCAG--G
```

```
>XXX
ACTACCGCTAGCTCAGAAG
>Alpha
ACTACGACTAGCTCAGG
>YYY
ACTACGGCAAGCACAGG
>Alpha
--ACTACGAC--TAGCTCAGG
>ZZZ
GGACTACGACAATAGCTCAGG
```

In this third example, because of the differing lengths, this cannot be treated as a single alignment containing all six records. However, it could be three pairwise alignments.

Clearly trying to store more than one alignment in a FASTA file is not ideal. However, if you are forced to deal with these as input files `Bio.AlignIO` can cope with the most common situation where all the alignments have the same number of records. One example of this is a collection of pairwise alignments, which can be produced by the EMBOSS tools `needle` and `water` – although in this situation, `Bio.AlignIO` should be able to understand their native output using "emboss" as the format string.

To interpret these FASTA examples as several separate alignments, we can use `Bio.AlignIO.parse()` with the optional `seq_count` argument which specifies how many sequences are expected in each alignment (in these examples, 3, 2 and 2 respectively). For example, using the third example as the input data:

```
>>> for alignment in AlignIO.parse(handle, "fasta", seq_count=2):
...     print("Alignment length %i" % alignment.get_alignment_length())
...     for record in alignment:
...         print("%s - %s" % (record.seq, record.id))
...     print()
...
```

giving:

```
Alignment length 19
ACTACGACTAGCTCAG--G - Alpha
ACTACCGCTAGCTCAGAAG - XXX

Alignment length 17
ACTACGACTAGCTCAGG - Alpha
ACTACGGCAAGCACAGG - YYY

Alignment length 21
--ACTACGAC--TAGCTCAGG - Alpha
GGACTACGACAATAGCTCAGG - ZZZ
```

Using `Bio.AlignIO.read()` or `Bio.AlignIO.parse()` without the `seq_count` argument would give a single alignment containing all six records for the first two examples. For the third example, an exception would be raised because the lengths differ preventing them being turned into a single alignment.

If the file format itself has a block structure allowing `Bio.AlignIO` to determine the number of sequences in each alignment directly, then the `seq_count` argument is not needed. If it is supplied, and doesn't agree with the file contents, an error is raised.

Note that this optional `seq_count` argument assumes each alignment in the file has the same number of sequences. Hypothetically you may come across stranger situations, for example a FASTA file containing several alignments each with a different number of sequences – although I would love to hear of a real world example of this. Assuming you cannot get the data in a nicer file format, there is no straight forward way

to deal with this using `Bio.AlignIO`. In this case, you could consider reading in the sequences themselves using `Bio.SeqIO` and batching them together to create the alignments as appropriate.

8.2 Writing Alignments

We've talked about using `Bio.AlignIO.read()` and `Bio.AlignIO.parse()` for alignment input (reading files), and now we'll look at `Bio.AlignIO.write()` which is for alignment output (writing files). This is a function taking three arguments: some `MultipleSeqAlignment` objects (or for backwards compatibility the obsolete `Alignment` objects), a handle or filename to write to, and a sequence format.

Here is an example, where we start by creating a few `MultipleSeqAlignment` objects the hard way (by hand, rather than by loading them from a file). Note we create some `SeqRecord` objects to construct the alignment from.

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

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

```
>>> from Bio import AlignIO
>>> AlignIO.write(my_alignments, "my_example.phy", "phylip")
```

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

```
 3 12
Alpha      ACTGCTAGCT AG
Beta       ACT-CTAGCT AG
Gamma      ACTGCTAGDT AG
 3 9
Delta      GTCAGC-AG
```

```
Epislon    GACAGCTAG
Zeta       GTCAGCTAG
 3 13
Eta        ACTAGTACAG CTG
Theta      ACTAGTACAG CT-
Iota       -CTACTACAG GTG
```

Its more common to want to load an existing alignment, and save that, perhaps after some simple manipulation like removing certain rows or columns.

Suppose you wanted to know how many alignments the `Bio.AlignIO.write()` function wrote to the handle? If your alignments were in a list like the example above, you could just use `len(my_alignments)`, however you can't do that when your records come from a generator/iterator. Therefore the `Bio.AlignIO.write()` function returns the number of alignments written to the file.

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

8.2.1 Converting between sequence alignment file formats

Converting between sequence alignment file formats with `Bio.AlignIO` works in the same way as converting between sequence file formats with `Bio.SeqIO` (Section 5.5.2). We load generally the alignment(s) using `Bio.AlignIO.parse()` and then save them using the `Bio.AlignIO.write()` – or just use the `Bio.AlignIO.convert()` helper function.

For this example, we'll load the PFAM/Stockholm format file used earlier and save it as a Clustal W format file:

```
>>> from Bio import AlignIO
>>> count = AlignIO.convert("PF05371_seed.sth", "stockholm", "PF05371_seed.aln", "clustal")
>>> print("Converted %i alignments" % count)
Converted 1 alignments
```

Or, using `Bio.AlignIO.parse()` and `Bio.AlignIO.write()`:

```
>>> from Bio import AlignIO
>>> alignments = AlignIO.parse("PF05371_seed.sth", "stockholm")
>>> count = AlignIO.write(alignments, "PF05371_seed.aln", "clustal")
>>> print("Converted %i alignments" % count)
Converted 1 alignments
```

The `Bio.AlignIO.write()` function expects to be given multiple alignment objects. In the example above we gave it the alignment iterator returned by `Bio.AlignIO.parse()`.

In this case, we know there is only one alignment in the file so we could have used `Bio.AlignIO.read()` instead, but notice we have to pass this alignment to `Bio.AlignIO.write()` as a single element list:

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> AlignIO.write([alignment], "PF05371_seed.aln", "clustal")
```

Either way, you should end up with the same new Clustal W format file "PF05371_seed.aln" with the following content:

```
CLUSTAL X (1.81) multiple sequence alignment
```

```
COATB_BPIKE/30-81                   AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSS
Q9T0Q8_BPIKE/1-52                   AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVS
COATB_BPI22/32-83                   DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSS
COATB_BPM13/24-72                   AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTS
COATB_BPZJ2/1-49                    AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFAS
Q9T0Q9_BPFD/1-49                    AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTS
COATB_BPIF1/22-73                   FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVS

COATB_BPIKE/30-81                   KA
Q9T0Q8_BPIKE/1-52                   RA
COATB_BPI22/32-83                   KA
COATB_BPM13/24-72                   KA
COATB_BPZJ2/1-49                    KA
Q9T0Q9_BPFD/1-49                    KA
COATB_BPIF1/22-73                   RA
```

Alternatively, you could make a PHYLIP format file which we'll name "PF05371_seed.phy":

```
>>> from Bio import AlignIO
>>> AlignIO.convert("PF05371_seed.sth", "stockholm", "PF05371_seed.phy", "phylip")
```

This time the output looks like this:

```
 7 52
COATB_BPIK AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIRLFKKFSS
Q9T0Q8_BPI AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIKLFKKFVS
COATB_BPI2 DGTSTATSYA TEAMNSLKTQ ATDLIDQTWP VVTSVAVAGL AIRLFKKFSS
COATB_BPM1 AEGDDP---A KAAFNSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPZJ AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFAS
Q9T0Q9_BPF AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPIF FAADDATSQA KAAFDSLTAQ ATEMSGYAWA LVVLVVGATV GIKLFKKFVS

           KA
           RA
           KA
           KA
           KA
           KA
           RA
```

One of the big handicaps of the original PHYLIP alignment file format is that the sequence identifiers are strictly truncated at ten characters. In this example, as you can see the resulting names are still unique - but they are not very readable. As a result, a more relaxed variant of the original PHYLIP format is now quite widely used:

```
>>> from Bio import AlignIO
>>> AlignIO.convert("PF05371_seed.sth", "stockholm", "PF05371_seed.phy", "phylip-relaxed")
```

This time the output looks like this, using a longer indentation to allow all the identifiers to be given in full:

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

```
Q9T0Q8_BPIKE/1-52  AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIKLFKKFVS
COATB_BPI22/32-83  DGTSTATSYA TEAMNSLKTQ ATDLIDQTWP VVTSVAVAGL AIRLFKKFSS
COATB_BPM13/24-72  AEGDDP---A KAAFNSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPZJ2/1-49   AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFAS
Q9T0Q9_BPFD/1-49   AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPIF1/22-73  FAADDATSQA KAAFDSLTAQ ATEMSGYAWA LVVLVVGATV GIKLFKKFVS

                   KA
                   RA
                   KA
                   KA
                   KA
                   KA
                   RA
```

If you have to work with the original strict PHYLIP format, then you may need to compress the identifiers somehow – or assign your own names or numbering system. This following bit of code manipulates the record identifiers before saving the output:

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> name_mapping = {}
>>> for i, record in enumerate(alignment):
...     name_mapping[i] = record.id
...     record.id = "seq%i" % i
...
>>> print(name_mapping)
{0: 'COATB_BPIKE/30-81', 1: 'Q9T0Q8_BPIKE/1-52', 2: 'COATB_BPI22/32-83', 3: 'COATB_BPM13/24-72', 4: 'CO
>>> AlignIO.write([alignment], "PF05371_seed.phy", "phylip")
```

This code used a Python dictionary to record a simple mapping from the new sequence system to the original identifier:

```
{
    0: "COATB_BPIKE/30-81",
    1: "Q9T0Q8_BPIKE/1-52",
    2: "COATB_BPI22/32-83",
    # ...
}
```

Here is the new (strict) PHYLIP format output:

```
 7 52
seq0       AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIRLFKKFSS
seq1       AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIKLFKKFVS
seq2       DGTSTATSYA TEAMNSLKTQ ATDLIDQTWP VVTSVAVAGL AIRLFKKFSS
seq3       AEGDDP---A KAAFNSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
seq4       AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFAS
seq5       AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
seq6       FAADDATSQA KAAFDSLTAQ ATEMSGYAWA LVVLVVGATV GIKLFKKFVS

           KA
           RA
```

```
KA
KA
KA
KA
RA
```

In general, because of the identifier limitation, working with *strict* PHYLIP file formats shouldn't be your first choice. Using the PFAM/Stockholm format on the other hand allows you to record a lot of additional annotation too.

8.2.2 Getting your alignment objects as formatted strings

The `Bio.AlignIO` interface is based on handles, which means if you want to get your alignment(s) into a string in a particular file format you need to do a little bit more work (see below). However, you will probably prefer to call Python's built-in `format` function on the alignment object. This takes an output format specification as a single argument, a lower case string which is supported by `Bio.AlignIO` as an output format. For example:

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> print(format(alignment, "clustal"))
CLUSTAL X (1.81) multiple sequence alignment


COATB_BPIKE/30-81                AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSS
Q9T0Q8_BPIKE/1-52                AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVS
COATB_BPI22/32-83                DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSS
...
```

Without an output format specification, `format` returns the same output as `str`.

As described in Section 4.6, the `SeqRecord` object has a similar method using output formats supported by `Bio.SeqIO`.

Internally `format` is calling `Bio.AlignIO.write()` with a `StringIO` handle. You can do this in your own code if for example you are using an older version of Biopython:

```
>>> from io import StringIO
>>> from Bio import AlignIO
>>> alignments = AlignIO.parse("PF05371_seed.sth", "stockholm")
>>> out_handle = StringIO()
>>> AlignIO.write(alignments, out_handle, "clustal")
1
>>> clustal_data = out_handle.getvalue()
>>> print(clustal_data)
CLUSTAL X (1.81) multiple sequence alignment


COATB_BPIKE/30-81                AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSS
Q9T0Q8_BPIKE/1-52                AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVS
COATB_BPI22/32-83                DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSS
COATB_BPM13/24-72                AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTS
...
```

8.3 Manipulating Alignments

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

8.3.1 Slicing alignments

First of all, in some senses the alignment objects act like a Python `list` of `SeqRecord` objects (the rows). With this model in mind hopefully the actions of `len()` (the number of rows) and iteration (each row as a `SeqRecord`) make sense:

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> print("Number of rows: %i" % len(alignment))
Number of rows: 7
>>> for record in alignment:
...     print("%s - %s" % (record.seq, record.id))
...
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA - COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA - Q9T0Q8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA - COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA - COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - Q9T0Q9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA - COATB_BPIF1/22-73
```

You can also use the list-like `append` and `extend` methods to add more rows to the alignment (as `SeqRecord` objects). Keeping the list metaphor in mind, simple slicing of the alignment should also make sense - it selects some of the rows giving back another alignment object:

```
>>> print(alignment)
Alignment with 7 rows and 52 columns
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRL...SKA COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKL...SRA Q9T0Q8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRL...SKA COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA Q9T0Q9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKL...SRA COATB_BPIF1/22-73
>>> print(alignment[3:7])
Alignment with 4 rows and 52 columns
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA Q9T0Q9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKL...SRA COATB_BPIF1/22-73
```

What if you wanted to select by column? Those of you who have used the NumPy matrix or array objects won't be surprised at this - you use a double index.

```
>>> print(alignment[2, 6])
T
```

Using two integer indices pulls out a single letter, short hand for this:

```
>>> print(alignment[2].seq[6])
T
```

You can pull out a single column as a string like this:

```
>>> print(alignment[:, 6])
TTT---T
```

You can also select a range of columns. For example, to pick out those same three rows we extracted earlier, but take just their first six columns:

```
>>> print(alignment[3:6, :6])
Alignment with 3 rows and 6 columns
AEGDDP COATB_BPM13/24-72
AEGDDP COATB_BPZJ2/1-49
AEGDDP Q9T0Q9_BPFD/1-49
```

Leaving the first index as `:` means take all the rows:

```
>>> print(alignment[:, :6])
Alignment with 7 rows and 6 columns
AEPNAA COATB_BPIKE/30-81
AEPNAA Q9T0Q8_BPIKE/1-52
DGTSTA COATB_BPI22/32-83
AEGDDP COATB_BPM13/24-72
AEGDDP COATB_BPZJ2/1-49
AEGDDP Q9T0Q9_BPFD/1-49
FAADDA COATB_BPIF1/22-73
```

This brings us to a neat way to remove a section. Notice columns 7, 8 and 9 which are gaps in three of the seven sequences:

```
>>> print(alignment[:, 6:9])
Alignment with 7 rows and 3 columns
TNY COATB_BPIKE/30-81
TNY Q9T0Q8_BPIKE/1-52
TSY COATB_BPI22/32-83
--- COATB_BPM13/24-72
--- COATB_BPZJ2/1-49
--- Q9T0Q9_BPFD/1-49
TSQ COATB_BPIF1/22-73
```

Again, you can slice to get everything after the ninth column:

```
>>> print(alignment[:, 9:])
Alignment with 7 rows and 43 columns
ATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA COATB_BPIKE/30-81
ATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA Q9T0Q8_BPIKE/1-52
ATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA COATB_BPI22/32-83
AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA COATB_BPM13/24-72
AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA COATB_BPZJ2/1-49
AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA Q9T0Q9_BPFD/1-49
AKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA COATB_BPIF1/22-73
```

Now, the interesting thing is that addition of alignment objects works by column. This lets you do this as a way to remove a block of columns:

```
>>> edited = alignment[:, :6] + alignment[:, 9:]
>>> print(edited)
Alignment with 7 rows and 49 columns
AEPNAAATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA COATB_BPIKE/30-81
AEPNAAATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA Q9T0Q8_BPIKE/1-52
DGTSTAATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA COATB_BPI22/32-83
AEGDDPAKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA COATB_BPM13/24-72
AEGDDPAKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA COATB_BPZJ2/1-49
AEGDDPAKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA Q9T0Q9_BPFD/1-49
FAADDAAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA COATB_BPIF1/22-73
```

Another common use of alignment addition would be to combine alignments for several different genes into a meta-alignment. Watch out though - the identifiers need to match up (see Section 4.8 for how adding `SeqRecord` objects works). You may find it helpful to first sort the alignment rows alphabetically by id:

```
>>> edited.sort()
>>> print(edited)
Alignment with 7 rows and 49 columns
DGTSTAATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA COATB_BPI22/32-83
FAADDAAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA COATB_BPIF1/22-73
AEPNAAATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA COATB_BPIKE/30-81
AEGDDPAKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA COATB_BPM13/24-72
AEGDDPAKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA COATB_BPZJ2/1-49
AEPNAAATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA Q9T0Q8_BPIKE/1-52
AEGDDPAKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA Q9T0Q9_BPFD/1-49
```

Note that you can only add two alignments together if they have the same number of rows.

8.3.2 Alignments as arrays

Depending on what you are doing, it can be more useful to turn the alignment object into an array of letters – and you can do this with NumPy:

```
>>> import numpy as np
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> align_array = np.array(alignment)
>>> print("Array shape %i by %i" % align_array.shape)
Array shape 7 by 52
>>> align_array[:, :10]  # doctest:+ELLIPSIS
array([['A', 'E', 'P', 'N', 'A', 'A', 'T', 'N', 'Y', 'A'],
       ['A', 'E', 'P', 'N', 'A', 'A', 'T', 'N', 'Y', 'A'],
       ['D', 'G', 'T', 'S', 'T', 'A', 'T', 'S', 'Y', 'A'],
       ['A', 'E', 'G', 'D', 'D', 'P', '-', '-', '-', 'A'],
       ['A', 'E', 'G', 'D', 'D', 'P', '-', '-', '-', 'A'],
       ['A', 'E', 'G', 'D', 'D', 'P', '-', '-', '-', 'A'],
       ['F', 'A', 'A', 'D', 'D', 'A', 'T', 'S', 'Q', 'A']],...
```

Note that this leaves the original Biopython alignment object and the NumPy array in memory as separate objects - editing one will not update the other!

8.3.3 Counting substitutions

The `substitutions` property of an alignment reports how often letters in the alignment are substituted for each other. This is calculated by taking all pairs of rows in the alignment, counting the number of times two letters are aligned to each other, and summing this over all pairs. For example,

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment
>>> msa = MultipleSeqAlignment(
...     [
...         SeqRecord(Seq("ACTCCTA"), id="seq1"),
...         SeqRecord(Seq("AAT-CTA"), id="seq2"),
...         SeqRecord(Seq("CCTACT-"), id="seq3"),
...         SeqRecord(Seq("TCTCCTC"), id="seq4"),
...     ]
... )
>>> print(msa)
Alignment with 4 rows and 7 columns
ACTCCTA seq1
AAT-CTA seq2
CCTACT- seq3
TCTCCTC seq4
>>> substitutions = msa.substitutions
>>> print(substitutions)
    A    C    T
A 2.0  4.5  1.0
C 4.5 10.0  0.5
T 1.0  0.5 12.0
<BLANKLINE>
```

As the ordering of pairs is arbitrary, counts are divided equally above and below the diagonal. For example, the 9 alignments of `A` to `C` are stored as 4.5 at position `['A', 'C']` and 4.5 at position `['C', 'A']`. This arrangement helps to make the math easier when calculating a substitution matrix from these counts, as described in Section 7.9.

Note that `msa.substitutions` contains entries for the letters appearing in the alignment only. You can use the `select` method to add entries for missing letters, for example

```
>>> m = substitutions.select("ATCG")
>>> print(m)
    A    T    C   G
A 2.0  1.0  4.5 0.0
T 1.0 12.0  0.5 0.0
C 4.5  0.5 10.0 0.0
G 0.0  0.0  0.0 0.0
<BLANKLINE>
```

This also allows you to change the order of letters in the alphabet.

8.3.4 Calculating summary information

Once you have an alignment, you are very likely going to want to find out information about it. Instead of trying to have all of the functions that can generate information about an alignment in the alignment object itself, we've tried to separate out the functionality into separate classes, which act on the alignment.

Getting ready to calculate summary information about an object is quick to do. Let's say we've got an alignment object called `alignment`, for example read in using `Bio.AlignIO.read(...)` as described in Chapter 8. All we need to do to get an object that will calculate summary information is:

```
>>> from Bio.Align import AlignInfo
>>> summary_align = AlignInfo.SummaryInfo(msa)
```

The `summary_align` object is very useful, and will do the following neat things for you:

1. Calculate a quick consensus sequence – see section 8.3.5
2. Get a position specific score matrix for the alignment – see section 8.3.6
3. Calculate the information content for the alignment – see section 8.3.7
4. Generate information on substitutions in the alignment – section 7.9 details using this to generate a substitution matrix.

8.3.5 Calculating a quick consensus sequence

The `SummaryInfo` object, described in section 8.3.4, provides functionality to calculate a quick consensus of an alignment. Assuming we've got a `SummaryInfo` object called `summary_align` we can calculate a consensus by doing:

```
>>> consensus = summary_align.dumb_consensus()
>>> consensus
Seq('XCTXCTX')
```

As the name suggests, this is a really simple consensus calculator, and will just add up all of the residues at each point in the consensus, and if the most common value is higher than some threshold value will add the common residue to the consensus. If it doesn't reach the threshold, it adds an ambiguity character to the consensus. The returned consensus object is a `Seq` object.

You can adjust how `dumb_consensus` works by passing optional parameters:

the threshold This is the threshold specifying how common a particular residue has to be at a position before it is added. The default is 0.7 (meaning 70%).

the ambiguous character This is the ambiguity character to use. The default is 'N'.

Alternatively, you can convert the multiple sequence alignment object `msa` to a new-style `Alignment` object (see section 6.1) by using the `alignment` attribute (see section 8.4):

```
>>> alignment = msa.alignment
```

You can then create a `Motif` object (see section 17.1):

```
>>> from Bio.motifs import Motif
>>> motif = Motif("ACGT", alignment)
```

and obtain a quick consensus sequence:

```
>>> motif.consensus
Seq('ACTCCTA')
```

The `motif.counts.calculate_consensus` method (see section 17.1.2) lets you specify in detail how the consensus sequence should be calculated. For example,

```
>>> motif.counts.calculate_consensus(identity=0.7)
'NCTNCTN'
```

8.3.6 Position Specific Score Matrices

Position specific score matrices (PSSMs) summarize the alignment information in a different way than a consensus, and may be useful for different tasks. Basically, a PSSM is a count matrix. For each column in the alignment, the number of each alphabet letters is counted and totaled. The totals are displayed relative to some representative sequence along the left axis. This sequence may be the consensus sequence, but can also be any sequence in the alignment.

For instance for the alignment above:

```
>>> print(msa)
Alignment with 4 rows and 7 columns
ACTCCTA seq1
AAT-CTA seq2
CCTACT- seq3
TCTCCTC seq4
```

we get a PSSM with the consensus sequence along the side using

```
>>> my_pssm = summary_align.pos_specific_score_matrix(consensus, chars_to_ignore=["N"])
>>> print(my_pssm)
    A   C   T
X  2.0 1.0 1.0
C  1.0 3.0 0.0
T  0.0 0.0 4.0
X  1.0 2.0 0.0
C  0.0 4.0 0.0
T  0.0 0.0 4.0
X  2.0 1.0 0.0
<BLANKLINE>
```

where we ignore any `N` ambiguity residues when calculating the PSSM.

Two notes should be made about this:

1. To maintain strictness with the alphabets, you can only include characters along the top of the PSSM that are in the alphabet of the alignment object. Gaps are not included along the top axis of the PSSM.

2. The sequence passed to be displayed along the left side of the axis does not need to be the consensus. For instance, if you wanted to display the second sequence in the alignment along this axis, you would need to do:

   ```
   >>> second_seq = msa[1]
   >>> my_pssm = summary_align.pos_specific_score_matrix(second_seq, chars_to_ignore=["N"])
   >>> print(my_pssm)
       A   C   T
   A  2.0 1.0 1.0
   A  1.0 3.0 0.0
   T  0.0 0.0 4.0
   -  1.0 2.0 0.0
   C  0.0 4.0 0.0
   T  0.0 0.0 4.0
   A  2.0 1.0 0.0
   <BLANKLINE>
   ```

The command above returns a `PSSM` object. You can access any element of the PSSM by subscripting like `your_pssm[sequence_number][residue_count_name]`. For instance, to get the counts for the 'A' residue in the second element of the above PSSM you would do:

```
>>> print(my_pssm[5]["T"])
4.0
```

The structure of the PSSM class hopefully makes it easy both to access elements and to pretty print the matrix.

Alternatively, you can convert the multiple sequence alignment object `msa` to a new-style `Alignment` object (see section 6.1) by using the `alignment` attribute (see section 8.4):

```
>>> alignment = msa.alignment
```

You can then create a `Motif` object (see section 17.1):

```
>>> from Bio.motifs import Motif
>>> motif = Motif("ACGT", alignment)
```

and obtain the counts of each nucleotide in each position:

```
>>> counts = motif.counts
>>> print(counts)
       0     1     2     3     4     5     6
A:   2.00  1.00  0.00  1.00  0.00  0.00  2.00
C:   1.00  3.00  0.00  2.00  4.00  0.00  1.00
G:   0.00  0.00  0.00  0.00  0.00  0.00  0.00
T:   1.00  0.00  4.00  0.00  0.00  4.00  0.00
<BLANKLINE>
>>> print(counts["T"][5])
4.0
```

8.3.7 Information Content

A potentially useful measure of evolutionary conservation is the information content of a sequence.

A useful introduction to information theory targeted towards molecular biologists can be found at http://www.lecb.ncifcrf.gov/~toms/paper/primer/. For our purposes, we will be looking at the information content of a consensus sequence, or a portion of a consensus sequence. We calculate information content at a particular column in a multiple sequence alignment using the following formula:

$$IC_j = \sum_{i=1}^{N_a} P_{ij} \log\left(\frac{P_{ij}}{Q_i}\right)$$

where:

- IC_j – The information content for the j-th column in an alignment.
- N_a – The number of letters in the alphabet.
- P_{ij} – The frequency of a particular letter i in the j-th column (i. e. if G occurred 3 out of 6 times in an alignment column, this would be 0.5)
- Q_i – The expected frequency of a letter i. This is an optional argument, usage of which is left at the user's discretion. By default, it is automatically assigned to $0.05 = 1/20$ for a protein alphabet, and $0.25 = 1/4$ for a nucleic acid alphabet. This is for getting the information content without any assumption of prior distributions. When assuming priors, or when using a non-standard alphabet, you should supply the values for Q_i.

Well, now that we have an idea what information content is being calculated in Biopython, let's look at how to get it for a particular region of the alignment.

First, we need to use our alignment to get an alignment summary object, which we'll assume is called `summary_align` (see section 8.3.4) for instructions on how to get this. Once we've got this object, calculating the information content for a region is as easy as:

```
>>> e_freq_table = {"A": 0.3, "G": 0.2, "T": 0.3, "C": 0.2}
>>> info_content = summary_align.information_content(
...     2, 6, e_freq_table=e_freq_table, chars_to_ignore=["N"]
... )
>>> info_content  # doctest:+ELLIPSIS
6.3910647...
```

Now, `info_content` will contain the relative information content over the region [2:6] in relation to the expected frequencies.

The value return is calculated using base 2 as the logarithm base in the formula above. You can modify this by passing the parameter `log_base` as the base you want:

```
>>> info_content = summary_align.information_content(
...     2, 6, e_freq_table=e_freq_table, log_base=10, chars_to_ignore=["N"]
... )
>>> info_content  # doctest:+ELLIPSIS
1.923902...
```

By default nucleotide or amino acid residues with a frequency of 0 in a column are not take into account when the relative information column for that column is computed. If this is not the desired result, you can use `pseudo_count` instead.

```
>>> info_content = summary_align.information_content(
...     2, 6, e_freq_table=e_freq_table, chars_to_ignore=["N", "-"], pseudo_count=1
... )
>>> info_content  # doctest:+ELLIPSIS
4.299651...
```

In this case, the observed frequency P_{ij} of a particular letter i in the j-th column is computed as follows:

$$P_{ij} = \frac{n_{ij} + k \times Q_i}{N_j + k}$$

where:

- k – the pseudo count you pass as argument.
- k – the pseudo count you pass as argument.
- Q_i – The expected frequency of the letter i as described above.

Well, now you are ready to calculate information content. If you want to try applying this to some real life problems, it would probably be best to dig into the literature on information content to get an idea of how it is used. Hopefully your digging won't reveal any mistakes made in coding this function!

8.4 Getting a new-style Alignment object

Use the `alignment` property to create a new-style `Alignment` object (see section 6.1) from an old-style `MultipleSeqAlignment` object:

```
>>> type(msa)
<class 'Bio.Align.MultipleSeqAlignment'>
>>> print(msa)
Alignment with 4 rows and 7 columns
ACTCCTA seq1
AAT-CTA seq2
CCTACT- seq3
TCTCCTC seq4
>>> alignment = msa.alignment
>>> type(alignment)
<class 'Bio.Align.Alignment'>
>>> print(alignment)
seq1              0 ACTCCTA 7
seq2              0 AAT-CTA 6
seq3              0 CCTACT- 6
seq4              0 TCTCCTC 7
<BLANKLINE>
```

Note that the `alignment` property creates and returns a new `Alignment` object that is consistent with the information stored in the `MultipleSeqAlignment` object at the time the `Alignment` object is created. Any changes to the `MultipleSeqAlignment` after calling the `alignment` property will not propagate to the `Alignment` object. However, you can of course call the `alignment` property again to create a new `Alignment` object consistent with the updated `MultipleSeqAlignment` object.

8.5 Calculating a substitution matrix from a multiple sequence alignment

You can create your own substitution matrix from an alignment. In this example, we'll first read a protein sequence alignment from the Clustalw file protein.aln (also available online here)

```
>>> from Bio import AlignIO
>>> filename = "protein.aln"
>>> msa = AlignIO.read(filename, "clustal")
```

Section 8.6.1 contains more information on doing this.

The `substitutions` property of the alignment stores the number of times different residues substitute for each other:

```
>>> observed_frequencies = msa.substitutions
```

To make the example more readable, we'll select only amino acids with polar charged side chains:

```
>>> observed_frequencies = observed_frequencies.select("DEHKR")
>>> print(observed_frequencies)
       D      E      H      K      R
D 2360.0  255.5    7.5    0.5   25.0
E  255.5 3305.0   16.5   27.0    2.0
```

```
H    7.5   16.5 1235.0   16.0    8.5
K    0.5   27.0   16.0 3218.0  116.5
R   25.0    2.0    8.5  116.5 2079.0
<BLANKLINE>
```

Rows and columns for other amino acids were removed from the matrix.

Next, we normalize the matrix:

```
>>> import numpy as np
>>> observed_frequencies /= np.sum(observed_frequencies)
```

Summing over rows or columns gives the relative frequency of occurrence of each residue:

```
>>> residue_frequencies = np.sum(observed_frequencies, 0)
>>> print(residue_frequencies.format("%.4f"))
D 0.2015
E 0.2743
H 0.0976
K 0.2569
R 0.1697
<BLANKLINE>
>>> np.sum(residue_frequencies)
1.0
```

The expected frequency of residue pairs is then

```
>>> expected_frequencies = np.dot(
...     residue_frequencies[:, None], residue_frequencies[None, :]
... )
>>> print(expected_frequencies.format("%.4f"))
      D      E      H      K      R
D 0.0406 0.0553 0.0197 0.0518 0.0342
E 0.0553 0.0752 0.0268 0.0705 0.0465
H 0.0197 0.0268 0.0095 0.0251 0.0166
K 0.0518 0.0705 0.0251 0.0660 0.0436
R 0.0342 0.0465 0.0166 0.0436 0.0288
<BLANKLINE>
```

Here, `residue_frequencies[:, None]` creates a 2D array consisting of a single column with the values of `residue_frequencies`, and `residue_frequencies[None, :]` a 2D array with these values as a single row. Taking their dot product (inner product) creates a matrix of expected frequencies where each entry consists of two `residue_frequencies` values multiplied with each other. For example, `expected_frequencies['D', 'E']` is equal to `residue_frequencies['D'] * residue_frequencies['E']`.

We can now calculate the log-odds matrix by dividing the observed frequencies by the expected frequencies and taking the logarithm:

```
>>> m = np.log2(observed_frequencies / expected_frequencies)
>>> print(m)
     D    E    H     K    R
D  2.1 -1.5 -5.1 -10.4 -4.2
E -1.5  1.7 -4.4  -5.1 -8.3
H -5.1 -4.4  3.3  -4.4 -4.7
K -10.4 -5.1 -4.4   1.9 -2.3
R -4.2 -8.3 -4.7  -2.3  2.5
<BLANKLINE>
```

This matrix can be used as the substitution matrix when performing alignments. For example,

```
>>> from Bio.Align import PairwiseAligner
>>> aligner = PairwiseAligner()
>>> aligner.substitution_matrix = m
>>> aligner.gap_score = -3.0
>>> alignments = aligner.align("DEHEK", "DHHKK")
>>> print(alignments[0])
target            0 DEHEK 5
                  0 |.|.| 5
query             0 DHHKK 5
<BLANKLINE>
>>> print("%.2f" % alignments.score)
-2.18
>>> score = m["D", "D"] + m["E", "H"] + m["H", "H"] + m["E", "K"] + m["K", "K"]
>>> print("%.2f" % score)
-2.18
```

8.6 Alignment Tools

There are *lots* of algorithms out there for aligning sequences, both pairwise alignments and multiple sequence alignments. These calculations are relatively slow, and you generally wouldn't want to write such an algorithm in Python. For pairwise alignments, you can use Biopython's `PairwiseAligner` (see Chapter 7), which is implemented in C and therefore fast. Alternatively, you can run an external alignment program by invoking it from Python. Normally you would:

1. Prepare an input file of your unaligned sequences, typically this will be a FASTA file which you might create using `Bio.SeqIO` (see Chapter 5).
2. Run the alignment program by running its command using Python's `subprocess` module.
3. Read the output from the tool, i.e. your aligned sequences, typically using `Bio.AlignIO` (see earlier in this chapter).

Here, we will show a few examples of this workflow.

8.6.1 ClustalW

ClustalW is a popular command line tool for multiple sequence alignment (there is also a graphical interface called ClustalX). Before trying to use ClustalW from within Python, you should first try running the ClustalW tool yourself by hand at the command line, to familiarize yourself the other options.

For the most basic usage, all you need is to have a FASTA input file, such as opuntia.fasta (available online or in the Doc/examples subdirectory of the Biopython source code). This is a small FASTA file containing seven prickly-pear DNA sequences (from the cactus family *Opuntia*). By default ClustalW will generate an alignment and guide tree file with names based on the input FASTA file, in this case `opuntia.aln` and `opuntia.dnd`, but you can override this or make it explicit:

```
>>> import subprocess
>>> cmd = "clustalw2 -infile=opuntia.fasta"
>>> results = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, text=True)
```

Notice here we have given the executable name as `clustalw2`, indicating we have version two installed, which has a different filename to version one (`clustalw`, the default). Fortunately both versions support the same set of arguments at the command line (and indeed, should be functionally identical).

You may find that even though you have ClustalW installed, the above command doesn't work – you may get a message about "command not found" (especially on Windows). This indicated that the ClustalW executable is not on your PATH (an environment variable, a list of directories to be searched). You can either update your PATH setting to include the location of your copy of ClustalW tools (how you do this will depend on your OS), or simply type in the full path of the tool. Remember, in Python strings `\n` and `\t` are by default interpreted as a new line and a tab – which is why we're put a letter "r" at the start for a raw string that isn't translated in this way. This is generally good practice when specifying a Windows style file name.

```
>>> import os
>>> clustalw_exe = r"C:\Program Files\new clustal\clustalw2.exe"
>>> assert os.path.isfile(clustalw_exe), "Clustal W executable missing"
>>> cmd = clustalw_exe + " -infile=opuntia.fasta"
>>> results = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, text=True)
```

Now, at this point it helps to know about how command line tools "work". When you run a tool at the command line, it will often print text output directly to screen. This text can be captured or redirected, via two "pipes", called standard output (the normal results) and standard error (for error messages and debug messages). There is also standard input, which is any text fed into the tool. These names get shortened to stdin, stdout and stderr. When the tool finishes, it has a return code (an integer), which by convention is zero for success, while a non-zero return code indicates that an error has occurred.

In the example of ClustalW above, when run at the command line all the important output is written directly to the output files. Everything normally printed to screen while you wait is captured in `results.stdout` and `results.stderr`, while the return code is stored in `results.returncode`.

What we care about are the two output files, the alignment and the guide tree. We didn't tell ClustalW what filenames to use, but it defaults to picking names based on the input file. In this case the output should be in the file `opuntia.aln`. You should be able to work out how to read in the alignment using `Bio.AlignIO` by now:

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("opuntia.aln", "clustal")
>>> print(align)
Alignment with 7 rows and 906 columns
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.1|AF191
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273291|gb|AF191665.1|AF191
```

In case you are interested (and this is an aside from the main thrust of this chapter), the `opuntia.dnd` file ClustalW creates is just a standard Newick tree file, and `Bio.Phylo` can parse these:

```
>>> from Bio import Phylo
>>> tree = Phylo.read("opuntia.dnd", "newick")
>>> Phylo.draw_ascii(tree)
                            ________________ gi|6273291|gb|AF191665.1|AF191665
  ____________________________|
 |                            |    _______ gi|6273290|gb|AF191664.1|AF191664
 |                            |___|
 |                                |______ gi|6273289|gb|AF191663.1|AF191663
 |
```

```
_|_________________ gi|6273287|gb|AF191661.1|AF191661
 |
 |__________ gi|6273286|gb|AF191660.1|AF191660
 |
 |    __ gi|6273285|gb|AF191659.1|AF191659
 |___|
     | gi|6273284|gb|AF191658.1|AF191658
<BLANKLINE>
```

Chapter 16 covers Biopython's support for phylogenetic trees in more depth.

8.6.2 MUSCLE

MUSCLE is a more recent multiple sequence alignment tool than ClustalW. As before, we recommend you try using MUSCLE from the command line before trying to run it from Python.

For the most basic usage, all you need is to have a FASTA input file, such as opuntia.fasta (available online or in the Doc/examples subdirectory of the Biopython source code). You can then tell MUSCLE to read in this FASTA file, and write the alignment to an output file named `opuntia.txt`:

```
>>> import subprocess
>>> cmd = "muscle -align opuntia.fasta -output opuntia.txt"
>>> results = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, text=True)
```

MUSCLE will output the alignment as a FASTA file (using gapped sequences). The `Bio.AlignIO` module is able to read this alignment using `format="fasta"`:

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("opuntia.txt", "fasta")
>>> print(align)
Alignment with 7 rows and 906 columns
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.1|AF191663
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273291|gb|AF191665.1|AF191665
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.1|AF191664
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.1|AF191661
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.1|AF191660
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.1|AF191659
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.1|AF191658
```

You can also set the other optional parameters; see MUSCLE's built-in help for details.

8.6.3 EMBOSS needle and water

The EMBOSS suite includes the `water` and `needle` tools for Smith-Waterman algorithm local alignment, and Needleman-Wunsch global alignment. The tools share the same style interface, so switching between the two is trivial – we'll just use `needle` here.

Suppose you want to do a global pairwise alignment between two sequences, prepared in FASTA format as follows:

```
>HBA_HUMAN
MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHG
KKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTP
AVHASLDKFLASVSTVLTSKYR
```

in a file `alpha.faa`, and secondly in a file `beta.faa`:

```
>HBB_HUMAN
MVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPK
VKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLAHHFG
KEFTPPVQAAYQKVVAGVANALAHKYH
```

You can find copies of these example files with the Biopython source code under the `Doc/examples/` directory.

The command to align these two sequences against each other using `needle` is as follows:

```
needle -outfile=needle.txt -asequence=alpha.faa -bsequence=beta.faa -gapopen=10 -gapextend=0.5
```

Why not try running this by hand at the command prompt? You should see it does a pairwise comparison and records the output in the file `needle.txt` (in the default EMBOSS alignment file format).

Even if you have EMBOSS installed, running this command may not work – you might get a message about "command not found" (especially on Windows). This probably means that the EMBOSS tools are not on your PATH environment variable. You can either update your PATH setting, or simply use the full path to the tool, for example:

```
C:\EMBOSS\needle.exe -outfile=needle.txt -asequence=alpha.faa -bsequence=beta.faa -gapopen=10 -gapexten
```

Next we want to use Python to run this command for us. As explained above, for full control, we recommend you use Python's built-in `subprocess` module:

```
>>> import sys
>>> import subprocess
>>> cmd = "needle -outfile=needle.txt -asequence=alpha.faa -bsequence=beta.faa -gapopen=10 -gapextend=0
>>> results = subprocess.run(
...     cmd,
...     stdout=subprocess.PIPE,
...     stderr=subprocess.PIPE,
...     text=True,
...     shell=(sys, platform != "win32"),
... )
>>> print(results.stdout)

>>> print(results.stderr)
Needleman-Wunsch global alignment of two sequences
```

Next we can load the output file with `Bio.AlignIO` as discussed earlier in this chapter, as the `emboss` format:

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("needle.txt", "emboss")
>>> print(align)
Alignment with 2 rows and 149 columns
MV-LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTY...KYR HBA_HUMAN
MVHLTPEEKSAVTALWGKV--NVDEVGGEALGRLLVVYPWTQRF...KYH HBB_HUMAN
```

In this example, we told EMBOSS to write the output to a file, but you *can* tell it to write the output to stdout instead (useful if you don't want a temporary output file to get rid of – use `outfile=stdout` argument):

```
>>> cmd = "needle -outfile=stdout -asequence=alpha.faa -bsequence=beta.faa -gapopen=10 -gapextend=0.5"
>>> child = subprocess.Popen(
...     cmd,
```

```
...     stdout=subprocess.PIPE,
...     stderr=subprocess.PIPE,
...     text=True,
...     shell=(sys.platform != "win32"),
... )
>>> align = AlignIO.read(child.stdout, "emboss")
>>> print(align)
Alignment with 2 rows and 149 columns
MV-LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTY...KYR HBA_HUMAN
MVHLTPEEKSAVTALWGKV--NVDEVGGEALGRLLVVYPWTQRF...KYH HBB_HUMAN
```

Similarly, it is possible to read *one* of the inputs from stdin (e.g. `asequence="stdin"`).

This has only scratched the surface of what you can do with `needle` and `water`. One useful trick is that the second file can contain multiple sequences (say five), and then EMBOSS will do five pairwise alignments.

Chapter 9

Pairwise alignments using pairwise2

Please note that Bio.pairwise2 was deprecated in Release 1.80. As an alternative, please consider using `Bio.Align.PairwiseAligner` (described in Chapter 7).

`Bio.pairwise2` contains essentially the same algorithms as `water` (local) and `needle` (global) from the EMBOSS suite (see above) and should return the same results. The `pairwise2` module has undergone some optimization regarding speed and memory consumption recently (Biopython versions >1.67) so that for short sequences (global alignments: ~2000 residues, local alignments ~600 residues) it's faster (or equally fast) to use `pairwise2` than calling EMBOSS' `water` or `needle` via the command line tools.

Suppose you want to do a global pairwise alignment between the same two hemoglobin sequences from above (`HBA_HUMAN`, `HBB_HUMAN`) stored in `alpha.faa` and `beta.faa`:

```
>>> from Bio import pairwise2
>>> from Bio import SeqIO
>>> seq1 = SeqIO.read("alpha.faa", "fasta")
>>> seq2 = SeqIO.read("beta.faa", "fasta")
>>> alignments = pairwise2.align.globalxx(seq1.seq, seq2.seq)
```

As you see, we call the alignment function with `align.globalxx`. The tricky part are the last two letters of the function name (here: `xx`), which are used for decoding the scores and penalties for matches (and mismatches) and gaps. The first letter decodes the match score, e.g. `x` means that a match counts 1 while mismatches have no costs. With `m` general values for either matches or mismatches can be defined (for more options see Biopython's API). The second letter decodes the cost for gaps; `x` means no gap costs at all, with `s` different penalties for opening and extending a gap can be assigned. So, `globalxx` means that only matches between both sequences are counted.

Our variable `alignments` now contains a list of alignments (at least one) which have the same optimal score for the given conditions. In our example this are 80 different alignments with the score 72 (`Bio.pairwise2` will return up to 1000 alignments). Have a look at one of these alignments:

```
>>> len(alignments)
80
>>> print(alignments[0])  # doctest:+ELLIPSIS
Alignment(seqA='MV-LSPADKTNV---K-A--A-WGKVGAHAG...YR-', seqB='MVHL-----T--PEEKSAVTALWGKV----...Y-H', sc
```

Each alignment is a named tuple consisting of the two aligned sequences, the score, the start and the end positions of the alignment (in global alignments the start is always 0 and the end the length of the alignment). `Bio.pairwise2` has a function `format_alignment` for a nicer printout:

```
>>> print(pairwise2.format_alignment(*alignments[0]))  # doctest:+ELLIPSIS
MV-LSPADKTNV---K-A--A-WGKVGAHAG---EY-GA-EALE-RMFLSF----PTTK-TY--F...YR-
```

```
|| |     |     | |  | ||||        |  |  |||  |  |      |    |   |...|
MVHL-----T--PEEKSAVTALWGKV-----NVDE-VG-GEAL-GR--L--LVVYP---WT-QRF...Y-H
  Score=72
<BLANKLINE>
```

Since Biopython 1.77 the required parameters can be supplied with keywords. The last example can now also be written as:

```
>>> alignments = pairwise2.align.globalxx(sequenceA=seq1.seq, sequenceB=seq2.seq)
```

Better alignments are usually obtained by penalizing gaps: higher costs for opening a gap and lower costs for extending an existing gap. For amino acid sequences match scores are usually encoded in matrices like `PAM` or `BLOSUM`. Thus, a more meaningful alignment for our example can be obtained by using the BLOSUM62 matrix, together with a gap open penalty of 10 and a gap extension penalty of 0.5 (using `globalds`):

```
>>> from Bio import pairwise2
>>> from Bio import SeqIO
>>> from Bio.Align import substitution_matrices
>>> blosum62 = substitution_matrices.load("BLOSUM62")
>>> seq1 = SeqIO.read("alpha.faa", "fasta")
>>> seq2 = SeqIO.read("beta.faa", "fasta")
>>> alignments = pairwise2.align.globalds(seq1.seq, seq2.seq, blosum62, -10, -0.5)
>>> len(alignments)
2
>>> print(pairwise2.format_alignment(*alignments[0]))
MV-LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTY...KYR
|| |.|..|..|.|.|||| ......|............|.......||.
MVHLTPEEKSAVTALWGKV-NVDEVGGEALGRLLVVYPWTQRFF...KYH
  Score=292.5
```

This alignment has the same score that we obtained earlier with EMBOSS needle using the same sequences and the same parameters.

Local alignments are called similarly with the function `align.localXX`, where again XX stands for a two letter code for the match and gap functions:

```
>>> from Bio import pairwise2
>>> from Bio.Align import substitution_matrices
>>> blosum62 = substitution_matrices.load("BLOSUM62")
>>> alignments = pairwise2.align.localds("LSPADKTNVKAA", "PEEKSAV", blosum62, -10, -1)
>>> print(pairwise2.format_alignment(*alignments[0]))
3 PADKTNV
  |..|..|
1 PEEKSAV
  Score=16
<BLANKLINE>
```

In recent Biopython versions, `format_alignment` will only print the aligned part of a local alignment (together with the start positions in 1-based notation, as shown in the above example). If you are also interested in the non- aligned parts of the sequences, use the keyword-parameter `full_sequences=True`:

```
>>> from Bio import pairwise2
>>> from Bio.Align import substitution_matrices
```

```
>>> blosum62 = substitution_matrices.load("BLOSUM62")
>>> alignments = pairwise2.align.localds("LSPADKTNVKAA", "PEEKSAV", blosum62, -10, -1)
>>> print(pairwise2.format_alignment(*alignments[0], full_sequences=True))
LSPADKTNVKAA
  |..|..|
--PEEKSAV---
  Score=16
<BLANKLINE>
```

Note that local alignments must, as defined by Smith & Waterman, have a positive score (>0). Thus, `pairwise2` may return no alignments if no score >0 has been obtained. Also, `pairwise2` will not report alignments which are the result of the addition of zero-scoring extensions on either site. In the next example, the pairs serine/aspartic acid (S/D) and lysine/asparagine (K/N) both have a match score of 0. As you see, the aligned part has not been extended:

```
>>> from Bio import pairwise2
>>> from Bio.Align import substitution_matrices
>>> blosum62 = substitution_matrices.load("BLOSUM62")
>>> alignments = pairwise2.align.localds("LSSPADKTNVKKAA", "DDPEEKSAVNN", blosum62, -10, -1)
>>> print(pairwise2.format_alignment(*alignments[0]))
4 PADKTNV
  |..|..|
3 PEEKSAV
  Score=16
<BLANKLINE>
```

Instead of supplying a complete match/mismatch matrix, the match code `m` allows for easy defining general match/mismatch values. The next example uses match/mismatch scores of 5/-4 and gap penalties (open/extend) of 2/0.5 using `localms`:

```
>>> alignments = pairwise2.align.localms("AGAACT", "GAC", 5, -4, -2, -0.5)
>>> print(pairwise2.format_alignment(*alignments[0]))
2 GAAC
  | ||
1 G-AC
  Score=13
<BLANKLINE>
```

One useful keyword argument of the `Bio.pairwise2.align` functions is `score_only`. When set to `True` it will only return the score of the best alignment(s), but in a significantly shorter time. It will also allow the alignment of longer sequences before a memory error is raised. Another useful keyword argument is `one_alignment_only=True` which will also result in some speed gain.

Unfortunately, `Bio.pairwise2` does not work with Biopython's multiple sequence alignment objects (yet). However, the module has some interesting advanced features: you can define your own match and gap functions (interested in testing affine logarithmic gap costs?), gap penalties and end gaps penalties can be different for both sequences, sequences can be supplied as lists (useful if you have residues that are encoded by more than one character), etc. These features are hard (if at all) to realize with other alignment tools. For more details see the modules documentation in Biopython's API.

Chapter 10

BLAST

Hey, everybody loves BLAST right? I mean, geez, how can it get any easier to do comparisons between one of your sequences and every other sequence in the known world? But, of course, this section isn't about how cool BLAST is, since we already know that. It is about the problem with BLAST – it can be really difficult to deal with the volume of data generated by large runs, and to automate BLAST runs in general.

Fortunately, the Biopython folks know this only too well, so they've developed lots of tools for dealing with BLAST and making things much easier. This section details how to use these tools and do useful things with them.

Dealing with BLAST can be split up into two steps, both of which can be done from within Biopython. Firstly, running BLAST for your query sequence(s), and getting some output. Secondly, parsing the BLAST output in Python for further analysis.

Your first introduction to running BLAST was probably via the NCBI web-service. In fact, there are lots of ways you can run BLAST, which can be categorized in several ways. The most important distinction is running BLAST locally (on your own machine), and running BLAST remotely (on another machine, typically the NCBI servers). We're going to start this chapter by invoking the NCBI online BLAST service from within a Python script.

NOTE: The following Chapter 11 describes `Bio.SearchIO`. We intend this to ultimately replace the older `Bio.Blast` module, as it provides a more general framework handling other related sequence searching tools as well. However, for now you can use either that or the older `Bio.Blast` module for dealing with NCBI BLAST.

10.1 Running BLAST over the Internet

We use the function `qblast()` in the `Bio.Blast.NCBIWWW` module to call the online version of BLAST. This has three non-optional arguments:

- The first argument is the blast program to use for the search, as a lower case string. The options and descriptions of the programs are available at https://blast.ncbi.nlm.nih.gov/Blast.cgi. Currently `qblast` only works with blastn, blastp, blastx, tblast and tblastx.
- The second argument specifies the databases to search against. Again, the options for this are available on the NCBI Guide to BLAST ftp://ftp.ncbi.nlm.nih.gov/pub/factsheets/HowTo_BLASTGuide.pdf.
- The third argument is a string containing your query sequence. This can either be the sequence itself, the sequence in fasta format, or an identifier like a GI number.

NCBI guidelines, from https://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastDocs&DOC_TYPE=DeveloperInfo state:

1. Do not contact the server more often than once every 10 seconds.
2. Do not poll for any single RID more often than once a minute.
3. Use the URL parameter email and tool, so that the NCBI can contact you if there is a problem.
4. Run scripts weekends or between 9 pm and 5 am Eastern time on weekdays if more than 50 searches will be submitted.

To fulfill the third point, one can set the `NCBIWWW.email` variable.

```
>>> from Bio.Blast import NCBIWWW
>>> NCBIWWW.email = "A.N.Other@example.com"
```

The `qblast` function also takes a number of other option arguments, which are basically analogous to the different parameters you can set on the BLAST web page. We'll just highlight a few of them here:

- The argument `url_base` sets the base URL for running BLAST over the internet. By default it connects to the NCBI, but one can use this to connect to an instance of NCBI BLAST running in the cloud. Please refer to the documentation for the `qblast` function for further details.
- The `qblast` function can return the BLAST results in various formats, which you can choose with the optional `format_type` keyword: `"HTML"`, `"Text"`, `"ASN.1"`, or `"XML"`. The default is `"XML"`, as that is the format expected by the parser, described in section 10.3 below.
- The argument `expect` sets the expectation or e-value threshold.

For more about the optional BLAST arguments, we refer you to the NCBI's own documentation, or that built into Biopython:

```
>>> from Bio.Blast import NCBIWWW
>>> help(NCBIWWW.qblast)
```

Note that the default settings on the NCBI BLAST website are not quite the same as the defaults on QBLAST. If you get different results, you'll need to check the parameters (e.g., the expectation value threshold and the gap values).

For example, if you have a nucleotide sequence you want to search against the nucleotide database (nt) using BLASTN, and you know the GI number of your query sequence, you can use:

```
>>> from Bio.Blast import NCBIWWW
>>> result_handle = NCBIWWW.qblast("blastn", "nt", "8332116")
```

Alternatively, if we have our query sequence already in a FASTA formatted file, we just need to open the file and read in this record as a string, and use that as the query argument:

```
>>> from Bio.Blast import NCBIWWW
>>> fasta_string = open("m_cold.fasta").read()
>>> result_handle = NCBIWWW.qblast("blastn", "nt", fasta_string)
```

We could also have read in the FASTA file as a `SeqRecord` and then supplied just the sequence itself:

```
>>> from Bio.Blast import NCBIWWW
>>> from Bio import SeqIO
>>> record = SeqIO.read("m_cold.fasta", format="fasta")
>>> result_handle = NCBIWWW.qblast("blastn", "nt", record.seq)
```

Supplying just the sequence means that BLAST will assign an identifier for your sequence automatically. You might prefer to use the `SeqRecord` object's format method to make a FASTA string (which will include the existing identifier):

```
>>> from Bio.Blast import NCBIWWW
>>> from Bio import SeqIO
>>> record = SeqIO.read("m_cold.fasta", format="fasta")
>>> result_handle = NCBIWWW.qblast("blastn", "nt", record.format("fasta"))
```

This approach makes more sense if you have your sequence(s) in a non-FASTA file format which you can extract using `Bio.SeqIO` (see Chapter 5).

Whatever arguments you give the `qblast()` function, you should get back your results in a handle object (by default in XML format). The next step would be to parse the XML output into Python objects representing the search results (Section 10.3), but you might want to save a local copy of the output file first. I find this especially useful when debugging my code that extracts info from the BLAST results (because re-running the online search is slow and wastes the NCBI computer time).

We need to be a bit careful since we can use `result_handle.read()` to read the BLAST output only once – calling `result_handle.read()` again returns an empty string.

```
>>> with open("my_blast.xml", "w") as out_handle:
...     out_handle.write(result_handle.read())
...
>>> result_handle.close()
```

After doing this, the results are in the file `my_blast.xml` and the original handle has had all its data extracted (so we closed it). However, the `parse` function of the BLAST parser (described in 10.3) takes a file-handle-like object, so we can just open the saved file for input:

```
>>> result_handle = open("my_blast.xml")
```

Now that we've got the BLAST results back into a handle again, we are ready to do something with them, so this leads us right into the parsing section (see Section 10.3 below). You may want to jump ahead to that now ....

10.2 Running BLAST locally

10.2.1 Introduction

Running BLAST locally (as opposed to over the internet, see Section 10.1) has at least major two advantages:

- Local BLAST may be faster than BLAST over the internet;
- Local BLAST allows you to make your own database to search for sequences against.

Dealing with proprietary or unpublished sequence data can be another reason to run BLAST locally. You may not be allowed to redistribute the sequences, so submitting them to the NCBI as a BLAST query would not be an option.

Unfortunately, there are some major drawbacks too – installing all the bits and getting it setup right takes some effort:

- Local BLAST requires command line tools to be installed.
- Local BLAST requires (large) BLAST databases to be setup (and potentially kept up to date).

To further confuse matters there are several different BLAST packages available, and there are also other tools which can produce imitation BLAST output files, such as BLAT.

10.2.2 Standalone NCBI BLAST+

The "new" NCBI BLAST+ suite was released in 2009. This replaces the old NCBI "legacy" BLAST package (see below).

This section will show briefly how to use these tools from within Python. If you have already read or tried the alignment tool examples in Section 8.6 this should all seem quite straightforward. First, we construct a command line string (as you would type in at the command line prompt if running standalone BLAST by hand). Then we can execute this command from within Python.

For example, taking a FASTA file of gene nucleotide sequences, you might want to run a BLASTX (translation) search against the non-redundant (NR) protein database. Assuming you (or your systems administrator) has downloaded and installed the NR database, you might run:

```
$ blastx -query opuntia.fasta -db nr -out opuntia.xml -evalue 0.001 -outfmt 5
```

This should run BLASTX against the NR database, using an expectation cut-off value of 0.001 and produce XML output to the specified file (which we can then parse). On my computer this takes about six minutes - a good reason to save the output to a file so you can repeat any analysis as needed.

From within python we can use the `subprocess` module to build the command line string, and run it:

```
>>> import subprocess
>>> cmd = "blastx -query opuntia.fasta -db nr -out opuntia.xml"
>>> cmd += " -evalue 0.001 -outfmt 5"
>>> subprocess.run(cmd, shell=True)
```

In this example there shouldn't be any output from BLASTX to the terminal. You may want to check the output file `opuntia.xml` has been created.

As you may recall from earlier examples in the tutorial, the `opuntia.fasta` contains seven sequences, so the BLAST XML output should contain multiple results. Therefore use `Bio.Blast.NCBIXML.parse()` to parse it as described below in Section 10.3.

10.2.3 Other versions of BLAST

NCBI BLAST+ (written in C++) was first released in 2009 as a replacement for the original NCBI "legacy" BLAST (written in C) which is no longer being updated. There were a lot of changes – the old version had a single core command line tool `blastall` which covered multiple different BLAST search types (which are now separate commands in BLAST+), and all the command line options were renamed. Biopython's wrappers for the NCBI "legacy" BLAST tools have been deprecated and will be removed in a future release. To try to avoid confusion, we do not cover calling these old tools from Biopython in this tutorial.

You may also come across Washington University BLAST (WU-BLAST), and its successor, Advanced Biocomputing BLAST (AB-BLAST, released in 2009, not free/open source). These packages include the command line tools `wu-blastall` and `ab-blastall`, which mimicked `blastall` from the NCBI "legacy" BLAST suite. Biopython does not currently provide wrappers for calling these tools, but should be able to parse any NCBI compatible output from them.

10.3 Parsing BLAST output

As mentioned above, BLAST can generate output in various formats, such as XML, HTML, and plain text. Originally, Biopython had parsers for BLAST plain text and HTML output, as these were the only output formats offered at the time. Unfortunately, the BLAST output in these formats kept changing, each time breaking the Biopython parsers. Our HTML BLAST parser has been removed, while the deprecated plain text BLAST parser is now only available via `Bio.SearchIO`. Use it at your own risk, it may or may not work, depending on which BLAST version you're using.

As keeping up with changes in BLAST became a hopeless endeavor, especially with users running different BLAST versions, we now recommend to parse the output in XML format, which can be generated by recent versions of BLAST. Not only is the XML output more stable than the plain text and HTML output, it is also much easier to parse automatically, making Biopython a whole lot more stable.

You can get BLAST output in XML format in various ways. For the parser, it doesn't matter how the output was generated, as long as it is in the XML format.

- You can use Biopython to run BLAST over the internet, as described in section 10.1.
- You can use Biopython to run BLAST locally, as described in section 10.2.
- You can do the BLAST search yourself on the NCBI site through your web browser, and then save the results. You need to choose XML as the format in which to receive the results, and save the final BLAST page you get (you know, the one with all of the interesting results!) to a file.
- You can also run BLAST locally without using Biopython, and save the output in a file. Again, you need to choose XML as the format in which to receive the results.

The important point is that you do not have to use Biopython scripts to fetch the data in order to be able to parse it. Doing things in one of these ways, you then need to get a handle to the results. In Python, a handle is just a nice general way of describing input to any info source so that the info can be retrieved using `read()` and `readline()` functions (see Section 25.1).

If you followed the code above for interacting with BLAST through a script, then you already have `result_handle`, the handle to the BLAST results. For example, using a GI number to do an online search:

```
>>> from Bio.Blast import NCBIWWW
>>> result_handle = NCBIWWW.qblast("blastn", "nt", "8332116")
```

If instead you ran BLAST some other way, and have the BLAST output (in XML format) in the file `my_blast.xml`, all you need to do is to open the file for reading:

```
>>> result_handle = open("my_blast.xml")
```

Now that we've got a handle, we are ready to parse the output. The code to parse it is really quite small. If you expect a single BLAST result (i.e., you used a single query):

```
>>> from Bio.Blast import NCBIXML
>>> blast_record = NCBIXML.read(result_handle)
```

or, if you have lots of results (i.e., multiple query sequences):

```
>>> from Bio.Blast import NCBIXML
>>> blast_records = NCBIXML.parse(result_handle)
```

Just like `Bio.SeqIO` and `Bio.Align` (see Chapters 5 and 6), we have a pair of input functions, `read` and `parse`, where `read` is for when you have exactly one object, and `parse` is an iterator for when you can have lots of objects – but instead of getting `SeqRecord` or `MultipleSeqAlignment` objects, we get BLAST record objects.

To be able to handle the situation where the BLAST file may be huge, containing thousands of results, `NCBIXML.parse()` returns an iterator. In plain English, an iterator allows you to step through the BLAST output, retrieving BLAST records one by one for each BLAST search result:

```
>>> from Bio.Blast import NCBIXML
>>> blast_records = NCBIXML.parse(result_handle)
>>> blast_record = next(blast_records)
```

```
# ... do something with blast_record
>>> blast_record = next(blast_records)
# ... do something with blast_record
>>> blast_record = next(blast_records)
# ... do something with blast_record
>>> blast_record = next(blast_records)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
# No further records
```

Or, you can use a `for`-loop:

```
>>> for blast_record in blast_records:
...     pass  # Do something with blast_record
...
```

Note though that you can step through the BLAST records only once. Usually, from each BLAST record you would save the information that you are interested in. If you want to save all returned BLAST records, you can convert the iterator into a list:

```
>>> blast_records = list(blast_records)
```

Now you can access each BLAST record in the list with an index as usual. If your BLAST file is huge though, you may run into memory problems trying to save them all in a list.

Usually, you'll be running one BLAST search at a time. Then, all you need to do is to pick up the first (and only) BLAST record in `blast_records`:

```
>>> from Bio.Blast import NCBIXML
>>> blast_records = NCBIXML.parse(result_handle)
>>> blast_record = next(blast_records)
```

or more elegantly:

```
>>> from Bio.Blast import NCBIXML
>>> blast_record = NCBIXML.read(result_handle)
```

I guess by now you're wondering what is in a BLAST record.

10.4 The BLAST record class

A BLAST Record contains everything you might ever want to extract from the BLAST output. Right now we'll just show an example of how to get some info out of the BLAST report, but if you want something in particular that is not described here, look at the info on the record class in detail, and take a gander into the code or automatically generated documentation – the docstrings have lots of good info about what is stored in each piece of information.

To continue with our example, let's just print out some summary info about all hits in our blast report greater than a particular threshold. The following code does this:

```
>>> E_VALUE_THRESH = 0.04

>>> for alignment in blast_record.alignments:
...     for hsp in alignment.hsps:
...         if hsp.expect < E_VALUE_THRESH:
```

```
...             print("****Alignment****")
...             print("sequence:", alignment.title)
...             print("length:", alignment.length)
...             print("e value:", hsp.expect)
...             print(hsp.query[0:75] + "...")
...             print(hsp.match[0:75] + "...")
...             print(hsp.sbjct[0:75] + "...")
...
```

This will print out summary reports like the following:

```
****Alignment****
sequence: >gb|AF283004.1|AF283004 Arabidopsis thaliana cold acclimation protein WCOR413-like protein
alpha form mRNA, complete cds
length: 783
e value: 0.034
tacttgttgatattggatcgaacaaactggagaaccaacatgctcacgtcacttttagtcccttacatattcctc...
||||||||| | |||||||||||  || ||||  || || ||||||||  ||||||  |  | ||||||||  ||| ||...
tacttgttggtgttggatcgaaccaattggaagacgaatatgctcacatcacttctcattccttacatcttcttc...
```

Basically, you can do anything you want to with the info in the BLAST report once you have parsed it. This will, of course, depend on what you want to use it for, but hopefully this helps you get started on doing what you need to do!

An important consideration for extracting information from a BLAST report is the type of objects that the information is stored in. In Biopython, the parsers return `Record` objects, either `Blast` or `PSIBlast` depending on what you are parsing. These objects are defined in `Bio.Blast.Record` and are quite complete.

Figures 10.1 and 10.2 and are my attempts at UML class diagrams for the `Blast` and `PSIBlast` record classes. The PSIBlast record object is similar, but has support for the rounds that are used in the iteration steps of PSIBlast.

If you are good at UML and see mistakes/improvements that can be made, please let me know.

10.5 Dealing with PSI-BLAST

You can run the standalone version of PSI-BLAST (the legacy NCBI command line tool `blastpgp`, or its replacement `psiblast`) directly from the command line or using python's `subprocess` module.

At the time of writing, the NCBI do not appear to support tools running a PSI-BLAST search via the internet.

Note that the `Bio.Blast.NCBIXML` parser can read the XML output from current versions of PSI-BLAST, but information like which sequences in each iteration is new or reused isn't present in the XML file.

10.6 Dealing with RPS-BLAST

You can run the standalone version of RPS-BLAST (either the legacy NCBI command line tool `rpsblast`, or its replacement with the same name) directly from the command line or using python's `subprocess` module.

At the time of writing, the NCBI do not appear to support tools running an RPS-BLAST search via the internet.

You can use the `Bio.Blast.NCBIXML` parser to read the XML output from current versions of RPS-BLAST.

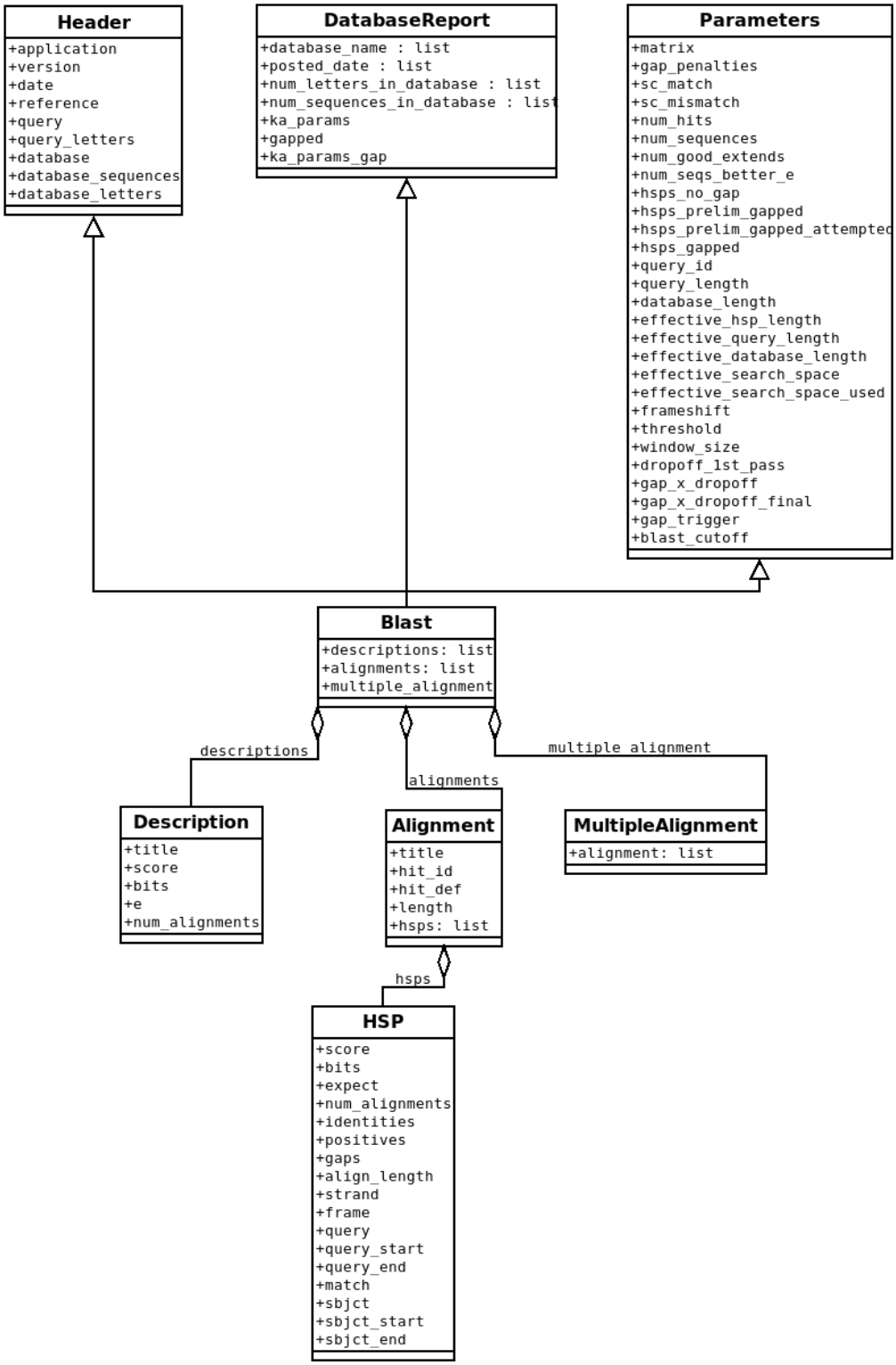


Figure 10.1: Class diagram for the Blast Record class representing all of the info in a BLAST report

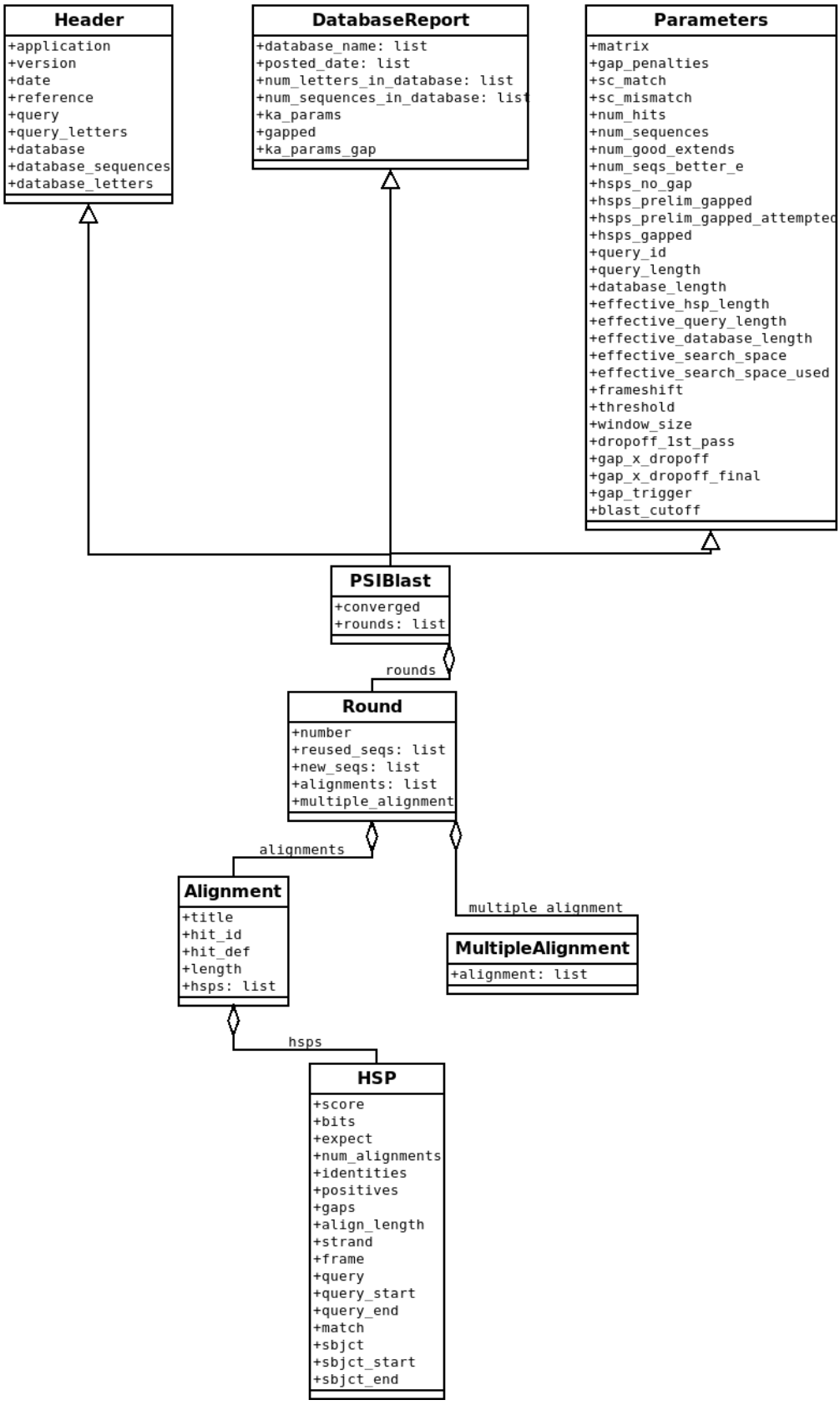


Figure 10.2: Class diagram for the PSIBlast Record class.

Chapter 11

BLAST and other sequence search tools

Biological sequence identification is an integral part of bioinformatics. Several tools are available for this, each with their own algorithms and approaches, such as BLAST (arguably the most popular), FASTA, HMMER, and many more. In general, these tools usually use your sequence to search a database of potential matches. With the growing number of known sequences (hence the growing number of potential matches), interpreting the results becomes increasingly hard as there could be hundreds or even thousands of potential matches. Naturally, manual interpretation of these searches' results is out of the question. Moreover, you often need to work with several sequence search tools, each with its own statistics, conventions, and output format. Imagine how daunting it would be when you need to work with multiple sequences using multiple search tools.

We know this too well ourselves, which is why we created the `Bio.SearchIO` submodule in Biopython. `Bio.SearchIO` allows you to extract information from your search results in a convenient way, while also dealing with the different standards and conventions used by different search tools. The name `SearchIO` is a homage to BioPerl's module of the same name.

In this chapter, we'll go through the main features of `Bio.SearchIO` to show what it can do for you. We'll use two popular search tools along the way: BLAST and BLAT. They are used merely for illustrative purposes, and you should be able to adapt the workflow to any other search tools supported by `Bio.SearchIO` in a breeze. You're very welcome to follow along with the search output files we'll be using. The BLAST output file can be downloaded here, and the BLAT output file here or are included with the Biopython source code under the `Doc/examples/` folder. Both output files were generated using this sequence:

```
>mystery_seq
CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
```

The BLAST result is an XML file generated using `blastn` against the NCBI `refseq_rna` database. For BLAT, the sequence database was the February 2009 `hg19` human genome draft and the output format is PSL.

We'll start from an introduction to the `Bio.SearchIO` object model. The model is the representation of your search results, thus it is core to `Bio.SearchIO` itself. After that, we'll check out the main functions in `Bio.SearchIO` that you may often use.

Now that we're all set, let's go to the first step: introducing the core object model.

11.1 The SearchIO object model

Despite the wildly differing output styles among many sequence search tools, it turns out that their underlying concept is similar:

- The output file may contain results from one or more search queries.
- In each search query, you will see one or more hits from the given search database.
- In each database hit, you will see one or more regions containing the actual sequence alignment between your query sequence and the database sequence.
- Some programs like BLAT or Exonerate may further split these regions into several alignment fragments (or blocks in BLAT and possibly exons in exonerate). This is not something you always see, as programs like BLAST and HMMER do not do this.

Realizing this generality, we decided use it as base for creating the `Bio.SearchIO` object model. The object model consists of a nested hierarchy of Python objects, each one representing one concept outlined above. These objects are:

- `QueryResult`, to represent a single search query.
- `Hit`, to represent a single database hit. `Hit` objects are contained within `QueryResult` and in each `QueryResult` there is zero or more `Hit` objects.
- `HSP` (short for high-scoring pair), to represent region(s) of significant alignments between query and hit sequences. `HSP` objects are contained within `Hit` objects and each `Hit` has one or more `HSP` objects.
- `HSPFragment`, to represent a single contiguous alignment between query and hit sequences. `HSPFragment` objects are contained within `HSP` objects. Most sequence search tools like BLAST and HMMER unify `HSP` and `HSPFragment` objects as each `HSP` will only have a single `HSPFragment`. However there are tools like BLAT and Exonerate that produce `HSP` containing multiple `HSPFragment`. Don't worry if this seems a tad confusing now, we'll elaborate more on these two objects later on.

These four objects are the ones you will interact with when you use `Bio.SearchIO`. They are created using one of the main `Bio.SearchIO` methods: `read`, `parse`, `index`, or `index_db`. The details of these methods are provided in later sections. For this section, we'll only be using read and parse. These functions behave similarly to their `Bio.SeqIO` and `Bio.AlignIO` counterparts:

- `read` is used for search output files with a single query and returns a `QueryResult` object
- `parse` is used for search output files with multiple queries and returns a generator that yields `QueryResult` objects

With that settled, let's start probing each `Bio.SearchIO` object, beginning with `QueryResult`.

11.1.1 QueryResult

The QueryResult object represents a single search query and contains zero or more Hit objects. Let's see what it looks like using the BLAST file we have:

```
>>> from Bio import SearchIO
>>> blast_qresult = SearchIO.read("my_blast.xml", "blast-xml")
>>> print(blast_qresult)
Program: blastn (2.2.27+)
  Query: 42291 (61)
         mystery_seq
 Target: refseq_rna
   Hits: ----  -----  ------------------------------------------------------------
            #  # HSP  ID + description
```