# Contents

1 Introduction ........................................... 10
   1.1 What is Biopython? .................................. 10
   1.2 What can I find in the Biopython package ................. 10
   1.3 Installing Biopython ................................... 11
   1.4 Frequently Asked Questions (FAQ) ......................... 11

2 Quick Start – What can you do with Biopython? ................. 15
   2.1 General overview of what Biopython provides ............... 15
   2.2 Working with sequences ................................ 15
   2.3 A usage example ....................................... 16
   2.4 Parsing sequence file formats ............................ 16
      2.4.1 Simple FASTA parsing example ...................... 17
      2.4.2 Simple GenBank parsing example ..................... 17
      2.4.3 I love parsing – please don’t stop talking about it! 18
   2.5 Connecting with biological databases ...................... 18
   2.6 What to do next ...................................... 19

3 Sequence objects ........................................ 20
   3.1 Sequences act like strings ................................ 20
   3.2 Slicing a sequence ..................................... 21
   3.3 Turning Seq objects into strings ......................... 22
   3.4 Concatenating or adding sequences ......................... 22
   3.5 Changing case ......................................... 23
   3.6 Nucleotide sequences and (reverse) complements ........... 23
   3.7 Transcription .......................................... 24
   3.8 Translation ............................................ 25
   3.9 Translation Tables ...................................... 27
   3.10 Comparing Seq objects .................................. 29
   3.11 Sequences with unknown sequence contents ............... 29
   3.12 Sequences with partially defined sequence contents ....... 29
   3.13 MutableSeq objects .................................... 30
   3.14 Finding subsequences ................................... 31
   3.15 Working with strings directly ............................ 32

4 Sequence annotation objects ................................ 34
   4.1 The SeqRecord object .................................. 34
   4.2 Creating a SeqRecord ................................. 35
      4.2.1 SeqRecord objects from scratch .................... 35
      4.2.2 SeqRecord objects from FASTA files ............... 36
      4.2.3 SeqRecord objects from GenBank files .............. 37
4.3 Feature, location and position objects ........................................ 38
  4.3.1 SeqFeature objects ......................................................... 38
  4.3.2 Positions and locations .................................................... 39
  4.3.3 Sequence described by a feature or location ......................... 41

4.4 Comparison ............................................................................. 42

4.5 References ............................................................................. 43

4.6 The format method ................................................................. 43

4.7 Slicing a SeqRecord ............................................................... 44

4.8 Adding SeqRecord objects ...................................................... 46

4.9 Reverse-complementing SeqRecord objects ............................... 48

5 Sequence Input/Output ............................................................... 50
  5.1 Parsing or Reading Sequences ............................................... 50
    5.1.1 Reading Sequence Files .................................................. 50
    5.1.2 Iterating over the records in a sequence file ....................... 51
    5.1.3 Getting a list of the records in a sequence file ................. 52
    5.1.4 Extracting data ............................................................ 53
    5.1.5 Modifying data ............................................................. 55
  5.2 Parsing sequences from compressed files .................................. 55
  5.3 Parsing sequences from the net .............................................. 56
    5.3.1 Parsing GenBank records from the net ............................. 57
    5.3.2 Parsing SwissProt sequences from the net ....................... 58
  5.4 Sequence files as Dictionaries ............................................... 58
    5.4.1 Sequence files as Dictionaries – In memory ...................... 59
    5.4.2 Sequence files as Dictionaries – Indexed files .................. 62
    5.4.3 Sequence files as Dictionaries – Database indexed files .... 63
    5.4.4 Indexing compressed files .............................................. 64
    5.4.5 Discussion .................................................................. 65
  5.5 Writing Sequence Files .......................................................... 66
    5.5.1 Round trips ................................................................. 67
    5.5.2 Converting between sequence file formats ....................... 68
    5.5.3 Converting a file of sequences to their reverse complements .. 69
    5.5.4 Getting your SeqRecord objects as formatted strings .......... 70
  5.6 Low level FASTA and FASTQ parsers ..................................... 70

6 Sequence alignments ................................................................. 72
  6.1 Alignment objects ............................................................... 72
    6.1.1 Creating an Alignment object from sequences and coordinates .. 72
    6.1.2 Creating an Alignment object from aligned sequences ........... 73
    6.1.3 Common alignment attributes ....................................... 74
  6.2 Slicing and indexing an alignment ......................................... 75
  6.3 Getting information about the alignment ............................... 77
    6.3.1 Alignment shape .......................................................... 77
    6.3.2 Comparing alignments ............................................... 77
    6.3.3 Finding the indices of aligned sequences ......................... 77
    6.3.4 Counting identities, mismatches, and gaps ..................... 79
    6.3.5 Letter frequencies ...................................................... 79
    6.3.6 Substitutions ................................................................ 79
    6.3.7 Alignments as arrays ................................................. 80
  6.4 Operations on an alignment .................................................. 81
    6.4.1 Sorting an alignment .................................................... 81
    6.4.2 Reverse-complementing the alignment ............................. 81
6.4.3 Adding alignments .................................................. 82
6.4.4 Mapping a pairwise sequence alignment ............................. 83
6.4.5 Mapping a multiple sequence alignment .............................. 86
6.5 The Alignments class .................................................. 88
6.6 Reading and writing alignments ......................................... 89
6.6.1 Reading alignments .................................................. 90
6.6.2 Writing alignments .................................................. 90
6.6.3 Printing alignments .................................................. 90
6.7 Alignment file formats ................................................ 92
6.7.1 Aligned FASTA ..................................................... 92
6.7.2 ClustalW ............................................................ 94
6.7.3 Stockholm .......................................................... 96
6.7.4 PHYLIP output files ................................................ 99
6.7.5 EMBOSS ............................................................ 102
6.7.6 GCG Multiple Sequence Format (MSF) ............................ 104
6.7.7 Exonerate ............................................................ 106
6.7.8 NEXUS ............................................................... 109
6.7.9 Tabular output from BLAST or FASTA ............................ 110
6.7.10 HH-suite output files .............................................. 112
6.7.11 A2M ................................................................. 116
6.7.12 Mauve eXtended Multi-FastA (xmfa) format ................... 117
6.7.13 Sequence Alignment/Map (SAM) .................................. 122
6.7.14 Browser Extensible Data (BED) .................................. 126
6.7.15 bigBed ............................................................... 128
6.7.16 Pattern Space Layout (PSL) ....................................... 131
6.7.17 bigPsl ............................................................... 133
6.7.18 Multiple Alignment Format (MAF) ................................ 136
6.7.19 bigMaf ............................................................... 139
6.7.20 UCSC chain file format ............................................ 142

7 Pairwise sequence alignment ............................................. 145
7.1 Basic usage .................................................................. 145
7.2 The pairwise aligner object .............................................. 149
7.3 Substitution scores ...................................................... 150
7.4 Affine gap scores ....................................................... 151
7.5 General gap scores ...................................................... 152
7.6 Using a pre-defined substitution matrix and gap scores .......... 154
7.7 Iterating over alignments ............................................. 155
7.8 Aligning to the reverse strand ....................................... 157
7.9 Substitution matrices ................................................... 158
7.9.1 Array objects ......................................................... 158
7.9.2 Calculating a substitution matrix from a pairwise sequence alignment .................................................. 158
7.9.3 Calculating a substitution matrix from a multiple sequence alignment .................................................. 161
7.9.4 Reading Array objects from file ................................ 165
7.9.5 Loading predefined substitution matrices ....................... 167
7.10 Examples .................................................................. 168
7.11 Generalized pairwise alignments ..................................... 170
7.11.1 Generalized pairwise alignments using a substitution matrix and alphabet .............................................. 170
7.11.2 Generalized pairwise alignments using match/mismatch scores and an alphabet .................................. 171
7.11.3 Generalized pairwise alignments using match/mismatch scores and integer sequences ..................... 172
7.11.4 Generalized pairwise alignments using a substitution matrix and integer sequences ......................... 173
7.12 Codon alignments ...................................................... 174
### 8 Multiple Sequence Alignment objects

8.1 Parsing or Reading Sequence Alignments .......................... 182
    8.1.1 Single Alignments ...................................... 183
    8.1.2 Multiple Alignments .................................... 185
    8.1.3 Ambiguous Alignments .................................... 187
8.2 Writing Alignments .................................................. 189
    8.2.1 Converting between sequence alignment file formats .......... 190
    8.2.2 Getting your alignment objects as formatted strings .......... 193
8.3 Manipulating Alignments ............................................ 194
    8.3.1 Slicing alignments ........................................ 194
    8.3.2 Alignments as arrays ..................................... 196
    8.3.3 Counting substitutions .................................... 197
    8.3.4 Calculating summary information ............................ 197
    8.3.5 Calculating a quick consensus sequence ....................... 198
    8.3.6 Position Specific Score Matrices ............................ 199
    8.3.7 Information Content ....................................... 200
8.4 Getting a new-style Alignment object ............................. 202
8.5 Calculating a substitution matrix from a multiple sequence alignment .......... 202
8.6 Alignment Tools ..................................................... 204
    8.6.1 ClustalW .................................................. 204
    8.6.2 MUSCLE ................................................... 206
    8.6.3 EMBOSS needle and water .................................. 206

### 9 Pairwise alignments using pairwise2

### 10 BLAST

10.1 Running BLAST over the Internet .................................. 212
10.2 Running BLAST locally ............................................. 214
    10.2.1 Introduction .............................................. 214
    10.2.2 Standalone NCBI BLAST+ .................................. 215
    10.2.3 Other versions of BLAST .................................. 215
10.3 Parsing BLAST output ............................................... 215
10.4 The BLAST record class ............................................ 217
10.5 Dealing with PSI-BLAST ............................................ 218
10.6 Dealing with RPS-BLAST ............................................ 218

### 11 BLAST and other sequence search tools

11.1 The SearchIO object model ........................................ 221
    11.1.1 QueryResult ............................................... 222
    11.1.2 Hit ....................................................... 227
    11.1.3 HSP ....................................................... 230
    11.1.4 HSPFragment ............................................... 233
11.2 A note about standards and conventions ............................ 234
11.3 Reading search output files ....................................... 235
11.4 Dealing with large search output files with indexing .............. 236
11.5 Writing and converting search output files ........................ 236
12 Accessing NCBI's Entrez databases

12.1 Entrez Guidelines .............................................................. 239
12.2 EInfo: Obtaining information about the Entrez databases ............... 240
12.3 ESearch: Searching the Entrez databases .................................. 242
12.4 EPost: Uploading a list of identifiers ...................................... 243
12.5 ESummary: Retrieving summaries from primary IDs ....................... 244
12.6 EFetch: Downloading full records from Entrez ........................... 244
12.7 ELink: Searching for related items in NCBI Entrez ....................... 247
12.8 EGQuery: Global Query - counts for search terms ....................... 248
12.9 ESspell: Obtaining spelling suggestions .................................. 249
12.10 Parsing huge Entrez XML files .......................................... 249
12.11 HTML escape characters ..................................................... 250
12.12 Handling errors .................................................................... 251
12.13 Specialized parsers ............................................................... 253
  12.13.1 Parsing Medline records ............................................... 253
  12.13.2 Parsing GEO records ..................................................... 255
  12.13.3 Parsing UniGene records ................................................. 256
12.14 Using a proxy ..................................................................... 258
12.15 Examples ............................................................................ 258
  12.15.1 PubMed and Medline ....................................................... 258
  12.15.2 Searching, downloading, and parsing Entrez Nucleotide records .... 260
  12.15.3 Searching, downloading, and parsing GenBank records ............. 261
  12.15.4 Finding the lineage of an organism .................................... 263
12.16 Using the history and WebEnv .............................................. 264
  12.16.1 Searching for and downloading sequences using the history ......... 264
  12.16.2 Searching for and downloading abstracts using the history .......... 265
  12.16.3 Searching for citations .................................................... 266

13 Swiss-Prot and ExPASy .................................................................. 267

13.1 Parsing Swiss-Prot files ......................................................... 267
  13.1.1 Parsing Swiss-Prot records ............................................... 267
  13.1.2 Parsing the Swiss-Prot keyword and category list .................... 269
13.2 Parsing Prosite records ............................................................ 270
13.3 Parsing Prosite documentation records ....................................... 271
13.4 Parsing Enzyme records .......................................................... 272
13.5 Accessing the ExPASy server ................................................... 273
  13.5.1 Retrieving a Swiss-Prot record .......................................... 273
  13.5.2 Searching Swiss-Prot ....................................................... 274
  13.5.3 Retrieving Prosite and Prosite documentation records ............... 274
13.6 Scanning the Prosite database .................................................. 275

14 Going 3D: The PDB module ......................................................... 277

14.1 Reading and writing crystal structure files .................................... 277
  14.1.1 Reading an mmCIF file .................................................... 277
  14.1.2 Reading files in the MMTF format .................................... 278
  14.1.3 Reading a PDB file .......................................................... 278
  14.1.4 Reading a PQR file .......................................................... 279
  14.1.5 Reading files in the PDB XML format .................................. 279
  14.1.6 Writing mmCIF files ........................................................ 279
  14.1.7 Writing PDB files ............................................................. 279
  14.1.8 Writing PQR files ............................................................. 280
  14.1.9 Writing MMTF files ........................................................ 280
14.2 Structure representation ........................................... 281
  14.2.1 Structure ......................................................... 283
  14.2.2 Model .......................................................... 283
  14.2.3 Chain .......................................................... 283
  14.2.4 Residue ......................................................... 284
  14.2.5 Atom ........................................................... 285
  14.2.6 Extracting a specific Atom/Residue/Chain/Model from a Structure ........................................... 286

14.3 Disorder ............................................................. 286
  14.3.1 General approach ............................................... 286
  14.3.2 Disordered atoms .............................................. 286
  14.3.3 Disordered residues ........................................... 287

14.4 Hetero residues .................................................... 287
  14.4.1 Associated problems .......................................... 287
  14.4.2 Water residues ............................................... 287
  14.4.3 Other hetero residues ........................................ 288

14.5 Navigating through a Structure object .............................. 288

14.6 Analyzing structures .............................................. 291
  14.6.1 Measuring distances ........................................... 291
  14.6.2 Measuring angles .............................................. 291
  14.6.3 Measuring torsion angles ...................................... 291
  14.6.4 Internal coordinates - distances, angles, torsion angles, distance plots, etc ........................................... 291
  14.6.5 Determining atom-atom contacts ................................ 299
  14.6.6 Superimposing two structures .................................. 300
  14.6.7 Mapping the residues of two related structures onto each other ........................................... 300
  14.6.8 Calculating the Half Sphere Exposure .......................... 300
  14.6.9 Determining the secondary structure .................................. 301
  14.6.10 Calculating the residue depth ................................... 301

14.7 Common problems in PDB files ..................................... 301
  14.7.1 Examples ......................................................... 302
  14.7.2 Automatic correction ........................................... 303
  14.7.3 Fatal errors ..................................................... 303

14.8 Accessing the Protein Data Bank ................................... 303
  14.8.1 Downloading structures from the Protein Data Bank ........... 303
  14.8.2 Downloading the entire PDB .................................... 304
  14.8.3 Keeping a local copy of the PDB up to date ...................... 304

14.9 General questions .................................................. 304
  14.9.1 How well tested is Bio.PDB? ..................................... 304
  14.9.2 How fast is it? .................................................. 304
  14.9.3 Is there support for molecular graphics? ........................ 305
  14.9.4 Who’s using Bio.PDB? ........................................... 305

15 Bio.PopGen: Population genetics ........................................ 306
  15.1 GenePop .......................................................... 306

16 Phylogenetics with Bio.Phylo ........................................ 308
  16.1 Demo: What’s in a Tree? ........................................... 308
    16.1.1 Coloring branches within a tree ................................ 309
  16.2 I/O functions ..................................................... 312
  16.3 View and export trees ............................................. 313
  16.4 Using Tree and Clade objects .................................... 314
    16.4.1 Search and traversal methods ................................ 315
    16.4.2 Information methods ........................................ 316
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.

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:

   ```python
   >>> 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.html)

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.

    Check the built in docstrings (from Bio import SeqIO, then help(SeqIO)), or see [http://biopython.org/wiki/SeqIO](http://biopython.org/wiki/SeqIO) and [http://biopython.org/wiki/AlignIO](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.

    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 cool 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.

```python
>>> from Bio import Seq
>>> seq = Seq('AGTACACTGGT')
```
followed by what you would type in:

```python
>>> 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:

```python
>>> 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
CGTAACAAGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGGAATAAACGATCGAGTG
AATCCGGAGGACCGGTGTACTCAGCTCACCGGGGGCATTGCTCCCGTGAGG
```

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:

```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('CGTAACAAGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGGAATAAACGATCGAGTG
AATCCGGAGGACCGGTGTACTCAGCTCACCGGGGGCATTGCTCCCGTGAGG...CGC')
740
```

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:

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

17
This should give:

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

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](http://biopython.org/wiki/SeqIO) and [http://biopython.org/wiki/AlignIO](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:

```python
>>> 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!):

```python
>>> 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:

```python
>>> 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:

```python
>>> from Bio.Seq import Seq
>>> my_seq = Seq("GATCGATGGGCTATAGGATCGAAAATCGC")
>>> 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:

```python
>>> from Bio.Seq import Seq
>>> from Bio.SeqUtils import gc_fraction
>>> my_seq = Seq("GATCGATGGGCTATAGGATCGAAAATCGC")
>>> 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:

```python
>>> from Bio.Seq import Seq
>>> my_seq = Seq("GATCGATGGGCTATAGGATCGAAAATCGC")
>>> 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('CGCTAAAAGCTAGATATATCCGGGTAGCTAG')

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
%s
' % 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("AAACCGG")
>>> 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('ATGNNNNNNNNNATCCCGNNNNNNNNNNTTGCA')

### 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("GATCGATGGGCTATAGGATCGAAAATCGC")

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

```python
>>> 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:

```python
>>> 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:

```
DNA coding strand (aka Crick strand, strand +1)
5’
ATGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG
3’

DNA template strand (aka Watson strand, strand −1)
3’
TACCGGTAACATTACCCGGCGACTTTCCCACGGGCTATC
5’
```

<table>
<thead>
<tr>
<th>Transcription</th>
</tr>
</thead>
</table>

5’ AUGGCCAUUGUAUUGGGCCCGCUUGAAAGGGUGCCCGAUAG 3’

Single stranded messenger RNA

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('CTATCGGGCCACCCTTACGCGGCCCATTACAATGGCCAT')

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 → 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 meth-
ods:

>>> 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"
... )

26

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

>>> gene.translate(table="Bacterial")
Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HR*',
ProteinAlphabet())

>>> 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


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

```python
>>> 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:

```python
>>> 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:

```python
>>> print(standard_table)
Table 1 Standard, SGCU

|   | 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 |
```

3.9 Translation Tables


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

```python
>>> 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:

```python
>>> 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:

```python
>>> print(standard_table)
Table 1 Standard, SGCU

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

```python
>>> 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.

```python
>>> 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:

```python
>>> 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:

```python
>>> 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  TTGAAAACCTGAATGTGAGAGTCCTCAAGGATAGT
```
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

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

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

Observe what happens if you try to edit the sequence:

```python
>>> 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:

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

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

```python
>>> 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:

```python
>>> 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:

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

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

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

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

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

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

```python
>>> 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:

```python
>>> 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 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:

```python
>>> 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)
'CTAAACCAGCAGCGACCCCGCCCTTCAACGACCCATAACAGC'
```
'GCTTTATGGGTCGTGGAAAGGTCGTCGCTGCTGGTATGGGTCGGTAAAGGGTGGTCGTGCTGCTGGTTAG'

>>> 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):

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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):

```plaintext
>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):

```python
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.fna", "fasta")
```

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:

```python
>>> record.seq
Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGGGGGTAATCTGCTCTCC')
```

Next, the identifiers and description:

```python
>>> 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
TGTAACGAACGGTGCAATAGTGATCCACACCCACACGGCTGAAATCAGATCCAGGGGGTAAATCTGCTCTCC
...

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

```python
>>> record.dbxrefs
[]
```

```python
>>> record.annotations
{}
```

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

```python
>>> 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:

<table>
<thead>
<tr>
<th>LOCUS</th>
<th>NC_005816</th>
<th>9609 bp</th>
<th>DNA</th>
<th>circular</th>
<th>BCT 21-JUL-2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINITION</td>
<td>Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCESSION</td>
<td>NC_005816</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VERSION</td>
<td>NC_005816.1</td>
<td>GI:45478711</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROJECT</td>
<td>GenomeProject:10638</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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):

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

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

```python
>>> record.seq
```

```
Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCACACGGCTGAAATCAGATCCAGGGGGTAAATCTGCTCTCC')
```

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 were 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:

```python
c from Bio import SeqFeature
c start_pos = SeqFeature.AfterPosition(5)
c end_pos = SeqFeature.BetweenPosition(9, left=8, right=9)
c 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:

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

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

```python
c my_location.start
```

`AfterPosition(5)`

```python
c print(my_location.start)
c >5
```

```python
c my_location.end
```

`BetweenPosition(9, left=8, right=9)`

```python
c print(my_location.end)
c (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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:
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:

```python
>>> 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):

```python
>>> 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.

```python
>>> 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:

```python
>>> 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?

```python
>>> 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!

```python
>>> 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:

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

which should give:
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:

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

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

print(record.features[21])
type: CDS
location: [4342:4780](+)
qualifiers:
  - Key: codon_start, Value: ['1']
  - Key: db_xref, Value: ['GI:45478716', 'GeneID:2767712']
  - Key: gene, Value: ['pim']
  - Key: locus_tag, Value: ['YP_pPCP05']
```
Key: note, Value: ['similar to many previously sequenced pesticin immunity protein entries of Yersinia pestis plasmid \textit{pPCP}, e.g. gi|16082683|, ref|NP\_395230.1| (NC\_003132), gi|12001066|emb|CA90861.1| (Z54145), gi|1488655|emb|CA63439.1| (X92856), gi|2996219|gb|AA626543.1| (AF053945), and gi|5763814|emb|CAB531 67.1| (AL109969)'

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

```python
>>> 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 \textit{YP\_pPCP05}:

```python
>>> 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']

>>> 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 \textit{pPCP}, e.g. gi|16082683|, ref|NP\_395230.1| (NC\_003132), gi|12001066|emb|CA90861.1| (Z54145), gi|1488655|emb|CA63439.1| (X92856), gi|2996219|gb|AA626543.1| (AF053945), and gi|5763814|emb|CAB531 67.1| (AL109969)']
```

45
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.

```python
>>> 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:

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

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

```python
>>> 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:

```python
>>> 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.
```

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:

```python
>>> from Bio import SeqIO
>>> record = next(SeqIO.parse("example.fastq", "fastq"))
>>> len(record)
25
```
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:

```python
>>> 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, 26, 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:

```python
>>> edited = left + right
>>> len(edited)
24
```

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

```python
>>> 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
```

```python
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:

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

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:
```python
>>> 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):

```
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):

```python
>>> 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:

```python
>>> 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:

```python
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(‘CATTGTGGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...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`.

```python
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:

```python
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:

```python
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
CGTAAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGGAATAAACGATCGAGTG
AATCCGGAGGACCGGTGTACTCAGCTCACCGGGGGCATTGCTCCCGTGGTGACCCTGATTTGTTGGG
...
```

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:

```python
>>> 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:

```python
>>> from Bio import SeqIO
>>> all_species = [
...     seq_record.description.split()[1]
...     for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta")
... ]
```

```python
>>> 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:

```python
>>> 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`:

```python
>>> 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
CGTAACAAGGTTTCCGGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGGAATAAA
CGATCGAGTGAATCCGGAGGACCGGTGTACTCAGCTCACCGGGGGCATTGCTCCCGTGTTGACCCTGATTTGTGCACCGGCGGCTGCTCCCGTGGT
```

### 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")))
>>> 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:

```python
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.

```python
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:

```python
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"):
```

57
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:

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

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

- Bio.SeqIO.to_dict() is the most flexible but also the most memory demanding option (see Section 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:

```python
>>> 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:

```python
>>> len(orchid_dict)
94
```
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.keys())
['Z78484.1', 'Z78464.1', 'Z78455.1', 'Z78442.1', 'Z78532.1', 'Z78453.1', ...

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

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

```python
>>> seq_record = orchid_dict['Z78475.1']
```

```console
P. supardi 5.8S rRNA gene and ITS1 and ITS2 DNA
```

```python
>>> print(seq_record.seq)
Seq('CGTAACAAGGTTCGTTCCGTAGGTGACCTGCGGAAGGATCATTGTTGAGATCAGATCACT...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:

```python
from Bio import SeqIO
orchid_dict = SeqIO.to_dict(SeqIO.parse("ls_orchid.fasta", "fasta"))
print(orchid_dict.keys())
```

This time the keys are:

```console
['gi|2765596|emb|Z78471.1|PDZ78471', 'gi|2765646|emb|Z78521.1|CCZ78521', ...
```

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):

```python
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]
```

60
Then we can give this function to the `SeqIO.to_dict()` function to use in building the dictionary:

```python
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:

```python
>>> 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:

```python
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:

```python
>>> 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:

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

>>> orchid_dict.keys()
['Z78484.1', 'Z78484.1', '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:

```python
>>> 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:

```python
def get_acc(identifier):
    """Given a SeqRecord identifier string, return the accession number as a string."
    e.g. "gi|2765613|emb|Z78458.1|PTZ78488" -> "Z78458.1"
    """
    parts = identifier.split("|")
    assert len(parts) == 5 and parts[0] == "gi" and parts[2] == "emb"
    return parts[3]
```

62
Then we can give this function to the Bio.SeqIO.index() function to use in building the dictionary:

```python
>>> 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 downloaded the whole of UniProt in the plain text SwissProt 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:

```python
>>> 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, gbvr11.seq, ..., gbvr138.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
$ 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.ncbi.nih.gov/genbank/gbvrl3.seq.gz
$ curl -O ftp.ncbi.nih.gov/genbank/gbvrl4.seq.gz
$ gunzip gbvrl*.seq.gz

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

```python
>>> 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.

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

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

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

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

or:

```python
>>> 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):

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

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

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

rec3 = SeqRecord(
```

66
```
Seq(
    "MVTVEEFRAQCAEPATVMAIGTATPSNCVDQSTYPDYYFRITNSEHKVELKEKFKRMC"
    "EKSMMKRYMLTEILKENPNICAYMAPSLDARQDIVDEPPKLGKEAAQKAIKEWQQP"
    "KSHTHLVCFTTSVDMPGCDQYLTLLGLRRPSVRFMYQQCGAFGTVLRAKDLAEN"
    "NKGARVLCSEITAVTRGPNDTHLDSLQVQALFDGDAAAVIGSDPTEVERPFLERV"
    "SAQQTLDPSEGAIDGRLEVGTLFHHKDPVGLISKNEKSLVEAFQPLGISDWNLSFW"
    "IAHPGPAILDQVELKLKQEKGLATRKVLNSNYGNSACCVLFILDEMRKASAKEGLGT"
    "TGEGLEWVLFGLFGPGLTVETVLHSVAT",
),

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]
MMYQQGCAGGTVLRKAKDLAENRRGARVLCSEITAVTRGSETHDMSVQALFGD
GAGAVVIGSDPOLSVERPVLWLVGTATLLPSEGAIDGRLEVGTLFHHKDPVGLISK
NIEKSLKEAFPLGIDSNSTFIVAHPPGPAILDQVEAKLGLKEKMRATREVLSEYGM
SSAC
>gi|13919613|gb|AAK33142.1| chalcone synthase [Fragaria vesca subsp. bracteata]
YPDYYFRITNREHKELKEFQRMCDKSMIKRYMLTEILKENPSCVDYMAPSLDARQ
DMVVCPEIPKLGEAAAVKAIKEWQG
>gi|13925890|gb|AAK49457.1| chalcone synthase [Nicotiana tabacum]
MVTVEEFRAQCAEPATVMAIGTATPSNCVDQSTYPDYYFRITNSEHKVELKEKFKRMC
EKSMMKRYMLTEILKENPNICAYMAPSLDARQDIVDEPPKLGKEAAQKAIKEWQQP
KSHTHLVCFTTSVDMPGCDQYLTLLGLRRPSVRFMYQQCGAFGTVLRAKDLAEN
NKGARVLCSEITAVTRGPNDTHLDSLQVQALFDGDAAAVIGSDPTEVERPFLERV
SAQQTLDPSEGAIDGRLEVGTLFHHKDPVGLISKNEKSLVEAFQPLGISDWNLSFW
IAHPGPAILDQVELKLKQEKGLATRKVLNSNYGNSACCVLFILDEMRKASAKEGLGT
TGEGLEWVLFGLFGPGLTVETVLHSVAT
```

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:

```
67
```
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:

```python
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:

```python
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:

```python
>>> 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):

```python
>>> 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:

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

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

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

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:

```python
>>> 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:

```python
>>> 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)
```
... )

```python
>>> 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:

```python
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:

```python
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:

```python
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):

```python
>>> 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:

```python
...     out_handle.write(">%s
%s
" % (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:

```python
>>> 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:

```python
...     out_handle.write("@%s
%s
+
%s
" % (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:

```python
>>> 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:

```python
>>> 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`;

72
• 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:

```python
>>> 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:

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

and an attribute coordinates with the alignment coordinates:

```python
>>> 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:

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

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

### 6.1.2 Creating an Alignment object from aligned sequences

If you start out with the aligned sequences, with dashes representing gaps, then you can calculate the coordinates using the 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:

```python
>>> aligned_sequences = ["CCGGTTTTT", "AG-TTT--", "AGGTTT--"]
>>> sequences = [aligned_sequence.replace("-", "")
... for aligned_sequence in aligned_sequences] # fmt: skip
... sequences
['CCGGTTTTT', '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
['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

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

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_annotatoins: 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:

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

To get the aligned sequence strings individually, use

```python
>>> 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:
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

```

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
```

The column index can also be an iterable of integers:
6.3 Getting information about the alignment

6.3.1 Alignment shape

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

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
(((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,

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

```python
gap futile generosity
```

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

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

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:
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.

```python
>>> 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)
```

```python
>>> 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:

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

```python
>>> m = alignment.substitutions
>>> print(m)
A  C  G  T
A 1.0 0.0 0.0 0.0
C 2.0 0.0 0.0 0.0
G 0.0 0.0 4.0 0.0
T 0.0 0.0 0.0 9.0
```
Note that the matrix is not symmetric: The counts for a row letter R and a column letter C is the number of times letter R in a sequence is replaced by letter C in a sequence appearing below it. For example, the number of C’s that are aligned to an A in a later sequence is

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

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

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

To get a symmetric matrix, use

```python
>>> 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
```

```python
>>> 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.

```python
>>> align_array = np.array(alignment)
>>> align_array.shape
(3, 8)
```

```python
>>> 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'`:

```python
>>> 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.

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

>>> alignment.sort()

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

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

```python
>>> from Bio.SeqUtils import gc_fraction

>>> alignment.sort(key=gc_fraction)

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

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

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

```python
>>> alignment.sort(key=gc_fraction, reverse=True)

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

6.4.2 Reverse-complementing the alignment

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

```python
>>> 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
```
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:

```python
>>> 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:

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

>>> print(right)
```

Reverse-complementing an alignment preserves its column annotations (in reverse order), but discards all other annotations.
Adding the two alignments will combine the two alignments row-wise:

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

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

```python
>>> 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:

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

Similarly any common per-column-annotations are combined:

```python
>>> 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:

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

and a pairwise alignment between the transcript and a sequence (e.g., obtained by RNA-seq):
>>> rnaseq = "CCCCGGGG"
>>> sequences2 = [transcript, rnaseq]
>>> coordinates2 = np.array([[3, 11], [0, 8]])
>>> alignment2 = Alignment(sequences2, coordinates2)
>>> print(alignment2)
target  3  CCCCCGGG  11
   0  |||||    ||||  8
query   0  CCCCCGGG  8

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

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

>>> alignment3.coordinates
array([[11, 15, 26, 30],
       [ 0, 4, 4, 8]])

>>> format(alignment3, "psl")
'8	0	0	0	0	0	1	11	+	query	8	0	8	target	40	11	30	2	4,4,	0,4,	11,26,

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

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]

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

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:

```python
>>> 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
dm6.chr1 249250621
rheMac8.chr1 225584828
calJac3.chr18 47448759
mm10.chr3 160039680
rn6.chr2 266435125
```

```python
>>> 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, 188162957,
        188162953, 188162930, 188162930, 188162918]])
```

```python
>>> print(genome_alignment)
panTro5.c 133922962 ---ACTAGTTA--CA----GTAACAGAAAATAAAATTTAAATAGAAACTTAAAggcc
dm6.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 TATAATAATTGTATATGTCACAGAAAAAAATGAATTTTCAAT---AACTTAATAGCC
<BLANKLINE>
```

Suppose we want to replace the older versions of the genome assemblies (panTro5, dm6, rheMac8, calJac3, mm10, and rn6) by their current versions (panTro6, dm8, 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:

```python
>>> 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 133929497 133929639 1339296312 133932620]
chr1 [155184381 156354347 156354348 157128497 157128497 157137496]
chr1 [130382477 130383872 130383872 130384222 130384222 130388520]
chr18 [9786631 9788509 9788508 97955065 97959062 97959062 97959737]
chr3 [66807541 74196580 74196581 94705728 94705728 94708176 94708176 94708176 94708187 94708187 94708187]
chr2 [188111581 188158351 188158351 188171225 188171225 188228261 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:

```python
>>> 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, 95186245, 95186243, 95186243, 95186220, 95186217, 95186205],
[ 9758318, 9758318, 9758318, 9758326, 9758326, 9758326, 9758326, 9758326, 9758326],
[ 9758351, 9758354, 9758366],
```

87
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:

```python
>>> 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}"
...
```

```text
panTro6.c 130611000 ---ACTAGTTA--CA----GTAACAGAAAATTTAAATAGAAACTTAAAggcc
hg38.chr1 155814782 ---ACTAGTTA--CA----GTAACAGAAAATTTAAATAGAAACTTAAAggcc
rheMac10. 95186253 ---ACTAGTTA--CA----GTAACAGAAAATTTAAATAGAAACTTAAAggcc
calJac4.c 9758318 ---ACTAGTTA--CA----GTAACAGAAAATTTAAATAGAAACTTAAAggcc
mm39.chr3 88765346 TATAATAATTGTATATGTCACAGAAAAATGAATTTTCAAT---GACTTAATAGCC
rn7.chr2 174256702 TACAATAATTG--TATGTCATAGCAGAAAAATGAATTTTCAAT---GACTTAATAGCC
```

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).

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

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

```python
>>> 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), allowing you to store a metadata dictionary as an attribute on the alignments object:

```python
>>> 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:

```python
>>> str(alignment)
' 1 CGGTTTTT 9
 0 AGGTTT-- 6
 0 AG-TTT-- 5'

>>> format(alignment)
' 1 CGGTTTTT 9
 0 AGGTTT-- 6
 0 AG-TTT-- 5'

>>> print(format(alignment))
1 CGGTTTTT 9
0 AGGTTT-- 6
By specifying one of the formats shown in Section 6.7, format will create a string showing the alignment in the requested format:

```python
>>> format(alignment, "clustal")
'sequence_0
CGGTTTTT
sequence_1
AGGTTT--
sequence_2
AG-TTT--'
```

```python
>>> print(format(alignment, "clustal"))
sequence_0 CGGTTTTT
sequence_1 AGGTTT--
sequence_2 AG-TTT--
```

```python
>>> print(f"*** this is the alignment in Clustal format: ***
{alignment:clustal}
***")
*** this is the alignment in Clustal format: ***
sequence_0 CGGTTTTT
sequence_1 AGGTTT--
sequence_2 AG-TTT--
```

```python
>>> 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--'
```

```python
>>> print(format(alignment, "maf"))
a
s sequence_0 1 8 + 9 CGGTTTTT
s sequence_1 0 6 + 6 AGGTTT--

s sequence_2 0 5 + 7 AG-TTT--
```

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

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

```python
>>> print(pairwise_alignment.format("bed"))
# doctest: +NORMALIZE_WHITESPACE
target 1 7 query 0 + 1 7 0 2 2,3,
```

```python
>>> print(pairwise_alignment.format("bed", bedN=3))
# doctest: +NORMALIZE_WHITESPACE
target 1 7
```
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.

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

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:

```plaintext
>plas_horvu
D-VLGGANGVLVFEPNDFSVKAGITITIFKNACGYPHNVFVDEDAPSG-VDF-VSKISQEEYLTAPGETFSVTLTV---PGTYGYCPEPHAGCMVGKTVV
>plas_chlre
--VHLGADSGALEFVPTLTIKSGTVNLFVNNAGFPHNIVFVEDAIPOSG-VN-ADAISRDDYNAPGETYSVKLTA---AGEYGYCPEPHQAGCMVGKIIV
>plas_anava
--VHLGSDGQLLVFEPAKLTIKPGTVEFLNKKVPKNFVVDAPSAADLAKLSHKLMSMQSTSSTPAPAAGETFVGYCEPHRAGCMTGKTV
>plas_proho
VQIKMGTDKYAPLYEPKLSISAGDTEVFVMNKGPHNVIFDK--VPAG-ES-APALSNTKLRIAPGSFYETLTV---PGTYSFYCTPHRAGCMVGKTVV
```
To read this file, use

```python
from Bio import Align
alignment = Align.read("probcons.fa", "fasta")
```

We can print the alignment to see its default representation:

```python
print(alignment)
```

or we can print it in the aligned FASTA format:

```python
print(format(alignment, "fasta"))
```

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

```python
print(format(alignment, "clustal"))
```
The sequences associated with the alignment are `SeqRecord` objects:

```python
>>> alignment.sequences
[SeqRecord(seq=Seq('DVLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSGVVDVSKI...VT'), id='plas_horvu', name='<unknown name>', description='', dbxrefs=[]),
 SeqRecord(seq=Seq('---VKLGADSGALEFVPKTLTIKSGETVNFVNAGFPHNIVFDEDAIPSG---ADAISRDDYLNPAGTVKLT---AGEEGYCEPHQGAGMVIIV'), id='plas_chlre', name='<unknown name>', description='', dbxrefs=[]),
 SeqRecord(seq=Seq('---VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDEAALNPASADLAKSLSHKQLMSPQSTSTTFPAPAGTYCEFYCEPHRGAGMVGKITV'), id='plas_anava', name='<unknown name>', description='', dbxrefs=[]),
 SeqRecord(seq=Seq('VQIGMTDKYAPLYEPKALSISAGDTVEFMNKVPPHNVFDEPHSKLMPAQVHMTYPPG--APALSNTKRIAPGSFGSTVLT---PTGSYCTPHRAGMVGKITV'), id='plas_proho', name='<unknown name>', description='', dbxrefs=[]),
 SeqRecord(seq=Seq('VHMLNKGDGMVFEPASLKVAPGDTVITFIPTDK---GNHVEITKGMIPDG--AE-A-------FKSKINENYKVFTFA---PGVYGVKCTPHYGMGVMGVEV'), id='azup_achcy', name='<unknown name>', description='', dbxrefs=[])]
```

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

```python
>>> 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):

```python
>>> from io import StringIO
>>> stream = StringIO()
>>> Align.write(alignment, stream, "FASTA")
1
>>> print(stream.getvalue())
>plas_horvu
D-VLLGANGGVLVFEPNDFSVKAGETITFKNNAGYPHNVVFDEDAVPSGVVDVSKI...VT
>plas_chlre
---VKLGADSGALEFVPKTLTIKSGETVNFVNAGFPHNIVFDEDAIPSG---ADAISRDDYLNPAGTVKLT---AGEEGYCEPHQGAGMVIIV
>plas_anava
---VKLGSDKGLLVFEPAKLTIKPGDTVEFLNNKVPPHNVVFDEAALNPASADLAKSLSHKQLMSPQSTSTTFPAPAGTYCEFYCEPHRGAGMVGKITV
>plas_proho
VQIGMTDKYAPLYEPKALSISAGDTVEFMNKVPPHNVFDEPHSKLMPAQVHMTYPPG--APALSNTKRIAPGSFGSTVLT---PTGSYCTPHRAGMVGKITV
>azup_achcy
VHMLNKGDGMVFEPASLKVAPGDTVITFIPTDK---GNHVEITKGMIPDG--AE-A-------FKSKINENYKVFTFA---PGVYGVKCTPHYGMGVMGVEV
```

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
```
To parse this file, use

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

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

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

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

```python
>>> alignment = next(alignments)
>>> print(alignment)
```

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:

```python
>>> 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:

```python
>>> alignment.column_annotations
{'clustal_consensus': '******* **** **********************************...
```

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

```python
>>> print(format(alignment, "clustal"))
```

95
Writing the alignments in clustal format will include both the metadata and the sequence alignment:

```python
>>> 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>
gi|6273285|gb|AF191659.1|AF191 TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAA

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 archaeabacterium Sulfolobus acidocaldarius.
#=GF RA Choli T, Wittmann-Liebold B, Reinhardt R;
#=GF DR INTERPRO; IPR003212;
#=GF DR SCOP; 1soo; fa;
#=GF DR SO; 0000417; polypeptide_domain;
#=GF CC This family contains members of the hyper-thermophilic
#=GF CC archaeabacterium 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
```
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:

```python
>>> 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:

```python
>>> print(alignment)
DN7_METS5 0 KIKFKYKGQDELEISKVKVWKVQKMVSFTYDD-NGKTGRGAVSEKDAPKELLNMIGK
DN7A_SACS2 0 TVKFKYKGEEKQVDISKIKKWWVRQVXXMFSFYDD-NGKTVGRGSAKEDAPKELLMQLEK
DN7E_SULAC 0 KVRFKYGKEEKVDISKIKKWWVRQVXXMFSFYDD-NGKTVGRGSAKEDAPKELLMQLEK
```

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:

```python
>>> 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
```
Microsequence analysis of DNA-binding proteins 7a, 7b, and 7e from the archaebacterium Sulfolobus acidocaldarius.

This family contains members of the hyper-thermophilic archaebacterium 7kD DNA-binding/endoribonuclease P2 family. There are five 7kD DNA-binding proteins, 7a-7e, found as monomers in the cell. Protein 7e shows the tightest DNA-binding ability.

or alternatively as aligned FASTA (see section 6.7.1):

```python
>>> print(format(alignment, "fasta"))
>DN7_METS5/4-61
KIKFKYKGQDLEVISKVKWVWGMVSFTYDD-NGTKGRGAVSEKDAPKELLNMIGK
>DN7A_SACS2/3-61
TVKFKYKGEKQVDISKIKKVRWVGMSFTYDEGGKTGRGAVSEKDAPKELLQMLEK
>DN7A_SACS2/3-61 SS
EEEEEEEEEEEEEEEEEEEEEEEEEEEEE----CHHHHHTT
>DN7E_SULAC/3-60
KVRFKYKGEKEVDSIKIKVWVRGMFSFTYDD-NGTKGRGAVSEKDAPKELMDMLAR
>GC seq_cons
KVRFKYKGEKEVDSIKIKVWVRGMFSFTYDD-NGTKGRGAVSEKDAPKELMDMLAR
<BLANKLINE>
```

or in the PHYLIP format (see section 6.7.4):

```python
>>> print(format(alignment, "phylip"))
3 59
DN7_METS5/KIKFKYKGQDLEVISKVKWVWGMVSFTYDD-NGTKGRGAVSEKDAPKELLNMIGK
DN7A_SACS2/3-61 TVKFKYKGEKQVDISKIKKVRWVGMSFTYDEGGKTGRGAVSEKDAPKELLQMLEK
DN7E_SULAC/3-60 KVRFKYKGEKEVDSIKIKVWVRGMFSFTYDD-NGTKGRGAVSEKDAPKELMDMLAR
<BLANKLINE>
```

General information of the alignment is stored under the `annotations` attribute of the `Alignment` object, for example
The individual sequences in this alignment are stored under \texttt{alignment.sequences} as \texttt{SeqRecord}s, including any annotations associated with each sequence record:

```python
>>> for record in alignment.sequences:
... print("%s %s %s" % (record.id, record.annotations["accession"], record.dbxrefs))
```

DN7\_METS5/4-61 A4YEA2.1 []
DN7A\_SACS2/3-61 P61991.2 ['PDB; 1SSO A; 2-60;', 'PDB; 1JIC A; 2-60;', 'PDB; 2CVR A; 2-60;', 'PDB; 1B4O A; 2-60;']
DN7E\_SULAC/3-60 P13125.2 []

The secondary structure of the second sequence (DN7A\_SACS2/3-61) is stored in the \texttt{letter\_annotations} attribute of the \texttt{SeqRecord}:

```python
>>> alignment.sequences[1].letter_annotations
{'secondary structure': 'EEEEESSSSEEEEEETTTS-CHHHHHHTT'}
```

The consensus sequence and secondary structure are associated with the sequence alignment as a whole, and are therefore stored in the \texttt{column\_annotations} attribute of the \texttt{Alignment} object:

```python
>>> alignment.column_annotations  # doctest: +NORMALIZE_WHITESPACE
{'consensus secondary structure': 'EEEEESSSSEEEEEETTTS-CHHHHHHTT', 'consensus sequence': 'KVKFKYKGEKEKEVKWVGRVMVSFTYDD.NGKTQRYGAVSEKDAPKELuK'}
```

### 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):
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 LFVLAVTFVF VSS------- --------RG IPPEEQ------- -------SQ
ALLEU_HORVU MAHARVLLA LAVLATAAVA VASSSSFADS NPIRPVTDRA ASTLESAVLG ALGRTRHALR
CATH_HUMAN ------MWAT LPLLCAGAWL LGV------- -PVCGAAELS VNSLEK------ -------FH
```

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.

```python
>>> from Bio import Align
>>> alignment = Align.read("sequential.phy", "phylip")
>>> alignment # doctest: +ELLIPSIS
<Alignment object (3 rows x 384 columns) at ...>
```
Here, two alignments are considered to be equal if they have the same sequence contents and the same alignment coordinates.

```python
gena.run_tests(alignments, equality_checker=alignments, alignments2)
```

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

```python
>>> print(format(alignment, "phylip"))
```

We can write the alignment in PHYLIP format, parse the result, and confirm it is the same as the original alignment object:
>>> 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 TSPASIRPAGPSSRPAMVSSRTRPSPPGPRPTGRCCSAAPRPQAT 50
IXI_235 1 TSPASIRPAGPSSRRPSPPGPRPTGRCCSAAPRPQAT 41
IXI_234 51 GGWKTCSGTCTTSTSTRHRGSGSARTTTAAACLRAASRKSMRAACRSAG 100
IXI_235 42 GGWKTCSGTCTTSTSTRHRGSGW----------RASRKSMRAACRSAG 81
IXI_234 101 SRPNRFAPTLMSSCITSTTGPWAPAGDRSHE 131
```
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:

```python
>>> 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):

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

To pull out the alignment, we use

```python
>>> alignment = next(alignments)
>>> alignment
# doctest: +ELLIPSIS
<Alignment object (2 rows x 131 columns) at ...>
>>> alignment.shape
(2, 131)
>>> print(alignment)
IXI_234 0 TSPASIRPPAGPSSRPAMVSSRRTRPSPPGPRRPTGRPCSAAPRPQPATGWWKTSQGTC
0 |||||||||------|------------------|-----------------------------------------|
IXI_235 0 TSPASIRPPAGPSSR-----------RPSPGPRRPTGRPCSAAPRPQPATGWWKTSQGTC
<BLANKLINE>
IXI_234 60 TTSTSTRHRGRSGWSARTTTAACLRSKSMRAACSRSAGRSRPNRFAPTLMSSCITSTTG
60 |---------------------|-----------------------------------------|
IXI_235 51 TTSTSTRHRGRSGW-----------RASRSKRMRAACSRSAGRSRPNRFAPTLMSSCITSTTG
<BLANKLINE>
IXI_234 120 PPAWAGDRSHE 131
120 |---------------------| 131
IXI_235 101 PPAWAGDRSHE 112
<BLANKLINE>
```

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

```python
>>> alignment[0]
'TSPASIRPPAGPSSRPAMVSSRRTRPSPPGPRRPTGRPCSAAPRPQPATGWWKTSQGTC
0 |---------------------|-----------------------------------------|
>>> alignment[1]
'TSPASIRPPAGPSSR-----------RPSPGPRRPTGRPCSAAPRPQPATGWWKTSQGTC
<BLANKLINE>
>>> alignment[1, 10:30]
'GPSSR-----------RPSPGP'  
```

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

```python
>>> 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:

```python
>>> alignment[0]
'TSPASIRPPAGPSSRPAMVSSRRTRPSPPGPRRPTGRPCSAAPRPQPATGWWKTSQGTC
0 |---------------------|-----------------------------------------|
>>> alignment[1]
'TSPASIRPPAGPSSR-----------RPSPGPRRPTGRPCSAAPRPQPATGWWKTSQGTC
<BLANKLINE>
>>> alignment[1, 10:30]
'GPSSR-----------RPSPGP'  
```
>>> alignment.annotations
{'Matrix': 'EBLOSUM62', 'Gap_penalty': 10.0, 'Extend_penalty': 0.5, 'Identity': 112, 'Similarity': 112, 'Gaps': 19, 'Score': 591.5}

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

```python
>>> 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:

```python
>>> 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):

```python
>>> print(format(alignment, "phylip"))
2 131
IXI_234 TSPASIRPPAGPSSRPMVSRRTRFRSPGPRRPTGRCPSSAAPRRPQPATGKWKCTCGTCTTSTSTRHRSGWARTTTAACLRASRKSMRAACSRSAGSRPNRFAPTLMSSCITSTTGPPAWAGDRSHE
IXI_235 TSPASIRPPAGPSSR---------RPSPPGPRRPTGRCPSSAAPRRPQPATGKWKCTCGTCTTSTSTRHRSGW---------RASRKSMRAACSRSAGSRPNRFAPTLMSSCITSTTGPPAWAGDRSHE
<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:

```python
>>> alignment.target
SeqRecord(seq=Seq('TSPASIRPPAGPSSRPMVSRRTRFRSPGPRRPTGRCPSSAAPRRPQPATGKWKCTCGTCTTSTSTRHRSGWARTTTAACLRASRKSMRAACSRSAGSRPNRFAPTLMSSCITSTTGPPAWAGDRSHE'), id='IXI_234', name='IXI_234', description='', dbxrefs=[])

>>> alignment.query
SeqRecord(seq=Seq('TSPASIRPPAGPSSR---------RPSPPGPRRPTGRCPSSAAPRRPQPATGKWKCTCGTCTTSTSTRHRSGW---------RASRKSMRAACSRSAGSRPNRFAPTLMSSCITSTTGPPAWAGDRSHE'), id='IXI_235', name='IXI_235', description='', dbxrefs=[])
```

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

### 6.7.6 GCG Multiple Sequence Format (MSF)

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

```text
!!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
```
To parse this file with Biopython, use

```python
>>> 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:

```python
>>> alignment  # doctest: +ELLIPSIS
<Alignment object (11 rows x 99 columns) at ...>
```

```python
>>> print(alignment)
```

To parse this file with Biopython, use

```python
>>> 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:

```python
>>> alignment  # doctest: +ELLIPSIS
<Alignment object (11 rows x 99 columns) at ...>
```

```python
>>> print(alignment)
```

To parse this file with Biopython, use

```python
>>> 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:

```python
>>> alignment  # doctest: +ELLIPSIS
<Alignment object (11 rows x 99 columns) at ...>
```

```python
>>> print(alignment)
```

To parse this file with Biopython, use

```python
>>> 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:

```python
>>> alignment  # doctest: +ELLIPSIS
<Alignment object (11 rows x 99 columns) at ...>
```

```python
>>> print(alignment)
```
The sequences and their names are stored in the `alignment.sequences` attribute:

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

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

```python
>>> alignment.coordinates
array([[ 0,  93,  99],
        [ 0,  93,  99],
        [ 0,  93,  99],
        [ 0,  93,  99],
        [ 0,  93,  99],
        [ 0,  93,  99],
        [ 0,  93,  99],
        [ 0,  93,  99],
        [ 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/scer_s288c.fa --bestn 3 --showalignment no --showcigar no --showvulgar yes]
Hostname: [blackbriar]
vulgar: gi|296143771|ref|NM_001180731.1| 0 1230 + gi|330443520|ref|NC_001136.10| 1319275 1318045 - 6146 M 1 1 C 3 3 M 1226 1226
vulgar: gi|296143771|ref|NM_001180731.1| 1230 0 - gi|330443520|ref|NC_001136.10| 1318045 1319275 + 6146 M 129 129 C 3 3 M 1098 1098
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 ... M 12 12 G 0 2 M 5 5 G 1 0 M 7 7 G 0 2 M 1 1 G 0 1 M 12 12 C 75 75 M 6 6 G 1 0 M 4 4 G 0 1 M 2 2 G 0 1 M 3 3 G 0 1 M 41 41 — completed exonerate analysis
```

106
This file includes three alignments. To parse this file, use

```python
>>> 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:

```python
>>> alignments.metadata  # doctest: +NORMALIZE_WHITESPACE
{'Program': 'exonerate',
 'Command line': 'exonerate -m cdna2genome ../scer_cad1.fa /media/Waterloo/Downloads/genomes/scer_s288c/scer_s288c.fa --bestn 3 --showalignment no --showcigar no --showvulgar yes',
 'Hostname': 'blackbriar'}
```

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

```python
>>> 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:

```python
>>> print(format(alignment, "exonerate"))
vulgar: gi|296143771|ref|NM_001180731.1| 0 1230 + gi|330443520|ref|NC_001136.10| 1319275 1318045 - 6146 M 1 1 C 3 3 M 1226 1226
<BLANKLINE>
```

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

```python
>>> print(alignment.format("exonerate", "vulgar"))
vulgar: gi|296143771|ref|NM_001180731.1| 0 1230 + gi|330443520|ref|NC_001136.10| 1319275 1318045 - 6146 M 1 M 3 M 1226
<BLANKLINE>
```

```python
>>> print(alignment.format("exonerate", "cigar"))
cigar: gi|296143771|ref|NM_001180731.1| 0 1230 + gi|330443520|ref|NC_001136.10| 1319275 1318045 - 6146 M 1 M 3 M 1226
<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:
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:

```python
>>> print(format(alignment, "bed"))  # doctest: +NORMALIZE_WHITESPACE
gi|330443520|ref|NC_001136.10| 1318045 1319275 gi|296143771|ref|NM_001180731.1| 6146 - 1318045 1319275 0 3 1226,3,1, 0,1226,1229,
```

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:

```python
>>> del alignment.operations
>>> print(format(alignment, "sam"))  # doctest: +NORMALIZE_WHITESPACE
gi|296143771|ref|NM_001180731.1| 16 gi|330443520|ref|NC_001136.10| 1318046 255 1226M3M1M * 0 0 * * AS:i:6146
```

The third alignment contains four long gaps:

```python
>>> alignment = next(alignments)  # second alignment
>>> print(alignment)  # doctest: +ELLIPSIS
```

```python
>>> print(format(alignment, "exonerate"))  # doctest: +NORMALIZE_WHITESPACE
```

---

108
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   AAAAAAGCATTGTGGTGGGAAT [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

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

<BLANKLINE>
The sequences are stored under the `sequences` attribute:

```python
>>> 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

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

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

### 6.7.9 Tabular output from BLAST or FASTA

Alignment output in tabular output is generated by the FASTA aligner [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:

```bash
# fasta36 -m 8CC seq/mgstm1.nt seq/gst.nlib
# FASTA 36.3.8h May, 2020
# Query: pGT875 - 657 nt
# Database: seq/gst.nlib
# Fields: query id, subject id, % identity, alignment length, mismatches, gap opens, q. start, q. end, s. start, s. end, evalue, bit score, aln_code
# 12 hits found
pGT875    pGT875    100.00    657    0    0    1    657    38    697
pGT875    RABGLTR    79.10    646    135    0    1    646    34    646
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

```python
>>> 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`:

```python
>>> 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:

```python
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> print(alignment)
```

```
RABGSTB 156 ?????????????????????????????????--??????????????????????????????
0 |||||||||||||--|||||||||||--|||||||||||-|||||||||||
pGT875 158 ???????????????--??????????????????????????????-????????????

RABGSTB 214 ??????????????????????????????????????????????????????????-?
60 ||||||||||||||||||||||||||||||||||||||||||-|||||||||||-|
pGT875 215 ???????????????????????????????????????????????????????-??????????

>>> alignment.coordinates  # doctest: +NORMALIZE_WHITESPACE
array([[156, 171, 190, 190, 201, 202, 260, 261, 272, 272, 279, 279, 287],
       [158, 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`):
Information of the alignment is stored under the `annotations` attribute of the `alignment`:

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

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

### 6.7.10 HH-suite output files

Alignment files in the `hhr` format are generated by `hhsearch` or `hhblits` in HH-suite [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 1
>2uvo_A Agglutinin isolectin 1; carbohydrate-binding protein, hevein domain, chitin-binding, GERM agglutinin, PDB: 1wgc_A* 2cwg_A* 2x3t_A* 4aml_A* 7wga_A 9wga_A 2wgc_A 1wgt_A 1k7t_A* 1k7v_A* 1k7u_A 2x52_A* 1t0w_A*
Probab=99.95 E-value=3.7e-34 Score=210.31 Aligned_cols=171 Identities=100% Similarity=2.050 Sum_probs=166.9
```

```
Q 2UVO:A|PDBID|C
Q Consensus
T Consensus
T 2uvo_A
T ss_dssp
T ss_pred
Confidence
```

112
The file contains three sections:

- A header with general information about the alignments;

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

To parse this file, use

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

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

```python
>>> 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:

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

as it will reappear in each of the alignments.

Iterate over the alignments:

```python
>>> 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:

```python
>>> alignments.rewind()
>>> alignment = next(alignments)
>>> alignment
# doctest: +ELLIPSIS
<Alignment object (2 rows x 171 columns) at ...>
```

```python
>>> print(alignment)
2uvo_A 0 ERCGEQGSGNEMCPCNLCCSQQYGCMMGDYCGKGCQNGACWTSKRCGSGAGGATCTNNQC
0 ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
2UVO:A|PD 0 ERCGEQGSGNEMCPCNLCCSQQYGCMMGDYCGKGCQNGACWTSKRCGSGAGGATCTNNQC
<BLANKLINE>
2uvo_A 60 CSQYGYCGFGEYCGACQGGPGRADIKCGSGAGGKLCPNLCCSQWGFCGGLSEFGG
60 ||||||||||||||||||||||||||||||||||||||||||||||||||||
2UVO:A|PD 60 CSQYGYCGFGEYCGACQGGPGRADIKCGSGAGGKLCPNLCCSQWGFCGGLSEFGG
<BLANKLINE>
2uvo_A 120 CQSGACSTDKPCGKDAGGRVCTNNYCCSKWCGIGPGYCGACQSGGC
120 |||||||||||||||||||||||||||||||||||||||||||||||||||||||
2UVO:A|PD 120 CQSGACSTDKPCGKDAGGRVCTNNYCCSKWCGIGPGYCGACQSGGC
<BLANKLINE>
```

114
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`:

```python
>>> 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 `hhb` file is abbreviated):

```python
>>> 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):

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

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

```python
>>> 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:

```python
>>> alignment.target.annotations
# doctest: +NORMALIZE_WHITESPACE
{'hmm_name': '2uvo_A',
 'hmm_description': 'Agglutinin isolectin 1; carbohydrate-binding protein, hevein domain, chitin-binding...aestivum} PDB: 1wgc_A* 2cwg_A* 2x3t_A* 4aml_A* 7wga_A 9wga_A 2wgc_A 1wgt_A 1k7t_A* 1k7v_A* 1k7u_A 2x52_A* 1t0w_A*'}
```

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

```python
>>> alignment.target.letter_annotations
# doctest: +NORMALIZE_WHITESPACE
{'Consensus': '~~cg~~~~~~~c~~~~CCs~~g~Cg~~~~~c~~~c~~~~c~~~~~Cg~~~~~~~c~~~~CCs~~g~Cg~~~~~c~~~c~~~~~~~~~~Cg~~~~~~~c~~~~CCS~~G~CG~~~~~C~~~Cq~~~c~~~~~Cg~~~~~~~c~~~~CCS~~G~CG~~~~~C~~gCq~~~c~~',
 'ss_pred': 'CCCCCCCCCcCCCCCCeeCCCCeECCCCccccccccccccccccccccccccccCCcccCCcccCCcccCCcccCCcccCCcccCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```
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:

```python
>>> alignment.column_annotations  # doctest: +NORMALIZE_WHITESPACE
{'column score': '||||+++.++..||++.|||+|+|||.+.+||+++||.+.|++..+|+++++.++|....|||.++||+.+.+||+.+||.+++++|+.|+...+++.||++.|||.|||||...+||+.+||+++|++|.+|++.+++++|..+.|||+++-||+...||+++||++.|||',
'Confidence': '799999999999999999999999999999999999999999999999999999999999999999999999999999999999999999997999999999999999999999999999999999999999999999999999999999999999999999999999986'}
```

### 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):

```plaintext
>plas_horvu
D.VLLGANGGVLFEPNDFSVKAGETITFKNAGYPHNVVFDDAVPSG.VD.VSKISQEEYLTAPGETFSVTLLTV...PGTYGYPECHAGAMGVKTV
>plas_chlre
-.VKLGADSGALEFVKPTLTIKSGETTVNFVNAGFPHNIVFDDAIPSG.VN.ADAISRDDYNAPGETYSVKLT...AGEYGYCEPHQAGMVKII
>plas_anava
-.VKLGSKGLLFEPKPTLTIKSGEVTNFVNAGFPHNIVFDDAIPSG.VN.ADAISRDDYNAPGETYSVKLT...AGEYGYCEPHQAGMVKII
>plas_proho
VqIKMGTDKYAPLYEPKALSIGATDVEFVMNKVGPHNVIFDK--VPAG.ES.APALSNTKRLIAIPGFSYVSTLG...PGTYSFYCTPHRGAMGVKIT
>azup_achcy
VhMLNKGDKGMFEPALKVAPGVDTIFTIDK--GNVETIKGMIPDG.AE.A-------FKSKINEYKVFT...PGVYGVKCTPHGCMVGVVE
```

To parse this alignment, use

```python
>>> 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-VLLGANGGVLFEPNDFSVKAGETITFKNAGYPHNVVFDDAVPSG-VD-VSKISQE
plas_chlr 0 --VKLGADSGALEFVKPTLTIKSGEVTNFVNAGFPHNIVFDDAIPSG-VN-ADAISRD
plas_anava 0 --VKLGSKGLLFEPKPTLTIKSGEVTNFVNAGFPHNIVFDDAIPSG-VN-ADAISRD
plas_proho 0 VqIKMGTDKYAPLYEPKALSIGATDVEFVMNKVGPHNVIFDK--VPAG.ES-APALSNT
azup_achcy 0 VhMLNKGDKGMFEPALKVAPGVDTIFTIDK-GHNVETIKGMIPDG.AE-A-------FKSKINEYKVFT...PGVYGVKCTPHGCMVGVVE
```

To parse this alignment, use

```python
>>> 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-VLLGANGGVLFEPNDFSVKAGETITFKNAGYPHNVVFDDAVPSG-VD-VSKISQE
plas_chlr 0 --VKLGADSGALEFVKPTLTIKSGEVTNFVNAGFPHNIVFDDAIPSG-VN-ADAISRD
plas_anava 0 --VKLGSKGLLFEPKPTLTIKSGEVTNFVNAGFPHNIVFDDAIPSG-VN-ADAISRD
plas_proho 0 VqIKMGTDKYAPLYEPKALSIGATDVEFVMNKVGPHNVIFDK--VPAG.ES-APALSNT
azup_achcy 0 VhMLNKGDKGMFEPALKVAPGVDTIFTIDK-GHNVETIKGMIPDG.AE-A-------FKSKINEYKVFT...PGVYGVKCTPHGCMVGVVE
```

116
The parser analyzes the pattern of dashes, periods, and lower and upper case letters in the A2M file to determine if a column is a match/mismatch/deletion ("D") or an insertion ("I"). This information is stored under the match key of the alignment.column_annotations dictionary:

```python
>>> alignment.column_annotations
{'state': 'DIDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDIDDIDDDDDDDDDDDDDDDDDDDDDDDIIIDDDDDDDDDDDDDDDDDDDDDD'}
```

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

```python
>>> print(format(alignment, "a2m"))
>plas_horvu
D.VLLGANGVLFEPNDFSVKAGETITFKNNAGYPHNVFDEDAVPSG.VD.VSKISQEEYLTAPGETFSVTLT...PGTYGFYCEPHAGAMGVGTIV
>plas_chlre
-.VKLGADVGEVFPEPNTLTSKGETVNFVNNAGPHNVFVDADAPSG.VN.ADAISRDYLNAPGETYSKL...AGEYGFYCEPHQAGMVGKIV
>plas_anava
-.VKLGADVGEVFPEPNTLTSKGETVNFVNNAGPHNVFVDADAPSG.VN.ADAISRDYLNAPGETYSKL...AGEYGFYCEPHQAGMVGKIV
>plas_proho
VqIKMGTDKDYAPLYEPKALSISAGDTEVFMNKGPHNIFDK--VPAG.EA.PALSNTKLRIAPGSFYSVTILT...PGTYGFYCEPHAGAMGVGTIV
>azup_achcy
VhMLNKGDAVGFEASGLKAPGADTEVFMNKGPHNIFDK--VPAG.EA.PALSNTKLRIAPGSFYSVTILT...PGTYGFYCEPHAGAMGVGTIV
```

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
GAAAAGGAAAGTACGGCCCGCCACTCCGGGTGTTGCTAGGAGGGCTTA
>mm9
GAAAGGAAAGTACGGCCCGCCACTCCGGGTGTTGCTAGGAGGGCTTA
>canFam2
CAAGCCTGCGCGTCAGCCGGATGTCGCCGGGCCCCGCTGCTTTGCCTTT
```

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

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

117
#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
AAGCCCTG--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:

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

The file header data are stored in the metadata attribute:

```python
>>> 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:

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

These identifiers are used in the individual alignments:

```python
>>> for alignment in alignments:
...     print([record.id for record in alignment.sequences])
...     print(alignment)
...     print("*****")
...```
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
TAAGCCCTCCTAGCACACACCAGGAGTTGGC-CGGCCCTACTTCTTTT 1
> 2:1-49 + canFam2.fa
CAAGGCTGGACCCGGGAGTTGTCCCCGGGCGCCTGCTTTCTTTT C
> 3:1-19 - mm9.fa
-----------------------------------GGATCTACTTTTTC
= 119
```
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:

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

The file header data now does not include the input file name:

```python
>>> alignments.metadata  # doctest: +NORMALIZE_WHITESPACE
{'FormatVersion': 'Mauve1',
 'BackboneFile': 'separate.xmfa.bbcols'}
```

The `identifiers` attribute stores the sequence identifiers for the three sequences:

```python
>>> alignments.identifiers
['equCab1.fa', 'canFam2.fa', 'mm9.fa']
```

These identifiers are used in the individual alignments:

```python
>>> for alignment in alignments:
...    print([record.id for record in alignment.sequences])
...    print(alignment)
...    print("******")
...['equCab1.fa', 'canFam2.fa', 'mm9.fa']
equCab1.f 50 TAAGCCCTCTGCGACACACCCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC 0
canFam2.f 0 CAAGCCCTGC--GCGGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC 49
mm9.fa 19 ---------------------------------GGATCTACTTTTCCTCTTC 0

******
['mm9.fa']
mm9.fa 19 CTGGCGTCCGGAGCTGGGACGT 41

******
```

To output the alignments in Mauve format, use `Align.write`:

```python
>>> 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
```

120
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`:

```python
>>> 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
TAAGCCCTCCTAGCACACACCCGGAGTG GCC-GGGCCGTA C-TTTCCCTTTT
> 2:1-49 + canFam2.fa
CAAGCCCTGC--GGGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCCTTTT
> 3:1-19 - mm9.fa
---------------------------------GGATCTACTTTTCCCTCTTC
=
> 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):

```python
>>> alignments = Align.Alignments(alignments)
>>> alignments.metadata = metadata
>>> alignments.identifiers = identifiers
>>> stream = StringIO()
>>> Align.write(alignments, stream, "mauve", metadata=metadata, identifiers=identifiers)
```

```python
>>> print(stream.getvalue()) # doctest: +NORMALIZE_WHITESPACE
```

```plaintext
#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
TAAGCCCTCCTAGCACACACCGGAGTGGCC-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:

```python
>>> alignment = alignments[0]
>>> print(alignment.format("mauve", metadata=metadata, identifiers=identifiers))
```

```plaintext
> 1:1-50 - equCab1.fa
TAAGCCCTCCTAGCACACACCGGAGTGGCC-GGGCCGTAC-TTTCCTTTTC
> 2:1-49 + canFam2.fa
CAAGCCCTGC--GCGCTCAGCCGGAGTGTCCCGGGCCCTGC-TTTCCTTTTC
> 3:1-19 - mm9.fa
---------------------------------GGATCTACTTTTCCTCTTC
=  
> 3:20-41 + mm9.fa
CTGGCGTCCGGAGCTGGGACGT

<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:

```plaintext
EAS56_57:6:190:289:82 69 chr1 100 0 * = 100 0 CTCAAGGTTGTTGCAAGGGGGTCTATGTGAACAAA <<<7<<<;<<<<<<<<8;;<7;4<;<;;;;;94<; MF:i:192
EAS56_57:6:190:289:82 137 chr1 100 73 35M = 100 0 AGGGGTGCAGAGCCGAGTCACGGGGTTGCCAGCAC <<<<<<;<<<<<<<<<<;<<;<<<<;8<6;9;;2; MF:i:64 Aq:i:0 NM:i:0 UQ:i:0 H0:i:1 H1:i:0
EAS51_64:3:190:727:308 99 chr1 103 99 35M = 263 195 GGTGCAGAGCCGAGTCACGGGGTTGCCAGCACAGG <<<<<<<<<<<<<<<<<<<<<<<<<<<::<<<844 MF:i:18 Aq:i:73 NM:i:0 UQ:i:0 H0:i:1 H1:i:0
...

122
To parse this file, use

```python
>>> 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`:

```python
>>> alignment.flag
69
>>> bin(69)
'0b1000101'
>>> bin(4)
'0b100'
>>> if alignment.flag & 4:
...     print("unmapped")
... else:
...     print("mapped")
...
unmapped
```

```python
>>> alignment.sequences
[None, SeqRecord(seq=Seq('CTCAAGGTTGGTCAAGGGGGTTCTATGTGAACAAA'), id='EAS56_57:6:190:289:82', name='<unknown name>', description='', dbxrefs=[])]
```

```python
>>> alignment.target is None
True
```

The second line represents an alignment to chromosome 1:

```python
>>> alignment = next(alignments)
>>> if alignment.flag & 4:
...     print("unmapped")
... else:
...     print("mapped")
...
mapped
```

```python
>>> 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:

```python
>>> format(alignment, "sam")
'EAS56_57:6:190:289:82\tchr1\t100\t73\tM\t=	100\t0\tAGGGGTGAGCTGACAGCGGTGCCAGCAC\t<<<<<<;<<<<<<<<<<;<<;<<<<;8<6;9;;2;
```

```python
>>> format(alignment, "bed")
'chr1\tt99\tt134\tEAS56_57:6:190:289:82\t0\t+\tt99\tt134\tt1\tt35,\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`:

```python
>>> 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:

```python
>>> 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	0	0	0	0	0	0	0	+	EAS56_57:6:190:289:82	35	0	35	chr1	1575	99	134	1	35,	0,	99,
'
```

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:

```text
@HD	VN:1.3	SO:coordinate
@SQ	SN:chr1	LN:1575
@SQ	SN:chr2	LN:1584
EAS56_57:6:190:289:82 69 chr1 100 0 * = 100 0 CTCAAGGTTGTTCAGGGGGTCTATGTGAACAAA

... 
```

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:

```python
>>> 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:

```python
>>> alignments.metadata
{'HD': {'VN': '1.3', 'SO': 'coordinate'}}
```

With the target information, we can now also print the alignment in PSL format:

```python
>>> 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,
'
```

We can now also print the alignment in human-readable form, but note that the target sequence contents is not available from this file:

```python
>>> print(alignment)
chr1 99 ??????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????? 
```
The parser reconstructs the local genome sequence from the MD tag, allowing us to see the target sequence explicitly when printing the alignment:

```python
>>> from Bio import Align
>>> alignments = Align.parse("sam1.sam", "sam")
>>> for alignment in alignments:
...     if not alignment.flag & 4:  # Skip the unmapped lines
...         break
...     
... # doctest: +ELLIPSIS
>>> print(alignment)
<Alignment object (2 rows x 101 columns) at ...>

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:

```python
>>> from Bio import Align
>>> alignments = Align.parse("dna_rna.sam", "sam")
>>> alignment = next(alignments)
>>> alignment = next(alignments)
>>> alignment = next(alignments)

# doctest: +NORMALIZE_WHITESPACE
>>> print(format(alignment, "SAM"))
NR_111921.1 0 chr3 48663768 0 46M1827N82M3376N76M12H * 0 0 ... * AS:i:1000 NM:i:0

>>> 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:

```python
>>> from io import StringIO
>>> import numpy as np
>>> from Bio import Align

125
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")

print(stream.getvalue())

6.7.14 Browser Extensible Data (BED)

BED (Browser Extensible Data) files are typically used to store the alignments of gene transcripts to the genome. See the description from UCSC for a full explanation of the BED format.

BED files have three required fields and nine optional fields. The file bed12.bed in subdirectory Tests/Blat is an example of a BED file with 12 fields:
chr22  1000  5000  mRNA1  960  +  1200  4900  255,0,0  2  567,488, 0,3512,
chr22  2000  6000  mRNA2  900  -  2300  5960  0,255,0

To parse this file, use

```python
>>> 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:

```python
>>> alignment.target
SeqRecord(seq=Seq(None, length=9223372036854775807), id='chr22', name='<unknown name>', description='', dbxrefs=[])
```

The length of the query sequence can be inferred from its alignment information:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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
```
>>> print(alignment.format("bed", 6))  # doctest: +NORMALIZE_WHITESPACE
chr22  2000  6000  mRNA2  900  -

The same keyword argument can be used with `Align.write`:

```python
>>> 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:

```python
>>> from Bio import Align
>>> alignments = Align.parse("dna_rna.bb", "bigbed")
```

```python
>>> len(alignments)
4
```

```python
>>> 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"
)
```

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:
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):

```python
>>> 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
```

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):

```python
>>> alignments.rewind()
>>> for alignment in alignments:
...     print(format(alignment, "bed")) # doctest: +NORMALIZE_WHITESPACE
...    
...    chr3 42530895 42532606 NR_046654.1 1000 - 42530895 42532606 0 3 63,75,43, 0,1125,1668,

```
Use the search method on the alignments object to find regions on chr3 between positions 48000000 and 49000000. This method returns an iterator:

```python
>>> 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:

```python
>>> 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

```python
>>> 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:

```python
>>> 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)
"UCSC Genes for chr7 with color plus GeneSymbol and SwissProtID"
{
    string chrom;  "Reference sequence chromosome or scaffold"
    uint chromStart;  "Start position of feature on chromosome"
    uint chromEnd;  "End position of feature on chromosome"
    string name;  "Name of gene"
    uint score;  "Score"
    char[1] strand;  "+ or - for strand"
    uint thickStart;  "Coding region start"
    uint thickEnd;  "Coding region end"
    uint reserved;  "Green on + strand, Red on - strand"
    string geneSymbol;  "Gene Symbol"
    string spID;  "SWISS-PROT protein Accession number"
}
```

Now we can write a bigBed file with the 9 BED fields plus the additional fields geneSymbol and spID by calling
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 strand Q Q Q Q T T T T block blockSizes qStarts tStarts
```

<table>
<thead>
<tr>
<th>match</th>
<th>mis-match</th>
<th>rep. match</th>
<th>N's</th>
<th>Q gap count</th>
<th>Q gap bases</th>
<th>T gap count</th>
<th>T gap bases</th>
<th>strand</th>
<th>block blockSizes</th>
<th>qStarts</th>
<th>tStarts</th>
</tr>
</thead>
<tbody>
<tr>
<td>165</td>
<td>0</td>
<td>39</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5203</td>
<td>+</td>
<td>NR_111921.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>175</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1530</td>
<td>-</td>
<td>NR_046654.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>162</td>
<td>2</td>
<td>39</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5204</td>
<td>+</td>
<td>NR_111921.1_modified</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>172</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1532</td>
<td>-</td>
<td>NR_046654.1_modified</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To parse this file, use

```python
>>> 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:

```python
>>> 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:

```python
>>> alignment.matches
162
```
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:

```python
>>> 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:

```python
>>> print(format(alignment, "bed"))  # doctest: +NORMALIZE_WHITESPACE
chr3 48663767 48669174 NR_111921.1_modified 0 + 48663767 48669174 0 5 28,17,76,6,76, 0,29,1873,1949,5331,
```

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.

```python
>>> 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
```

```python
>>> line = alignment.format("psl")  # doctest: +NORMALIZE_WHITESPACE
6 1 0 1 0 0 0 0 0 + query 8 0 8 target 20 7 15 1 8, 0, 7,
```

```python
>>> line = alignment.format("psl", mask="lower")
```
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:

```python
>>> 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:

```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 15 1 8, 0, 7,'
```

Note that optional keyword arguments cannot be used with the format function or with formatted strings.

### 6.7.17 bigPsl

A bigPsl file is a bigBed file with a BED12+13 format consisting of the 12 predefined BED fields and 13 custom fields defined in the AutoSql file bigPsl.as provided by UCSC, creating an indexed binary version of a PSL file (see section 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).

```python
>>> 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 target"
  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"
```
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:

```python
>>> alignments.targets
[SeqRecord(seq=Seq(None, length=198295559), id='chr3', name='<unknown name>', description='<unknown description>', dbxrefs=[])]
```

Iterating over the alignments gives one Alignment object for each line:

```python
>>> 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):

```python
>>> alignment.coordinates
array([[48663767, 48663795, 48663796, 48663813, 48665640, 48665716,
        48665716, 48665722, 48669098, 48669174],
       [  3,  31,  31,  48,  48, 124,
         126, 132, 132, 208]])
```

```python
>>> 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:

```python
>>> print(format(alignment, "bed"))  # doctest: +NORMALIZE_WHITESPACE
chr3  48663767  48669174  NR_111921.1_modified  1000  +  48663767
>>> print(format(alignment, "psl"))  # doctest: +NORMALIZE_WHITESPACE
162  2   39  0  1  2  3  5204  +  NR_111921.1_modified
```

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:

```python
>>> 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.
- **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 alignments 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:

```plaintext
track name=euArc visibility=pack mafDot=off frames="multiz28wayFrames" speciesOrder="hg16 panTro1 baboon mm4 rn3" description="A sample alignment"
##maf version=1 scoring=tba.v8
# tba.v8 (((human chimp) baboon) (mouse rat))
# multiz.v7
# maf_project.v5 _tba_right.maf3 mouse _tba_C
# single_cov2.v4 single_cov2 /dev/stdin

a score=23262.0
s hg16.chr7 27578828 38 + 158545518 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s panTro1.chr6 28741140 38 + 161576975 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s baboon 116834 38 + 4622798 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s mm4.chr6 53215344 38 + 151104725 -AATGGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s rn3.chr4 81344243 40 + 187371129 -AA-GGGATGCTAAACCAATGAGTTGTCTCTCAATG

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

```python
>>> 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:

```python
>>> 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:

```python
>>> 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, 53215346, 53215346, 53215366, 53215366, 53215382],
       [81344243, 81344245, 81344245, 81344245, 81344264, 81344267, 81344283]])
>>> print(alignment)
hg16.chr7 27578828 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTCTTACGGTG 27578866
panTro1.chAAA-GGGAATGTTAACCAAATGA---ATTGTCTCTCTTACGGTG 28741178
baboon AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTCTTACGGTG
mm4.chr6 116834 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTCTTACGGTG 116872
rn3.chr4 81344243 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTCTTACGGTG 81344283

>>> print(format(alignment, "phylip"))
 5 42
hg16.chr7 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTCTTACGGTG
panTro1.chAAA-GGGAATGTTAACCAAATGA---ATTGTCTCTCTTACGGTG
baboon AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTCTTACGGTG
mm4.chr6 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTCTTACGGTG
rn3.chr4 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTCTTACGGTG
<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---CCTTTGCTAGAATTTTGTACTTACGTTAAAGTAG
i 1g18.chr6 15870786 46 - 170899992 CCACTACATTCTTTATGAGAA-TTTGTTTATCTCTAAAC-TTCC
i 1g18.chr6 9085 0 C 0
s panTro2.chr6 16389355 46 - 173908612 CTTATACCTTTCTTTATGAGAA-TTTGTTTATCTCTAAAC-TTCC
q panTro2.chr6 9999999999999999999-9999999999999999999
i panTro2.chr6 9106 0 C 0
s calJac1.Contig6394 6182 46 + 133105 CTTATACCTTTCTTTATGAGAA-TTTGTTTATCTCTAAAC-TTCC
i calJac1.Contig6394 0 0 C 0
s loxAfr1.scaffold_75566 1167 34 - 10574 -----------TTTGTTGTAAGA-TTTATGCTTTAATATACAATGCTCTCAATGG
q loxAfr1.scaffold_75566 9999999999999999999-9999999999999999999
i loxAfr1.scaffold_75566 0 0 C 0
e tupTel1.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
```

137
This is the 10th alignment block in the file `ucsc_mm9_chr10.maf` (available in the Tests/MAF subdirectory of the Biopython distribution):

```python
>>> 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   301464  CCTGTACC---CTTTGGTGAGAATTTTTGTTTCAGTGTTAAAAGTTTG  3014689
hg18.chr6   155029206  CCTATACCTTTCTTTTATGAGA-TTTTGTTTTAATCCTAAAC-TTTT  155029160
panTro2.c   157519257  CCTATACCTTTCTTTTATGAGA-TTTTGTTTTAATCCTAAAC-TTTT  157519211
calJac1.C   6182     CCTATACCTTTCTTTTATGAGA-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:

```python
>>> 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:

```python
>>> alignment.sequences[2].annotations["quality"]
'9999999999999999999999999999999999999999999999'
>>> alignment.sequences[4].annotations["quality"]
'9999969999999999999999998699989997'
```

The "e" lines show information about species with a contiguous sequence before and after this alignment block, but with no aligning nucleotides in this alignment block. This is stored under the "empty" key of the `alignment.annotations` dictionary:

```python
>>> alignment.annotations["empty"]  # doctest: +NORMALIZE_WHITESPACE
[(SeqRecord(seq=Seq(None, length=498454), id='tupBel1.scaffold_114895.1-498454', name='', description='', dbxrefs=[]), (331078, 326933), 'I'),
 (SeqRecord(seq=Seq(None, length=100002), id='echTel1.scaffold_288249', name='', description='', dbxrefs=[]), (87661, 95225), 'I'),
 (SeqRecord(seq=Seq(None, length=359464), id='otoGar1.scaffold_334.1-359464', name='', description='', dbxrefs=[]), (178247, 175316), 'I'),
 (SeqRecord(seq=Seq(None, length=174210431), id='ponAbe2.chr6', name='', description='', dbxrefs=[]), (158040498, 158048983), 'I')]
```

This shows for example that there were non-aligning bases inserted ("I") from position 158040939 to 158048983 on the opposite strand of the `ponAbe2.chr6` genomic sequence. Again, see the UCSC documentation for the full definition of "e" lines.

To print an alignment in MAF format, you can use Python's built-in `format` function:
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, creating 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

```python
>>> 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"
)```
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:

```python
>>> 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:

```python
>>> 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, 116837, 116837, 116837],
[53215344, 53215344, 53215347, 53215347, 53215366, 53215366, 53215382],
[81344243, 81344243, 81344245, 81344245, 81344245, 81344245, 81344245]])
>>> print(alignment)
hg16.chr7 27578828 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTCTTACGGTG 27578866
panTro1.chr6 28741140 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTCTTACGGTG 28741178
baboon 116834 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTCTTACGGTG 116872
mm4.chr6 53215344 -AATGGGAATGCTAAGCCAATGAGTTGTTGTCTCTCTTACGGTG 53215382
rn3.chr4 81344243 -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCTTACGGTG 81344283

>>> print(format(alignment, "phylip"))
5 42
hg16.chr7 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
panTro1.chr6 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
baboon AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
mm4.chr6 -AATGGGAATGCTAAGCCAATGAGTTGTTGTCTCTCTTACGGTG
rn3.chr4 -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCTTACGGTG
```

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):

```python
>>> 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)

<table>
<thead>
<tr>
<th>mm9.chr10</th>
<th>3014644</th>
<th>CCTGTACC---CTTTGGTGAGA-AAAAAAGTGTTTGAGGAATTTTTGTTTCAGTGTTAAAAGTTTG</th>
</tr>
</thead>
<tbody>
<tr>
<td>hg18.chr6</td>
<td>155029206</td>
<td>CCTACACCTTTCTTTATGAGAA-TTTGGTTTTAATCTAACATTCTAAAC-TTTT</td>
</tr>
<tr>
<td>panTro2.c</td>
<td>157519257</td>
<td>CCTACACCTTTCTTTATGAGAA-TTTGGTTTTAATCTAACATTCTAAAC-TTTT</td>
</tr>
<tr>
<td>calJac1.C</td>
<td>6182</td>
<td>CCTATACCTTTCTTTATGAGAA-TTTGGTTTTAATCTAACATTCTAAAC-TTTT</td>
</tr>
<tr>
<td>loxAfr1.s</td>
<td>9407</td>
<td>-------------------TTTGGTTTAGAA-TTATGCTTTAATTCAAAAC-TTCC</td>
</tr>
</tbody>
</table>

>>> print(format(alignment, "MAF"))
a score=19159.000000

<table>
<thead>
<tr>
<th>s mm9.chr10</th>
<th>3014644</th>
<th>45 + 129993255</th>
<th>CCTGTACC---CTTTGGTGAGA-AAAAAAGTGTTTGAGGAATTTTTGTTTCAGTGTTAAAAGTTTG</th>
</tr>
</thead>
<tbody>
<tr>
<td>s hg18.chr6</td>
<td>15870786</td>
<td>46 - 170899992</td>
<td>CCTACACCTTTCTTTATGAGAA-TTTGGTTTTAATCTAACATTCTAAAC-TTTT</td>
</tr>
<tr>
<td>i panTro2.chr6</td>
<td>16389355</td>
<td>I 9085 C 0</td>
<td></td>
</tr>
<tr>
<td>q panTro2.chr6</td>
<td>16389355</td>
<td>46 - 173908612</td>
<td>CCTACACCTTTCTTTATGAGAA-TTTGGTTTTAATCTAACATTCTAAAC-TTTT</td>
</tr>
<tr>
<td>i panTro2.chr6</td>
<td>16389355</td>
<td>I 9016 C 0</td>
<td></td>
</tr>
<tr>
<td>s calJac1.Contig6394</td>
<td>6182</td>
<td>46 + 133105</td>
<td>CCTACACCTTTCTTTATGAGAA-TTTGGTTTTAATCTAACATTCTAAAC-TTTT</td>
</tr>
<tr>
<td>i calJac1.Contig6394</td>
<td>6182</td>
<td>I 9016 C 0</td>
<td></td>
</tr>
<tr>
<td>s loxAfr1.scaffold_75566</td>
<td>1167</td>
<td>34 - 10574</td>
<td>-------------------TTTGGTTTAGAA-TTATGCTTTAATTCAAAAC-TTCC</td>
</tr>
<tr>
<td>q loxAfr1.scaffold_75566</td>
<td>1167</td>
<td>I 9085 C 0</td>
<td></td>
</tr>
</tbody>
</table>

To write a complete bigMaf file, use

```python
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:

```python
>>> 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

```python
>>> from Bio import Align
>>> alignments = Align.parse("psl_34_001.chain", "chain")
```

Iterate over alignments to get one `Alignment` object for each chain:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> print(format(alignment, "chain"))  # doctest: +NORMALIZE_WHITESPACE
chain 41 chr2 243199373 + 183925984 183926028 hg19_dna 50 + 1 49 7
6 0 4
38
```

We can also print the alignment in a few other alignment file formats:
>>> print(format(alignment, "BED"))  # doctest: +NORMALIZE_WHITESPACE
chr2   183925984  183926028   hg19_dna   41   +   183925984  183926028

>>> print(format(alignment, "PSL"))  # doctest: +NORMALIZE_WHITESPACE
  44   0   0   0   1   4   0   0   0   +   hg19_dna   50

>>> print(format(alignment, "exonerate"))
vulgar: hg19_dna 1 49 + chr2 183925984 183926028 + 41 M 6 6 G 4 0 M 38 38

>>> print(alignment.format("exonerate", "cigar"))
cigar: hg19_dna 1 49 + chr2 183925984 183926028 + 41 M 6 I 4 M 38

>>> print(format(alignment, "sam"))  # doctest: +NORMALIZE_WHITESPACE
hg19_dna   0   chr2   183925985   255   1S6M4I38M1S   *   0   0
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:

```python
>>> 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:

```python
>>> aligner = Align.PairwiseAligner(match_score=1.0)
```
or after the object is made:

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

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

```python
>>> 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:

```python
>>> alignments = aligner.align(target, query)
>>> alignment = alignments[0]
>>> alignment
<Alignment object (2 rows x 5 columns) at ...>
```

Iterate over the Alignment objects and print them to see the alignments:
for alignment in alignments:
...
print(alignment)
...
target 0 GAACT 5
  0 1---1 5
query 0 GA--T 3
<BLANKLINE>
target 0 GAACT 5
  0 1-1-1 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]],
       [[0, 2],
       [2, 3]])

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

>>> alignment = alignments[1]
>>> print(alignment)
<BLANKLINE>
>>> alignment.aligned
array([[0, 1],
       [2, 3],
       [4, 5]],
       [[0, 1],
       [1, 2],
       [2, 3]])

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

>>> aligner.mode = "global"
>>> aligner.mismatch_score = -10
>>> alignments = aligner.align("AAACAAA", "AAAGAAA")
>>> len(alignments)
2
>>> print(alignments[0])
<BLANKLINE>
>>> alignments[0].aligned
array([[0, 3],
       [4, 7]],
       [[0, 3],
       [4, 7]])
>>> print(alignments[1])
<BLANKLINE>
>>> alignments[1].aligned
array([[0, 3],
       [4, 7]],
       [[0, 3],
       [4, 7]])
The `map` method can be applied on a pairwise alignment `alignment1` to find the pairwise alignment of the query of `alignment2` to the target of `alignment1`, where the target of `alignment2` and the query of `alignment1` are identical. A typical example is where `alignment1` is the pairwise alignment between a chromosome and a transcript, `alignment2` is the pairwise alignment between the transcript and a sequence (e.g., an RNA-seq read), and we want to find the alignment of the sequence to the chromosome:

```python
>>> aligner.mode = "local"
>>> aligner.open_gap_score = -1
>>> aligner.extend_gap_score = 0
>>> chromosome = "AAAAAAAACCCCCAAAAAAAGGGGGGAAAAAAA"
>>> transcript = "CCCCCGGGGG"
>>> alignments1 = aligner.align(chromosome, transcript)
>>> len(alignments1)
1
>>> alignment1 = alignments1[0]
>>> print(alignment1)
target 8 CCCCCCCAAAAAAAAAAAGGGGGG 32
  0 |||||||-----------|||||| 24
query 0 CCCCCCC-----------GGGGGG 13

>>> sequence = "CCCCGGGG"
>>> alignments2 = aligner.align(transcript, sequence)
>>> len(alignments2)
1
>>> alignment2 = alignments2[0]
>>> print(alignment2)
target 3 CCCCGGGG 11
  0 |||||||| 8
query 0 CCCCGGGG 8

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

>>> format(mapped_alignment, "psl")
'8\t0\t0\t0\t0\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):

```python
>>> 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\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":

```python
>>> aligner.mode = "local"
>>> target = "AGAACTC"
>>> query = "GAACT"
>>> score = aligner.score(target, query)
>>> score
5.0
>>> alignments = aligner.align(target, query)
>>> for alignment in alignments:
...    print(alignment)
...
target 1 GAACT 6
      0 ||||| 5
query 0 GAACT 5
```

Note that there is some ambiguity in the definition of the best local alignments if segments with a score 0 can be added to the alignment. We follow the suggestion by Waterman & Eggert [49] 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

```python
>>> 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
    target_right_open_gap_score: 0.000000
    target_right_extend_gap_score: 0.000000
    query_internal_open_gap_score: 0.000000
    query_internal_extend_gap_score: 0.000000
    query_left_open_gap_score: 0.000000
    query_left_extend_gap_score: 0.000000
    query_right_open_gap_score: 0.000000
    query_right_extend_gap_score: 0.000000
    mode: local
```

See Sections 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

```python
>>> aligner.algorithm
'Smith-Waterman'
```

This attribute is read-only.

A PairwiseAligner object also stores the precision \( \epsilon \) to be used during alignment. The value of \( \epsilon \) is stored in the attribute `aligner.epsilon`, and by default is equal to \( 10^{-6} \):

```python
>>> 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 \( \epsilon \).

### 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> from Bio.Align import substitution_matrices
>>> substitution_matrices.load()  # doctest: +ELLIPSIS
['BENNER22', 'BENNER6', 'BENNER74', 'BLASTN', 'BLASTP', 'BLOSUM45', 'BLOSUM50', 'BLOSUM62', ..., 'TRANS']
>>> matrix = substitution_matrices.load("BLOSUM62")
>>> print(matrix)  # doctest: +ELLIPSIS
# Matrix made by matblas from blosum62.iij
...
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:

```python
>>> 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:

\[
gap\text{ score} = \text{open gap score} + (n - 1) \times \text{extend gap score},
\]

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.
### 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:

```python
>>> 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.

<table>
<thead>
<tr>
<th>Meta-attribute</th>
<th>Attributes it maps to</th>
</tr>
</thead>
<tbody>
<tr>
<td>gap_score</td>
<td>target_gap_score, query_gap_score</td>
</tr>
<tr>
<td>open_gap_score</td>
<td>target_open_gap_score, query_open_gap_score</td>
</tr>
<tr>
<td>extend_gap_score</td>
<td>target_extend_gap_score, query_extend_gap_score</td>
</tr>
<tr>
<td>internal_gap_score</td>
<td>target_internal_gap_score, query_internal_gap_score</td>
</tr>
<tr>
<td>internal_open_gap_score</td>
<td>target_internal_open_gap_score, query_internal_open_gap_score</td>
</tr>
<tr>
<td>internal_extend_gap_score</td>
<td>target_internal_extend_gap_score, query_internal_extend_gap_score</td>
</tr>
<tr>
<td>end_gap_score</td>
<td>target_end_gap_score, query_end_gap_score</td>
</tr>
<tr>
<td>end_open_gap_score</td>
<td>target_end_open_gap_score, query_end_open_gap_score</td>
</tr>
<tr>
<td>end_extend_gap_score</td>
<td>target_end_extend_gap_score, query_end_extend_gap_score</td>
</tr>
<tr>
<td>left_gap_score</td>
<td>target_left_gap_score, query_left_gap_score</td>
</tr>
<tr>
<td>right_gap_score</td>
<td>target_right_gap_score, query_right_gap_score</td>
</tr>
<tr>
<td>left_open_gap_score</td>
<td>target_left_open_gap_score, query_left_open_gap_score</td>
</tr>
<tr>
<td>right_open_gap_score</td>
<td>target_right_open_gap_score, query_right_open_gap_score</td>
</tr>
<tr>
<td>left_extend_gap_score</td>
<td>target_left_extend_gap_score, query_left_extend_gap_score</td>
</tr>
<tr>
<td>right_extend_gap_score</td>
<td>target_right_extend_gap_score, query_right_extend_gap_score</td>
</tr>
<tr>
<td>target_open_gap_score</td>
<td>target_internal_open_gap_score, target_left_open_gap_score, query_left_open_gap_score,</td>
</tr>
<tr>
<td>target_gap_score</td>
<td>target_open_gap_score, target_internal_gap_score, query_left_open_gap_score, query_left_open_gap_score, query_right_open_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, query_end_gap_score, query_end_open_gap_score, query_end_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_right_gap_score, query_right_open_gap_score, query_right_extend_gap_score, target_open_gap_score, target_open_gap_score, target_gap_score, target_extend_gap_score, gap_score</td>
</tr>
</tbody>
</table>
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.

```python
>>> 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 -1.0 -1.0 -1.0 -1.0
T -3.0  2.0 -3.0 -3.0 -3.0 -1.0 -1.0 -3.0 -3.0 -1.0 -1.0 -1.0 -1.0 -1.0
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 -1.0 -1.0
C -3.0 -3.0 -3.0  2.0 -1.0 -3.0 -1.0 -3.0 -1.0 -3.0 -1.0 -1.0 -1.0 -1.0
```
7.7 Iterating over alignments

The alignments returned by `aligner.align` are a kind of immutable iterable objects (similar to `range`). While they appear similar to a `tuple` or `list` of `Alignment` objects, they are different in the sense that each `Alignment` object is created dynamically when it is needed. This approach was chosen because the number of alignments can be extremely large, in particular for poor alignments (see Section 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 9,223,720,368,547,758,07, 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.

```python
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> alignments = aligner.align("AAA", "AA")
>>> len(alignments)
3
```

- You can extract a specific alignment by index:

```python
>>> 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
```

- You can iterate over alignments, for example as in

```python
>>> 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:

```python
>>> 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

>>> alignments.rewind()
>>> for alignment in alignments:
...    print(alignment)
... target 0 AAA 3
... 0 ||- 3
query 0 AA- 2

You can also convert the `alignments` iterator into a list or tuple:

```python
>>> 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:

```python
>>> 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 \texttt{strand="-"}:

```python
>>> from Bio import Align
>>> from Bio.Seq import reverse_complement
>>> target = "AAAAACC"
>>> 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 \texttt{strand="-"} when calling \texttt{aligner.align}:

```python
>>> alignments = aligner.align(target, query)
>>> len(alignments)
1
>>> print(alignments[0])
target 0 AAAACCC 7
0 --\\|
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 --\\|
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
```

157
query 4 ------GGTT 0
>>> print(alignments[1])
target 0 ----AAAACCC 7
0 -------- 11
query 4 GGTT------ 0

Note that the score for aligning query to the reverse strand of target may be different from the score
for aligning the reverse complement of query to the forward strand of target if the left and right gap scores
are different:

```python
>>> aligner.left_gap_score = -0.5
>>> aligner.right_gap_score = -0.2
>>> aligner.score(target, query)
2.8
>>> alignments = aligner.align(target, query)
>>> len(alignments)
1
>>> print(alignments[0])
target 0 AAAACCC 7
0 --|||- 7
query 0 --AACC- 4

>>> aligner.score(target, reverse_complement(query), strand="-")
3.1
>>> alignments = aligner.align(target, reverse_complement(query), strand="-")
>>> len(alignments)
1
>>> print(alignments[0])
target 0 AAAACCC 7
0 --|||- 7
query 4 --AACC- 0
```

### 7.9 Substitution matrices

Substitution matrices [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`:

```python
>>> 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`:

```python
>>> 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:

```python
>>> 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,

```python
>>> a = Array("ABCD", dims=2, data=np.arange(16).reshape(4, 4))
>>> print(a)
A  B  C  D
A  0.0 1.0 2.0 3.0
B  4.0 5.0 6.0 7.0
C  8.0 9.0 10.0 11.0
D 12.0 13.0 14.0 15.0

>>> b = a.select("CAD")
>>> print(b)
C  A  D
C 10.0 8.0 11.0
A  2.0 0.0 3.0
D 14.0 12.0 15.0

Note that this also allows you to reorder the alphabet.
Data for letters that are not found in the alphabet are set to zero:

```python
>>> c = a.select("DEC")
>>> print(c)
D  E  C
D 15.0 0.0 14.0
E  0.0 0.0  0.0
C 11.0 0.0 10.0
```

As the `Array` class is a subclass of numpy array, it can be used as such. A `ValueError` is triggered if the `Array` objects appearing in a mathematical operation have different alphabets, for example

```python
>>> from Bio.Align.substitution_matrices import Array
>>> d = Array("ACGT")
>>> r = Array("ACGU")
>>> d + r
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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> alignment = alignments[0]
>>> substitutions = alignment.substitutions
>>> print(substitutions)
A  C  G  T
A 307.0 19.0 34.0 19.0
C 15.0 280.0 25.0 29.0
G 34.0 24.0 401.0 20.0
T 24.0 36.0 20.0 228.0
```

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

```python
>>> 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
```

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:

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

The number of substitutions expected at random is simply the product of the background distribution with itself:
```python
>>> expected_frequencies = background[:, None].dot(background[None, :])
>>> print(format(expected_frequencies, "%.4f"))

A   C   G   T
A  0.0627 0.0585 0.0793 0.0499
C  0.0585 0.0546 0.0740 0.0466
G  0.0793 0.0740 0.1002 0.0631
T  0.0499 0.0466 0.0631 0.0397

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

```python
>>> oddsratios = observed_frequencies / expected_frequencies
>>> import numpy as np
>>> scoring_matrix = np.log2(oddsratios)
>>> print(scoring_matrix)

A   C   G   T
A  1.7 -2.4 -1.8 -1.8
C -2.4  1.8 -2.2 -1.1
G -1.8 -2.2  1.4 -2.3
T -1.8 -1.1 -2.3  1.9
```

The matrix can be used to set the substitution matrix for the pairwise aligner (see Chapter 7):

```python
>>> 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):

```python
>>> 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:

```python
>>> substitutions = alignment.substitutions
```

To make the example more readable, we'll select only amino acids with polar charged side chains:

```python
>>> substitutions = substitutions.select("DEHKR")
```

To the make the example more readable, we'll select only amino acids with polar charged side chains:

```python
>>> print(substitutions)
```

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>E</th>
<th>H</th>
<th>K</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>2360.0</td>
<td>270.0</td>
<td>15.0</td>
<td>1.0</td>
<td>48.0</td>
</tr>
<tr>
<td>E</td>
<td>241.0</td>
<td>3305.0</td>
<td>15.0</td>
<td>45.0</td>
<td>2.0</td>
</tr>
<tr>
<td>H</td>
<td>0.0</td>
<td>18.0</td>
<td>1235.0</td>
<td>8.0</td>
<td>0.0</td>
</tr>
<tr>
<td>K</td>
<td>0.0</td>
<td>9.0</td>
<td>24.0</td>
<td>3218.0</td>
<td>130.0</td>
</tr>
<tr>
<td>R</td>
<td>2.0</td>
<td>2.0</td>
<td>17.0</td>
<td>103.0</td>
<td>2079.0</td>
</tr>
</tbody>
</table>
```
Rows and columns for other amino acids were removed from the matrix.

Next, we normalize the matrix and make it symmetric.

```python
>>> observed_frequencies = substitutions / substitutions.sum()
>>> observed_frequencies = (observed_frequencies + observed_frequencies.transpose()) / 2.0
>>> print(format(observed_frequencies, "%.4f"))
D   E   H   K   R
D 0.1795 0.0194 0.0006 0.0000 0.0019
E 0.0194 0.2514 0.0013 0.0021 0.0002
H 0.0006 0.0013 0.0939 0.0012 0.0006
K 0.0000 0.0021 0.0012 0.2448 0.0089
R 0.0019 0.0002 0.0006 0.0089 0.1581
```

Summing over rows or columns gives the relative frequency of occurrence of each residue:

```python
>>> background = observed_frequencies.sum(0)
>>> print(format(background, "%.4f"))
D 0.2015
E 0.2743
H 0.0976
K 0.2569
R 0.1697
```

```python
>>> background.sum()
1.0
```

The expected frequency of residue pairs is then

```python
>>> expected_frequencies = background[:, None].dot(background[None, :])
>>> print(format(expected_frequencies, "%.4f"))
D   E   H   K   R
D 0.0406 0.0553 0.0197 0.0518 0.0342
E 0.0553 0.0752 0.0268 0.0705 0.0465
H 0.0197 0.0268 0.0095 0.0251 0.0166
K 0.0518 0.0705 0.0251 0.0660 0.0436
R 0.0342 0.0465 0.0166 0.0436 0.0288
```

Here, `background[:, None]` creates a 2D array consisting of a single column with the values of `expected_frequencies`, and `expected_frequencies[None, :]` a 2D array with these values as a single row. Taking their dot product (inner product) creates a matrix of expected frequencies where each entry consists of two `expected_frequencies` values multiplied with each other. For example, `expected_frequencies['D', 'E']` is equal to `residue_frequencies['D']` * `residue_frequencies['E']`.

We can now calculate the log-odds matrix by dividing the observed frequencies by the expected frequencies and taking the logarithm:

```python
>>> 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
```

164
This matrix can be used as the substitution matrix when performing alignments. For example,

```python
>>> 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
|--|---|---|---
|query| 0 | DHHKK | 5

>>> print("%.2f" % alignments.score)
-2.18

>>> score = (scoring_matrix["D", "D"]
+ scoring_matrix["E", "H"]
+ scoring_matrix["H", "H"]
+ scoring_matrix["E", "K"]
+ scoring_matrix["K", "K"])
>>> print("%.2f" % score)
-2.18
```

(see Chapter 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

```python
>>> from Bio.Align import substitution_matrices
>>> with open("hg38.chrom.sizes") as handle:
... table = substitution_matrices.read(handle)
...```

```python
>>> print(table)
|chr1 | 248956422.0
|chr2 | 242193529.0
|chr3 | 198295559.0
|chr4 | 190214555.0
|...  |
|chrUn_KI270385v1 | 990.0
|chrUn_KI270423v1 | 981.0
|chrUn_KI270392v1 | 971.0
|chrUn_KI270394v1 | 970.0
```

165
chr4  190214555.0  
...  
chrUn_KI270423v1  981.0  
chrUn_KI270392v1  971.0  
chrUn_KI270394v1  970.0  

Use \texttt{dtype=int} to read the values as integers:

```python
>>> with open("hg38.chrom.sizes") as handle:
...     table = substitution_matrices.read(handle, int)
...```

```python
>>> print(table)  # doctest: +ELLIPSIS
chr1 248956422  
chr2 242193529  
chr3 198295559  
chr4 190214555  
...  
chrUn_KI270423v1  981  
chrUn_KI270392v1  971  
chrUn_KI270394v1  970  
```

For two-dimensional arrays, we follow the file format of substitution matrices provided by NCBI. For example, the BLOSUM62 matrix, which is the default substitution matrix for NCBI's protein-protein BLAST [1] program \texttt{blastp}, is stored as follows:

```plaintext
# 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 -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 -1 -1 -3 -1 -2 -3 -1 -1 -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 -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 \texttt{Bio/Align/substitution_matrices/data}. To parse this file, use:

```python
>>> from Bio.Align import substitution_matrices
>>> with open("BLOSUM62") as handle:
...     matrix = substitution_matrices.read(handle)
...```

```python
>>> print(matrix.alphabet)
166
```
>>> 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 -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
'ARNDCQEGHILMFPSV*'

To get a full list of available substitution matrices, use load without an argument:
Note that the substitution matrix provided by Schneider et al. [40] uses an alphabet consisting of three-nucleotide codons:

```python
>>> 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:

```python
>>> 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

```python
>>> alignments = aligner.align(seq1.seq, seq2.seq)
```

In this example, the total number of optimal alignments is huge (more than $4 \times 10^{37}$), and calling `len(alignments)` will raise an `OverflowError`:

```python
>>> 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:

```python
>>> alignment = alignments[0]
```

The alignment object stores the alignment score, as well as the alignment itself:

```python
>>> print(alignment.score)
72.0
>>> print(alignment)
```

```text
<table>
<thead>
<tr>
<th>target</th>
<th>query</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```text
<table>
<thead>
<tr>
<th>target</th>
<th>query</th>
</tr>
</thead>
<tbody>
<tr>
<td>TY--FPHF-----DLSKGS----AQVK-G------HGKKV--A--DA-LTNAVAHV-DDMPN</td>
<td>--QRF--FESFGDLS----TPDA-V--MGNPKVKAHGKVVLGAFA--G--A--H--LD--N</td>
</tr>
<tr>
<td>41</td>
<td>39</td>
</tr>
<tr>
<td>60</td>
<td>29</td>
</tr>
</tbody>
</table>
```

168
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:

```python
>>> 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])
```

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':

```python
>>> aligner.mode = "local"
>>> aligner.open_gap_score = -10
>>> aligner.extend_gap_score = -1
>>> alignments = aligner.align("LSPADKTNVKA", "PEEKSV")
```
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:

```python
>>> from Bio.Align import substitution_matrices
>>> m = substitution_matrices.load("SCHNEIDER")
>>> m.alphabet
('AAA', 'AAC', 'AAG', 'AAT', 'ACA', 'ACC', 'ACG', 'ACT', ..., 'TTG', 'TTT')
```

We can use this matrix to align codon sequences to each other:

```python
>>> 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:
would get a much lower score:

```python
>>> 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:

```python
>>> 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
```

### 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:

```python
>>> 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:

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

We use +6/-1 match and mismatch scores as an approximation of the BLOSUM62 matrix, and align these sequences to each other:

```python
>>> aligner.match = +6
>>> aligner.mismatch = -1
>>> alignments = aligner.align(s1, s2)
>>> len(alignments)
2
```
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:

```python
>>> 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:

```python
>>> aligner.wildcard = "?"
>>> ord(aligner.wildcard)
63
>>> s2 = np.array([2, 63, 13], np.int32)
>>> aligner.gap_score = -3
```

172
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:

```python
>>> 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.

<table>
<thead>
<tr>
<th>Meta-attribute</th>
<th>Attributes it maps to</th>
</tr>
</thead>
<tbody>
<tr>
<td>frameshift_minus_score</td>
<td>frameshift_minus_two_score, frameshift_minus_one_score</td>
</tr>
<tr>
<td>frameshift_plus_score</td>
<td>frameshift_plus_two_score, frameshift_plus_one_score</td>
</tr>
<tr>
<td>frameshift_two_score</td>
<td>frameshift_minus_two_score, frameshift_plus_two_score</td>
</tr>
<tr>
<td>frameshift_one_score</td>
<td>frameshift_minus_one_score, frameshift_plus_one_score</td>
</tr>
<tr>
<td>frameshift_score</td>
<td>frameshift_minus_two_score, frameshift_minus_one_score, frameshift_plus_one_score, frameshift_plus_two_score</td>
</tr>
</tbody>
</table>

>>> 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:

```python
>>> from Bio import Align
>>> aligner = Align.CodonAligner()
```

The CodonAligner object `aligner` stores the alignment parameters to be used for the alignments:

```python
>>> print(aligner)
Codon aligner with parameters
    wildcard: 'X'
    match_score: 1.0
    mismatch_score: 0.0
    frameshift_minus_two_score: -3.0
    frameshift_minus_one_score: -3.0
    frameshift_plus_one_score: -3.0
    frameshift_plus_two_score: -3.0
```

The wildcard, match_score, and mismatch_score parameters are defined in the same was as for the PairwiseAligner class described above (see Section 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:

```python
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> nuc1 = Seq("TCAGGGACTGGAGAACCAAGCTACTGCTGCTGCTGGCTGCGCTCTGCGCCGCAGGTGGGGCGCTGGAG")
>>> rna1 = SeqRecord(nuc1, id="rna1")
>>> nuc2 = Seq("TCAGGGACTTCGAGAACCAAGCGCTCCTGCTGCTGGCTGCGCTCGGCGCCGCAGGTGGAGCACTGGAG")
```

174
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 consists of only 68 nucleotides:

```python
>>> 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:

```python
>>> 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 TCAGGGACTGCGAGAACCAAGCTACTGCTGCTGCTGCTGCGCTCGGCAGGTGGG
<BLANKLINE>
pro1  20 A L E 23
rna1  60 GCGCTGGAG 69
<BLANKLINE>
>>> alignment1.coordinates
array([[ 0, 23],
       [ 0, 69]])
```

```python
>>> alignment1[0]
'SGTARTKLLLLLAALCAAGGALE'
>>> alignment1[1]
'TCAGGGACTGCGAGAACCAAGCTACTGCTGCTGCTGCTGCGCTCGGCAGGTGGGCTGCGCTGAG'
```

```python
>>> 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 TCAGGGACTTGGACGAAACCAGCTACTGCTGCTGCTGCTGCTGCTGCGCCAGGTGGGCCAGGTGGAG
<BLANKLINE>
pro2  8 L L L L A A L G A A G G A L E 23
rna2  23 CTCCTGCTGCTGCTGCTGCTGCTGCTGCTGCTGCTGCTGCTGCTGCTGCTGCTGCTGCTGCTGAG
<BLANKLINE>
>>> alignment2[0]
'SGTSRTKLLLLLAALGAAGGALE'
>>> alignment2[1]
```

175
'TCAGGGACTTCGAGAACCAAGCGCCTCCTGCTGCTGGCTGCGCTCGGCGCCGCAGGTGGAGCACTGGAG'

>>> alignment2.coordinates
array([[ 0,  8,  8, 23],
       [ 109, 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])
'SGTSRTRLLLAALGAAGGALE'
>>> _ == 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("ATGGGAACCGCGCTTCTTGGCTACTGGCCGCGCTCTGGCGCCGAGGTGGGGCCCCTGGAG")
>>> 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:
The three amino acid sequences can be aligned to each other, for example using ClustalW. Here, we create the alignment by hand:

```python
>>> 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 SGTSRTKLLLLLAALGAAGGALE 23
pro3 0 MGTA---LLLLLAALCAAGGALE 20
```

Now we can use the `mapall` method on the protein alignment, with the nucleotide-to-protein pairwise alignments as the argument, to obtain the corresponding codon alignment:

```python
>>> codon_alignment = protein_alignment.mapall([alignment1, alignment2, alignment3])
>>> print(codon_alignment)
```

### 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) [50].

● **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:

```python
>>> from Bio.Align import analysis
>>> pairwise_codon_alignment = codon_alignment[:2]
>>> print(pairwise_codon_alignment)
rna1 0 TCAGGACTGCGAGACAACTGTA 24
0 ||||||||..||||||..||..||.||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..||..|
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:

```python
>>> 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=0.030102..., name='rna3')
      Clade(branch_length=0.030102..., 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.

```python
>>> 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) ...
```

179
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| D.melanogaster (strain Ja-F) ...

gi|156863|gb|M19547.1| D.melanogaster (strain Af-S) ...

gi|156877|gb|M17836.1| D.melanogaster (strain Af-F) ...

gi|156875|gb|M17835.1| D.melanogaster (strain Wa-F) ...

gi|156873|gb|M17834.1| D.melanogaster (strain Fr-F) ...

gi|156871|gb|M17833.1| D.melanogaster (strain Fl-F) ...

gi|156869|gb|M17832.1| D.melanogaster (strain Ja-S) ...

gi|156867|gb|M17831.1| D.melanogaster (strain Fl-2S) ...

gi|156865|gb|M17830.1| D.melanogaster (strain Fr-S) ...

gi|156861|gb|M17828.1| D.melanogaster (strain Fl-1S) ...

gi|156859|gb|M17827.1| D.melanogaster (strain Wa-S) ...

>>> protein_alignment = Align.read("adh.aln", "clustal")

>>> len(protein_alignment)
27

>>> print(protein_alignment) # doctest: +ELLIPSIS

0 MAFTLTNKNVVFVAGLGGIGLDTSKELVKRDLKNLVILDRIENPAAIAELKAINPKVTVT
0 MAFTLTNKNVVFVAGLGGIGLDTSKELVKRDLKNLVILDRIENPAAIAELKAINPKVTVT
0 MAFTLTNKNVVFVAGLGGIGLDTSKELVKRDLKNLVILDRIENPAAIAELKAINPKVTVT

... 0 MSFTLTKNVFVAGLGGIGLDTSKELRKDLKNVLIDRIENPAAIAELKAINPKVTVT

<BLANKLINE>

...<BLANKLINE>

0 GLEAIQWSKHWSGI 256
0 GLEAIQWSKHWSGI 256
0 GLEAIQWSKHWSGI 256

...<BLANKLINE>

0 GLEAIQWTKHWSGI 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

0 M S F T L T N K N V I F V A G L G GIG
0 ATGTCTGTATCTTTGACACAAACAGAGCTGTATTTCGGTCGTCGCCGTTCTGGAGGCACTTGT

<BLANKLINE>

180
In addition to the multiple codon alignment, the function \texttt{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 \texttt{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; 1if1 ; 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 AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVAGLWIRLFFKSSKA
#=GR COATB_BPIKE/30-81 SS -HHHHHHHHHHHHHH--HHHHHHHH--HHHHHHHHHHHHHHHHHHH----
Q9T0Q8_BPIKE/1-52 AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVAGLWIRLFFKSSKA
COATB_BPI22/32-83 DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLWIRLFFKSSKA
COATB_BPM13/24-72 AEGDDP...AKAANFLQASATEYIGAYAMVVIVVIGATIGKLFKFTSKA
#=GR COATB_BPM13/24-72 SS ---S-T...CHHHHCCCTCTCTCTHHHHHHHHHHHHHHHHHHCTTT--
COATB_BPZJ2/1-49 AEGDDP...AKAANFLQASATEYIGAYAMVVIVVIGATIGKLFKFTSKA
Q9T0Q9_BPFD/1-49 AEGDDP...AKAANFLQASATEYIGAYAMVVIVVIGATIGKLFKFTSKA
#=GR Q9T0Q9_BPFD/1-49 SS ---------------HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH--
COATB_BPIF1/22-73 FAADDATSQAKAANFLQASATEYIGAYAMVVIVVIGATIGKLFKFTSKA
#=GR COATB_BPIF1/22-73 SS XX-HHHH--HHHHHH--HHHHHHHHHHHHHHHHHHHHHHHHHHH--
#=GC SS_cons XHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH--
#=GC seq_cons AE吞噬...AptAhDSLpaspAT-hIu.sWhsVsslVsAsluIJKLFKFsK
```

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):

```python
>>> from Bio import AlignIO

>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
```

This code will print out a summary of the alignment:

```python
>>> print(alignment)
Alignment with 7 rows and 52 columns
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVAGLWIRL...SKA COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVAGLWIRL...SKA Q9T0Q8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLWIRL...SKA COATB_BPI22/32-83
AEGDDP...AKAANFLQASATEYIGAYAMVVIVVIGATIGKLFKFTSKA COATB_BPM13/24-72
AEGDDP...AKAANFLQASATEYIGAYAMVVIVVIGATIGKLFKFTSKA COATB_BPZJ2/1-49
AEGDDP...AKAANFLQASATEYIGAYAMVVIVVIGATIGKLFKFTSKA Q9T0Q9_BPFD/1-49
FAADDATSQAKAANFLQASATEYIGAYAMVVIVVIGATIGKLFKFTSKA 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))
...
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVAVGLVIRLKFKFSKA - COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVAVGLKLFKFKFSRA - Q9T0Q8_BPIKE/1-52
DGTSTATSYATEAMSLSLKTQATDLDIQTWPVVTSSAVAGLAIRLKFKFSKA - COATB_BPI22/32-83
AEGDDP---AKAASFSLQASATEYIGYAWVVMVIVGATIGIKLFKKFTSKA - COATB_BPM13/24-72
AEGDDP---AKAASFSLQASATEYIGYAWVVMVIVGATIGIKLFKKFASKA - COATB_BPZJ2/1-49
AEGDDP---AKAASFSLQASATEYIGYAWVVMVIVGATIGIKLFKKFTSKA - Q9T0Q9_BPFD/1-49
FAADDATSQAKAASFSLTAQATEMSFGYAWALVVLVVGATVIGIKLFKFSRA - 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
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVAVGLVIRLKFKFSKA
>Q9T0Q8_BPIKE/1-52
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVAVGLKLFKFKFSRA
>COATB_BPI22/32-83
DGTSTATSYATEAMSLSLKTQATDLDIQTWPVVTSSAVAGLAIRLKFKFSKA
>COATB_BPM13/24-72
AEGDDP---AKAASFSLQASATEYIGYAWVVMVIVGATIGIKLFKKFTSKA
>COATB_BPZJ2/1-49
AEGDDP---AKAASFSLQASATEYIGYAWVVMVIVGATIGIKLFKKFASKA
>Q9T0Q9_BPFD/1-49
AEGDDP---AKAASFSLQASATEYIGYAWVVMVIVGATIGIKLFKKFTSKA
>COATB_BPIF1/22-73
FAADDATSQAKAASFSLTAQATEMSFGYAWALVVLVVGATVIGIKLFKFSRA

184
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:

```python
>>> 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):

```python
>>> 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 AAACAAC
Beta AACC CCC
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 CCAACC
Epsilon CCAAAC
5 6
Alpha AAACCA
Beta AAACCC
Gamma ACCCAA
Delta CCAACC
Epsilon CCAAAC
5 6
Alpha AAACCA
Beta AAACCC
Gamma ACCCAA
Delta CCAACC
Epsilon CCAAAC
5 6
Alpha AAAAAC
```

185
If you wanted to read this in using Bio.AlignIO you could use:

```python
>>> 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
CCCAAC Epsilon

Alignment with 5 rows and 6 columns
AAACCA Alpha
AAACCC Beta
AACAAC Gamma
CCCCCA Delta
CCCAAC Epsilon

Alignment with 5 rows and 6 columns
AAAACC Alpha
AAACCC Beta
AAAACC Gamma
CCCCAA Delta
CAAAAC 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:

```python
>>> 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:

```plaintext
>Alpha
ACTACGACTAGCTCAG--G
>Beta
ACTACGCTAGCTCAGAAG
>Gamma
ACTACGGCTAGCACAGAAG
>Alpha
ACTACGACTAGCTCAGG--
>Beta
ACTACGCTAGCTCAGAAG
>Gamma
ACTACGGCTAGCACAGAAG
>Alpha
ACTACGACTAGCTCAGG--
>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?

```plaintext
>Alpha
ACTACGACTAGCTCAG--G
>Beta
ACTACGCTAGCTCAGAAG
>Alpha
ACTACGACTAGCTCAGG--
>Gamma
ACTACGCTAGCAGCAAG
>Alpha
ACTACGACTAGCTCAGG--
>Delta
ACTACGCTAGCAGCAAG
```

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:

```plaintext
>Alpha
ACTACGACTAGCTCAG--G
```
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:

```python
>>> 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
ACTACGACTAGCTCAGG - Alpha
ACTACCCTAGCTCAGGAAG - XXX

Alignment length 17
ACTACGACTAGCTCAGG - Alpha
ACTACGGCAAGCACAGG - YYY

Alignment length 21
--ACTACGAC--TAGCTCAGG - Alpha
GACTACGACAATTAGCTCAGG - 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.

```python
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Align import MultipleSeqAlignment

>>> align1 = MultipleSeqAlignment(  
...     [  
...         SeqRecord(Seq("ACTGCTAGCTAG"), id="Alpha"),  
...         SeqRecord(Seq("ACTGCTAGCTAG"), id="Beta"),  
...         SeqRecord(Seq("ACTGCTAGCTAG"), 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("ACTAGTACAGCTG"), 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:

```python
>>> 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   ACTGCTAGCT AG
Gamma  ACTGCTAGCT AG
3 9
Delta  GTCAGC-AG
```

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:

```python
>>> 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()`:

```python
>>> 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:

```python
>>> 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
```
Alternatively, you could make a PHYLIP format file which we’ll name “PF05371_seed.phy”:

```python
>>> 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 VIRLFFKSFSS
Q9T0Q8_BPIKE/1-52 AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIRLFFKSFSS
COATB_BPI22/32-83 DGTSTATSYA TEAMNSLKTQ AIDLISQTWP VVTTVVVAGL AIRLFFKSFSS
COATB_BPM13/24-72 AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKFTS
COATB_BPZJ2/1-49 AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKFTS
Q9T0Q9_BPF/1-49 AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKFTS
COATB_BPIF1/22-73 FAADDATSQA KAAFDSLTAQATEMSGYAWALVVNLVVGATVGIKLFFKFS
```

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:

```python
>>> 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_BPIK AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIRLFFKSFSS
Q9T0Q8_BPIKE/1-52 AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIRLFFKSFSS
COATB_BPI22/32-83 DGTSTATSYA TEAMNSLKTQ AIDLISQTWP VVTTVVVAGL AIRLFFKSFSS
COATB_BPM13/24-72 AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKFTS
COATB_BPZJ2/1-49 AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKFTS
Q9T0Q9_BPF/1-49 AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKFTS
COATB_BPIF1/22-73 FAADDATSQA KAAFDSLTAQATEMSGYAWALVVNLVVGATVGIKLFFKFS
```

191
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:

```python
>>> 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
...
>>> 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:

```python
{
    0: "COATB_BPIKE/30-81",
    1: "Q9T0Q8_BPIKE/1-52",
    2: "Q9T0Q9_BPFD/1-49",
    3: "Q9T0Q7_BPM13/24-72",
    4: "Q9T0Q2_BPFZJ2/1-49",
    5: "Q9T0Q9_BPFD/1-49",
    6: "Q9T0Q8_BPIKE/30-81",
}
```

Here is the new (strict) PHYLIP format output:

```
7 52
seq0  AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIKLFKKFSS
seq1  AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIKLFKKFSS
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 LVVVLVVGATV GIKLFKKFVS
```

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:

```python
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> print(format(alignment, "clustal"))
CLUSTAL X (1.81) multiple sequence alignment
```

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:

```python
>>> 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
```
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:

```python
>>> 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))
... AEPNAATNYATEAMDSLQTWPVYTVVAVGLVIRLKFKKFSSKA - COATB_BPIKE/30-81
AEPNAATNYATEAMDSLQTWPVYTVVAVGLVIRLKFKKFSSKA - COATB_BPIKE/30-81
AEPNAATNYATEAMDSLQTWPVYTVVAVGLVIRLKFKKFSSKA - COATB_BPIKE/30-81
```

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:

```python
>>> print(alignment)
Alignment with 7 rows and 52 columns
AEPNAATNYATEAMDSLQTWPVYTVVAVGLVIRLKFKKFSSKA - COATB_BPIKE/30-81
AEPNAATNYATEAMDSLQTWPVYTVVAVGLVIRLKFKKFSSKA - COATB_BPIKE/30-81
AEPNAATNYATEAMDSLQTWPVYTVVAVGLVIRLKFKKFSSKA - COATB_BPIKE/30-81
```

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.

```python
>>> print(alignment[2, 6])
T
```

Using two integer indices pulls out a single letter, short hand for this:
You can pull out a single column as a string like this:

```python
>>> print(alignment[2].seq[6])
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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> print(alignment[:, 9:])
Alignment with 7 rows and 43 columns
ATEAMDSLKTQAIIDLISQTWVPTTVVAGLVIRLKFKKFSSKA COATB_BPIKE/30-81
ATEAMDSLKTQAIIDLISQTWVPTTVVAGLVIKLKFKVSVRA Q9T0Q8_BPIKE/1-52
ATEAMNLSTLQATDLDQTWVPTVAVAGLAIRLKFKKFSSKA COATB_BPI22/32-83
AKAADFNSLGQASATEYIGYAWAVNVIAGTVIGKLFKKFSSKA COATB_BPM13/24-72
AKAADFSDLSQASATEYIGYAWAVNVIAGTVIGKLFKKFSSKA Q9T0Q9_BPFD/1-49
AKAADFSDLSQASATEYIGYAWAVNVIAGTVIGKLFKKFSSKA 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:

```python
>>> edited = alignment[:, :6] + alignment[:, 9:]
>>> print(edited)
```

This modification results in the following output:

```
Alignment with 7 rows and 49 columns
AEPNAAATEAMDSLKTQAIDLISQTWPVVTTVVAVGLVIRLFKKFSSKA COATB_BPIKE/30-81
AEPNAAATEAMDSLKTQAIDLISQTWPVVTTVVAVGLVIRLFKKFSSKA Q9T0Q8_BPIKE/1-52
DGTSTAATEAMDSLKTQAIDLISQTWPVVTTVVAVGLVIRLFKKFSSKA CDATB_BPI22/32-83
AEGDPAKAADFSLQASATEYIGYAWAMVVVIVGATIGIKLFFKFTSKA COATB_BPM13/24-72
AEGDPAKAADFSLQASATEYIGYAWAMVVVIVGATIGIKLFFKFTSKA COATB_BPM13/24-72
AEGDPAKAADFSLQASATEYIGYAWAMVVVIVGATIGIKLFFKFTSKA CDATB_BPFJ1/22-73
FAADDAAKAADFSLQASATEYIGYAWAMVVVIVGATIGIKLFFKFTSKA Q9T0Q9_BPFD/1-49
```

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:

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

This results in:

```
Alignment with 7 rows and 49 columns
DGTSTAATEAMDSLKTQAIDLISQTWPVVTTVVAVGLVIRLFKKFSSKA COATB_BPI22/32-83
FAADDAAKAADFSLQASATEYIGYAWAMVVVIVGATIGIKLFFKFTSKA CDATB_BPFJ1/22-73
AEPNAAATEAMDSLKTQAIDLISQTWPVVTTVVAVGLVIRLFKKFSSKA Q9T0Q8_BPIKE/1-52
AEGDPAKAADFSLQASATEYIGYAWAMVVVIVGATIGIKLFFKFTSKA COATB_BPM13/24-72
AEGDPAKAADFSLQASATEYIGYAWAMVVVIVGATIGIKLFFKFTSKA CDATB_BPFJ1/22-73
AEGDPAKAADFSLQASATEYIGYAWAMVVVIVGATIGIKLFFKFTSKA 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:

```python
>>> 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)
```

This gives:

```
Array shape 7 by 52
```

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,

```python
>>> 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"),
... ]
...
)
>>> 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

>>> print(msa)
Alignment with 4 rows and 7 columns
ACTCCTA seq1
AAT-CTA seq2
CCTACT- seq3
TCTCCTC seq4
```

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

```python
>>> m = substitutions.select("ATCG")
```

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:

```python
>>> 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:

```python
>>> 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):

```python
>>> alignment = msa.alignment
```

You can then create a `Motif` object (see section 17.1):

```python
>>> from Bio.motifs import Motif
>>> motif = Motif("ACGT", alignment)
```

and obtain a quick consensus sequence:

```python
>>> 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,

```python
>>> 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:

```python
>>> 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

```python
>>> my_pssm = summary_align.pos_specific_score_matrix(consensus, chars_to_ignore=['N'])
>>> print(my_pssm)

A C T
X 2.0 1.0 1.0
C 1.0 3.0 0.0
T 0.0 0.0 4.0
X 1.0 2.0 0.0
C 0.0 4.0 0.0
T 0.0 0.0 4.0
X 2.0 1.0 0.0
```

where we ignore any N ambiguity residues when calculating the PSSM.

Two notes should be made about this:

1. To maintain strictness with the alphabets, you can only include characters along the top of the PSSM that are in the alphabet of the alignment object. Gaps are not included along the top axis of the PSSM.

2. The sequence passed to be displayed along the left side of the axis does not need to be the consensus. For instance, if you wanted to display the second sequence in the alignment along this axis, you would need to do:

```python
>>> 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
```

199
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:

```python
>>> 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):

```python
>>> alignment = msa.alignment
```

You can then create a `Motif` object (see section 17.1):

```python
>>> from Bio.motifs import Motif
>>> motif = Motif("ACGT", alignment)
```

and obtain the counts of each nucleotide in each position:

```python
>>> 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
```

```python
>>> 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/](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_alignment` (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:

```python
>>> e_freq_table = {"A": 0.3, "C": 0.2, "T": 0.3, "G": 0.2}
>>> info_content = summary_alignment.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 returned 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:

```python
>>> info_content = summary_alignment.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 taken 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.

```python
>>> info_content = summary_alignment.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:

```python
>>> type(msa)
<class 'Bio.Align.MultipleSeqAlignment'>
>>> print(msa)
Alignment with 4 rows and 7 columns
| ACTCCTA seq1 |
| ACTCCTA seq1 |
| AAT-CTA seq2 |
| CCTACT- seq3 |
| TCTCCTC seq4 |

>>> alignment = msa.alignment
>>> type(alignment)
<class 'Bio.Align.Alignment'>
>>> print(alignment)
seq1 0 ACTCCTA 7
seq2 0 AAT-CTA 6
seq3 0 CCTACT- 6
seq4 0 TCTCCTC 7
```

Note that the alignment property creates and returns a new Alignment object that is consistent with the information stored in the MultipleSeqAlignment object at the time the Alignment object is created. Any changes to the MultipleSeqAlignment after calling the alignment property will not propagate to the Alignment object. However, you can of course call the alignment property again to create a new Alignment object consistent with the updated MultipleSeqAlignment object.

8.5 Calculating a substitution matrix from a multiple sequence alignment

You can create your own substitution matrix from an alignment. In this example, we’ll first read a protein sequence alignment from the Clustalw file protein.aln (also available online here)

```python
>>> 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")
```

```python
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
```
Rows and columns for other amino acids were removed from the matrix.

Next, we normalize the matrix:

```python
>>> import numpy as np
>>> observed_frequencies /= np.sum(observed_frequencies)
```

Summing over rows or columns gives the relative frequency of occurrence of each residue:

```python
>>> 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
```

The expected frequency of residue pairs is then

```python
>>> expected_frequencies = np.dot(
...     residue_frequencies[:, None], residue_frequencies[None, :]
... )
>>> print(expected_frequencies.format("%.4f"))
```

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:

```python
>>> m = np.log2( observed_frequencies / expected_frequencies)
>>> print(m)
```

This matrix can be used as the substitution matrix when performing alignments. For example,

```python
>>> 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:

```python
>>> 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’ve 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.

```python
>>> 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:

```python
>>> 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
TATACATAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.1|AF191
TATACATAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.1|AF191
TATACATAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.1|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:

```python
>>> 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
| |  |
| |  |
```

205
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`:

```python
>>> 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"`:

```python
>>> from Bio import AlignIO
>>> align = AlignIO.read("opuntia.txt", "fasta")
>>> print(align)
```

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:

```plaintext
>HBA_HUMAN
MVLSRAPDKTNVKAAGWAKVGAGAH.isConnected(a)
```

in a file `alpha.faa`, and secondly in a file `beta.faa`:
MVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFSESFGDLSTPDAVMGNPKV
VAHGKKVLGAFSGDLAHLDNLKGTFLATLSELHCDKLHVDPENFRLLGNVLCVLAHHFG
KEFTPPVQAAYQKVAVGANALAHKYH

You can find copies of these example files with the Biopython source code under the Doc/examples/ directory.

The command to align these two sequences against each other using **needle** is as follows:

```
needle -outfile=needle.txt -asequence=alpha.faa -bsequence=beta.faa -gapopen=10 -gapextend=0.5
```

Why not try running this by hand at the command prompt? You should see it does a pairwise comparison and records the output in the file *needle.txt* (in the default EMBOSS alignment file format).

Even if you have EMBOSS installed, running this command may not work – you might get a message about “command not found” (especially on Windows). This probably means that the EMBOSS tools are not on your PATH environment variable. You can either update your PATH setting, or simply use the full path to the tool, for example:

```
C:\EMBOSS\needle.exe -outfile=needle.txt -asequence=alpha.faa -bsequence=beta.faa -gapopen=10 -gapextend=0.5
```

Next we want to use Python to run this command for us. As explained above, for full control, we recommend you use Python’s built-in **subprocess** module:

```
>>> import sys
>>> import subprocess
>>> cmd = "needle -outfile=needle.txt -asequence=alpha.faa -bsequence=beta.faa -gapopen=10 -gapextend=0.5"
>>> results = subprocess.run(
...   cmd,
...   stdout=subprocess.PIPE,
...   stderr=subprocess.PIPE,
...   text=True,
...   shell=(sys, platform != "win32"),
... )
>>> print(results.stdout)
>>> print(results.stderr)
Needleman-Wunsch global alignment of two sequences
```

Next we can load the output file with **Bio.AlignIO** as discussed earlier in this chapter, as the **emboss** format:

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("needle.txt", "emboss")
>>> print(align)
```

Alignment with 2 rows and 149 columns
MV-LSPADKTNVKAAWGVAGFAGEYGAELERMFLSFPTTCTY...Kyr HBA_HUMAN
MVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFSESFGDLSTPDAVMG...

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,
...   text=True,
...   shell=(sys, platform != "win32"),
... )
```

207
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:

```python
>>> 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 n 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:

```python
>>> len(alignments)
80
>>> print(alignments[0])  # doctest:+ELLIPSIS
Alignment(seqA='MV-LSPADKTNV---K-A--A-WGKVGAHAG---EY-GA-EALE-RMFLSF----PTTK-TY--F...YR-', seqB='MVHL-----T--PEEKSAVTALWGKV----...Y-H', score=72.0, start=0, end=217)
```

Each alignment is a named tuple consisting of the two aligned sequences, the score, the start and the end positions of the alignment (in global alignments the start is always 0 and the end the length of the alignment). Bio.pairwise2 has a function format_alignment for a nicer printout:

```python
>>> print(pairwise2.format_alignment(*alignments[0]))  # doctest:+ELLIPSIS
MV-LSPADKTNV----K-A--A-WGKVGAHAG---EY-GA-EALE-RMFLSF----PTTK-TY--F...YR-
```
Since Biopython 1.77 the required parameters can be supplied with keywords. The last example can now also be written as:

```python
>>> 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`):

```python
>>> 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...KRYR
MVHLTPEEKSAVTLWGKV-NVDEVGGEALGRLLVVYPWTQRFF...KHY
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:

```python
>>> from Bio import pairwise2
>>> from Bio.Align import substitution_matrices
>>> blosum62 = substitution_matrices.load("BLOSUM62")
>>> alignments = pairwise2.align.localds("LSPADKTNVKAAGKVGAHAGEYGAEALERMFLSFPTTKTY...KRYR
MVHLTPEEKSAVTLWKGVT-NVDEVGGEALGRLLVVYPWTQRFF...KHY
Score=292.5
```

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`:

```python
>>> from Bio import pairwise2
>>> from Bio.Align import substitution_matrices
```

210
>>> 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

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
 |--PEEKSAV---
     Score=16

Instead of supplying a complete match/mismatch matrix, the match code m allows for easy defining general match/mismatch values. The next example uses match/mismatch scores of 5/-4 and gap penalties (open/extend) of 2/0.5 using locals:

>>> alignments = pairwise2.align.localms("AGAACT", "GAC", 5, -4, -2, -0.5)
>>> print(pairwise2.format_alignment(*alignments[0]))
2 GAAC
  | ||
1 G-AC
  Score=13

One useful keyword argument of the Bio.pairwise2.align functions is score_only. When set to True it will only return the score of the best alignment(s), but in a significantly shorter time. It will also allow the alignment of longer sequences before a memory error is raised. Another useful keyword argument is one_alignment_only=True which will also result in some speed gain.

Unfortunately, Bio.pairwise2 does not work with Biopython’s multiple sequence alignment objects (yet). However, the module has some interesting advanced features: you can define your own match and gap functions (interested in testing affine logarithmic gap costs?), gap penalties and end gaps penalties can be different for both sequences, sequences can be supplied as lists (useful if you have residues that are encoded by more than one character), etc. These features are hard (if at all) to realize with other alignment tools. For more details see the 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 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.

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.

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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):

```python
>>> 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.

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.

```python
>>> with open("my_blast.xml", "w") as out_handle:
...     out_handle.write(result_handle.read())
...     out_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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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):

```python
>>> from Bio.Blast import NCBIXML
>>> blast_record = NCBIXML.read(result_handle)
```

or, if you have lots of results (i.e., multiple query sequences):

```python
>>> 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:

```python
>>> 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:

```python
>>> E_VALUE_THRESH = 0.04

>>> for alignment in blast_record.alignments:
...     for hsp in alignment.hsps:
...         if hsp.expect < E_VALUE_THRESH:
```
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... |
| | | | | | | | | | | | | | | |
| tactgtttgttggatccaaactggagaacctgcgtcactcttttagtcccttacatattcctc... |

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 PSI blast 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 PSI blast record classes. The PSI blast record object is similar, but has support for the rounds that are used in the iteration steps of PSI blast.

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
CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTAGAGGG
```

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 to 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:

```python
>>> 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
```

222
<table>
<thead>
<tr>
<th>#</th>
<th>gi</th>
<th>ref</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>gi</td>
<td>262205317</td>
<td>ref</td>
</tr>
<tr>
<td>1</td>
<td>gi</td>
<td>301171311</td>
<td>ref</td>
</tr>
<tr>
<td>2</td>
<td>gi</td>
<td>270133242</td>
<td>ref</td>
</tr>
<tr>
<td>3</td>
<td>gi</td>
<td>301171322</td>
<td>ref</td>
</tr>
<tr>
<td>4</td>
<td>gi</td>
<td>301171267</td>
<td>ref</td>
</tr>
<tr>
<td>5</td>
<td>gi</td>
<td>262205302</td>
<td>ref</td>
</tr>
<tr>
<td>6</td>
<td>gi</td>
<td>301171259</td>
<td>ref</td>
</tr>
<tr>
<td>7</td>
<td>gi</td>
<td>262205451</td>
<td>ref</td>
</tr>
<tr>
<td>8</td>
<td>gi</td>
<td>301171447</td>
<td>ref</td>
</tr>
<tr>
<td>9</td>
<td>gi</td>
<td>301171276</td>
<td>ref</td>
</tr>
<tr>
<td>10</td>
<td>gi</td>
<td>262205290</td>
<td>ref</td>
</tr>
<tr>
<td>11</td>
<td>gi</td>
<td>301171354</td>
<td>ref</td>
</tr>
<tr>
<td>12</td>
<td>gi</td>
<td>301171254</td>
<td>ref</td>
</tr>
<tr>
<td>13</td>
<td>gi</td>
<td>262205298</td>
<td>ref</td>
</tr>
<tr>
<td>14</td>
<td>gi</td>
<td>301171394</td>
<td>ref</td>
</tr>
<tr>
<td>15</td>
<td>gi</td>
<td>262205429</td>
<td>ref</td>
</tr>
<tr>
<td>16</td>
<td>gi</td>
<td>262205423</td>
<td>ref</td>
</tr>
<tr>
<td>17</td>
<td>gi</td>
<td>301171401</td>
<td>ref</td>
</tr>
<tr>
<td>18</td>
<td>gi</td>
<td>270133247</td>
<td>ref</td>
</tr>
<tr>
<td>19</td>
<td>gi</td>
<td>262205309</td>
<td>ref</td>
</tr>
<tr>
<td>20</td>
<td>gi</td>
<td>270133217</td>
<td>ref</td>
</tr>
<tr>
<td>21</td>
<td>gi</td>
<td>301171437</td>
<td>ref</td>
</tr>
<tr>
<td>22</td>
<td>gi</td>
<td>270133306</td>
<td>ref</td>
</tr>
<tr>
<td>23</td>
<td>gi</td>
<td>301171428</td>
<td>ref</td>
</tr>
<tr>
<td>24</td>
<td>gi</td>
<td>301171211</td>
<td>ref</td>
</tr>
<tr>
<td>25</td>
<td>gi</td>
<td>301171153</td>
<td>ref</td>
</tr>
<tr>
<td>26</td>
<td>gi</td>
<td>301171146</td>
<td>ref</td>
</tr>
<tr>
<td>27</td>
<td>gi</td>
<td>270133254</td>
<td>ref</td>
</tr>
<tr>
<td>28</td>
<td>gi</td>
<td>262205445</td>
<td>ref</td>
</tr>
<tr>
<td>97</td>
<td>gi</td>
<td>356517317</td>
<td>ref</td>
</tr>
<tr>
<td>98</td>
<td>gi</td>
<td>297814701</td>
<td>ref</td>
</tr>
<tr>
<td>99</td>
<td>gi</td>
<td>397513516</td>
<td>ref</td>
</tr>
</tbody>
</table>

We’ve just begun to scratch the surface of the object model, but you can see that there’s already some useful information. By invoking `print` on the `QueryResult` object, you can see:

- The program name and version (blastn version 2.2.27+)
- The query ID, description, and its sequence length (ID is 42291, description is ‘mystery_seq’, and it is 61 nucleotides long)
- The target database to search against (refseq_rna)
- A quick overview of the resulting hits. For our query sequence, there are 100 potential hits (numbered 0–99 in the table). For each hit, we can also see how many HSPs it contains, its ID, and a snippet of its description. Notice here that `Bio.SearchIO` truncates the hit table overview, by showing only hits numbered 0–29, and then 97–99.

Now let’s check our BLAT results using the same procedure as above:
You’ll immediately notice that there are some differences. Some of these are caused by the way PSL format stores its details, as you’ll see. The rest are caused by the genuine program and target database differences between our BLAST and BLAT searches:

- The program name and version. Bio.SearchIO knows that the program is BLAT, but in the output file there is no information regarding the program version so it defaults to ‘¡unknown version¡’.

- The query ID, description, and its sequence length. Notice here that these details are slightly different from the ones we saw in BLAST. The ID is ‘mystery_seq’ instead of 42991, there is no known description, but the query length is still 61. This is actually a difference introduced by the file formats themselves. BLAST sometimes creates its own query IDs and uses your original ID as the sequence description.

- The target database is not known, as it is not stated in the BLAT output file.

- And finally, the list of hits we have is completely different. Here, we see that our query sequence only hits the ‘chr19’ database entry, but in it we see 17 HSP regions. This should not be surprising however, given that we are using a different program, each with its own target database.

All the details you saw when invoking the print method can be accessed individually using Python’s object attribute access notation (a.k.a. the dot notation). There are also other format-specific attributes that you can access using the same method.

For a complete list of accessible attributes, you can check each format-specific documentation. Here are the ones for BLAST and for BLAT.

Having looked at using print on QueryResult objects, let’s drill down deeper. What exactly is a QueryResult? In terms of Python objects, QueryResult is a hybrid between a list and a dictionary. In other words, it is a container object with all the convenient features of lists and dictionaries.

Like Python lists and dictionaries, QueryResult objects are iterable. Each iteration returns a Hit object:

```python
>>> for hit in blast_qresult:
...    print(hit)
...
Hit(id='gi|262205317|ref|NR_030195.1|', query_id='42291', 1 hsps)
Hit(id='gi|301171311|ref|NR_035856.1|', query_id='42291', 1 hsps)
Hit(id='gi|270133242|ref|NR_032573.1|', query_id='42291', 1 hsps)
Hit(id='gi|301171322|ref|NR_035857.1|', query_id='42291', 2 hsps)
Hit(id='gi|301171267|ref|NR_035851.1|', query_id='42291', 1 hsps)
...```
To check how many items (hits) a QueryResult has, you can simply invoke Python’s `len` method:

```python
>>> len(blast_qresult)
100
>>> len(blat_qresult)
1
```

Like Python lists, you can retrieve items (hits) from a QueryResult using the slice notation:

```python
>>> blast_qresult[0]  # retrieves the top hit
Hit(id='gi|262205317|ref|NR_030195.1|', query_id='42291', 1 hsps)
```

```python
>>> blast_qresult[-1]  # retrieves the last hit
Hit(id='gi|397513516|ref|XM_003827011.1|', query_id='42291', 1 hsps)
```

To retrieve multiple hits, you can slice QueryResult objects using the slice notation as well. In this case, the slice will return a new QueryResult object containing only the sliced hits:

```python
>>> blast_slice = blast_qresult[:3]  # slices the first three hits
```

```python
Program: blastn (2.2.27+)
Query: 42291 (61)
mystery_seq
Target: refseq_rna
```

```plaintext
# # HSP ID + description
---- ----- ----------------------------------------------------------
0 1 gi|262205317|ref|NR_030195.1| Homo sapiens microRNA 52...
1 1 gi|301171311|ref|NR_035856.1| Pan troglodytes microRNA...
2 1 gi|270133242|ref|NR_032573.1| Macaca mulatta microRNA ...
```

Like Python dictionaries, you can also retrieve hits using the hit’s ID. This is particularly useful if you know a given hit ID exists within a search query results:

```python
>>> blast_qresult["gi|262205317|ref|NR_030195.1"]
Hit(id='gi|262205317|ref|NR_030195.1|', query_id='42291', 1 hsps)
```

You can also get a full list of Hit objects using `hits` and a full list of Hit IDs using `hit_keys`:

```python
>>> blast_qresult.hits
[...]
# list of all hits
```

```python
>>> blast_qresult.hit_keys
[...]
# list of all hit IDs
```

What if you just want to check whether a particular hit is present in the query results? You can do a simple Python membership test using the `in` keyword:

```python
>>> "gi|262205317|ref|NR_030195.1|" in blast_qresult
True
```

```python
>>> "gi|262205317|ref|NR_030194.1|" in blast_qresult
False
```

Sometimes, knowing whether a hit is present is not enough; you also want to know the rank of the hit. Here, the `index` method comes to the rescue:
Remember that we’re using Python’s indexing style here, which is zero-based. This means our hit above is ranked at no. 23, not 22.

Also, note that the hit rank you see here is based on the native hit ordering present in the original search output file. Different search tools may order these hits based on different criteria.

If the native hit ordering doesn’t suit your taste, you can use the sort method of the QueryResult object. It is very similar to Python’s list.sort method, with the addition of an option to create a new sorted QueryResult object or not.

Here is an example of using QueryResult.sort to sort the hits based on each hit’s full sequence length. For this particular sort, we’ll set the in_place flag to False so that sorting will return a new QueryResult object and leave our initial object unsorted. We’ll also set the reverse flag to True so that we sort in descending order.

```python
>>> for hit in blast_qresult[:5]: # id and sequence length of the first five hits
...     print("%s %i" % (hit.id, hit.seq_len))
...
... gi|262205317|ref|NR_030195.1| 61
... gi|301171311|ref|NR_035856.1| 60
... gi|270133242|ref|NR_032573.1| 85
... gi|301171322|ref|NR_035857.1| 86
... gi|301171267|ref|NR_035851.1| 80

>>> sort_key = lambda hit: hit.seq_len
>>> sorted_qresult = blast_qresult.sort(key=sort_key, reverse=True, in_place=False)
>>> for hit in sorted_qresult[:5]:
...     print("%s %i" % (hit.id, hit.seq_len))
...
... gi|397513516|ref|XM_003827011.1| 6002
... gi|390332045|ref|XM_776818.2| 4082
... gi|390332043|ref|XM_003723358.1| 4079
... gi|356517317|ref|XM_003527287.1| 3251
... gi|356543101|ref|XM_003539954.1| 2936
```

The advantage of having the in_place flag here is that we’re preserving the native ordering, so we may use it again later. You should note that this is not the default behavior of QueryResult.sort, however, which is why we needed to set the in_place flag to True explicitly.

At this point, you’ve known enough about QueryResult objects to make it work for you. But before we go on to the next object in the Bio.SearchIO model, let’s take a look at two more sets of methods that could make it even easier to work with QueryResult objects: the filter and map methods.

If you’re familiar with Python’s list comprehensions, generator expressions or the built in filter and map functions, you’ll know how useful they are for working with list-like objects (if you’re not, check them out!). You can use these built in methods to manipulate QueryResult objects, but you’ll end up with regular Python lists and lose the ability to do more interesting manipulations.

That’s why, QueryResult objects provide its own flavor of filter and map methods. Analogous to filter, there are hit_filter and hsp_filter methods. As their name implies, these methods filter its QueryResult object either on its Hit objects or HSP objects. Similarly, analogous to map, QueryResult objects also provide the hit_map and hsp_map methods. These methods apply a given function to all hits or HSPs in a QueryResult object, respectively.

Let’s see these methods in action, beginning with hit_filter. This method accepts a callback function that checks whether a given Hit object passes the condition you set or not. In other words, the function must accept as its argument a single Hit object and returns True or False.
Here is an example of using hit_filter to filter out Hit objects that only have one HSP:
>>> filter_func = lambda hit: len(hit.hsps) > 1 # the callback function
>>> len(blast_qresult) # no. of hits before filtering
100
>>> filtered_qresult = blast_qresult.hit_filter(filter_func)
>>> len(filtered_qresult) # no. of hits after filtering
37
>>> for hit in filtered_qresult[:5]: # quick check for the hit lengths
...
print("%s %i" % (hit.id, len(hit.hsps)))
...
gi|301171322|ref|NR_035857.1| 2
gi|262205330|ref|NR_030198.1| 2
gi|301171447|ref|NR_035871.1| 2
gi|262205298|ref|NR_030190.1| 2
gi|270132717|ref|NR_032716.1| 2
hsp_filter works the same as hit_filter, only instead of looking at the Hit objects, it performs
filtering on the HSP objects in each hits.
As for the map methods, they too accept a callback function as their arguments. However, instead of
returning True or False, the callback function must return the modified Hit or HSP object (depending on
whether you’re using hit_map or hsp_map).
Let’s see an example where we’re using hit_map to rename the hit IDs:
>>> def map_func(hit):
...
# renames "gi|301171322|ref|NR_035857.1|" to "NR_035857.1"
...
hit.id = hit.id.split("|")[3]
...
return hit
...
>>> mapped_qresult = blast_qresult.hit_map(map_func)
>>> for hit in mapped_qresult[:5]:
...
print(hit.id)
...
NR_030195.1
NR_035856.1
NR_032573.1
NR_035857.1
NR_035851.1
Again, hsp_map works the same as hit_map, but on HSP objects instead of Hit objects.

11.1.2

Hit

Hit objects represent all query results from a single database entry. They are the second-level container in
the Bio.SearchIO object hierarchy. You’ve seen that they are contained by QueryResult objects, but they
themselves contain HSP objects.
Let’s see what they look like, beginning with our BLAST search:
>>> from Bio import SearchIO
>>> blast_qresult = SearchIO.read("my_blast.xml", "blast-xml")
>>> blast_hit = blast_qresult[3] # fourth hit from the query result
>>> print(blast_hit)
Query: 42291
227


You see that we've got the essentials covered here:

- The query ID and description is present. A hit is always tied to a query, so we want to keep track of the originating query as well. These values can be accessed from a hit using the `query_id` and `query_description` attributes.

- We also have the unique hit ID, description, and full sequence lengths. They can be accessed using `id`, `description`, and `seq_len`, respectively.

- Finally, there's a table containing quick information about the HSPs this hit contains. In each row, we've got the important HSP details listed: the HSP index, its e-value, its bit score, its span (the alignment length including gaps), its query coordinates, and its hit coordinates.

Now let's contrast this with the BLAT search. Remember that in the BLAT search we had one hit with 17 HSPs.
Here, we’ve got a similar level of detail as with the BLAST hit we saw earlier. There are some differences worth explaining, though:

- The e-value and bit score column values. As BLAT HSPs do not have e-values and bit scores, the display defaults to '?'.
- What about the span column? The span values is meant to display the complete alignment length, which consists of all residues and any gaps that may be present. The PSL format do not have this information readily available and Bio.SearchIO does not attempt to try guess what it is, so we get a ‘?’ similar to the e-value and bit score columns.

In terms of Python objects, Hit behaves almost the same as Python lists, but contain HSP objects exclusively. If you’re familiar with lists, you should encounter no difficulties working with the Hit object.

Just like Python lists, Hit objects are iterable, and each iteration returns one HSP object it contains:

```python
>>> for hsp in blast_hit:
...     hsp
...
HSP(hit_id='gi|301171322|ref|NR_035857.1|', query_id='42291', 1 fragments)
HSP(hit_id='gi|301171322|ref|NR_035857.1|', query_id='42291', 1 fragments)
```

You can invoke len on a Hit to see how many HSP objects it has:

```python
>>> len(blast_hit)
2
>>> len(blat_hit)
17
```

You can use the slice notation on Hit objects, whether to retrieve single HSP or multiple HSP objects. Like QueryResult, if you slice for multiple HSP, a new Hit object will be returned containing only the sliced HSP objects:

```python
>>> blat_hit[0]  # retrieve single items
HSP(hit_id='chr19', query_id='mystery_seq', 1 fragments)
>>> sliced_hit = blat_hit[4:9]  # retrieve multiple items
>>> len(sliced_hit)
5
>>> print(sliced_hit)
Query: mystery_seq
<unknown description>
Hit: chr19 (59128983)
<unknown description>
HSPs: # E-value Bit score Span Query range Hit range
---- ------- -------- --------- ----------------- ---------------------
0 ? ? ? [0:60] [54198476:54198536]
1 ? ? ? [0:61] [54265610:54265671]
2 ? ? ? [0:61] [54238143:54240175]
3 ? ? ? [0:60] [54189735:54189795]
4 ? ? ? [0:61] [54185425:54185486]
```

You can also sort the HSP inside a Hit, using the exact same arguments like the sort method you saw in the QueryResult object.

Finally, there are also the filter and map methods you can use on Hit objects. Unlike in the QueryResult object, Hit objects only have one variant of filter (Hit.filter) and one variant of map (Hit.map). Both of Hit.filter and Hit.map work on the HSP objects a Hit has.
11.1.3 HSP

HSP (high-scoring pair) represents region(s) in the hit sequence that contains significant alignment(s) to the query sequence. It contains the actual match between your query sequence and a database entry. As this match is determined by the sequence search tool’s algorithms, the HSP object contains the bulk of the statistics computed by the search tool. This also makes the distinction between HSP objects from different search tools more apparent compared to the differences you’ve seen in QueryResult or Hit objects.

Let’s see some examples from our BLAST and BLAT searches. We’ll look at the BLAST HSP first:

```python
>>> from Bio import SearchIO
>>> blast_qresult = SearchIO.read("my_blast.xml", "blast-xml")
>>> blast_hsp = blast_qresult[0][0]  # first hit, first hsp
>>> print(blast_hsp)

Query: 42291 mystery_seq
Hit: gi|262205317|ref|NR_030195.1| Homo sapiens microRNA 520b (MIR520...
Query range: [0:61] (1)
Hit range: [0:61] (1)
Quick stats: evalue 4.9e-23; bitscore 111.29
Fragments: 1 (61 columns)
Query - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
    |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Hit - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
```

Just like QueryResult and Hit, invoking print on an HSP shows its general details:

- There are the query and hit IDs and descriptions. We need these to identify our HSP.
- We’ve also got the matching range of the query and hit sequences. The slice notation we’re using here is an indication that the range is displayed using Python’s indexing style (zero-based, half open). The number inside the parenthesis denotes the strand. In this case, both sequences have the plus strand.
- Some quick statistics are available: the e-value and bitscore.
- There is information about the HSP fragments. Ignore this for now; it will be explained later on.
- And finally, we have the query and hit sequence alignment itself.

These details can be accessed on their own using the dot notation, just like in QueryResult and Hit:

```python
>>> blast_hsp.query_range
(0, 61)
```

```python
>>> blast_hsp.evalue
4.91307e-23
```

They’re not the only attributes available, though. HSP objects come with a default set of properties that makes it easy to probe their various details. Here are some examples:

```python
>>> blast_hsp.hit_start  # start coordinate of the hit sequence
0
>>> blast_hsp.query_span  # how many residues in the query sequence
61
>>> blast_hsp.aln_span  # how long the alignment is
61
```
Check out the HSP documentation for a full list of these predefined properties.

Furthermore, each sequence search tool usually computes its own statistics / details for its HSP objects. For example, an XML BLAST search also outputs the number of gaps and identical residues. These attributes can be accessed like so:

```python
>>> blast_hsp.gap_num  # number of gaps
0
>>> blast_hsp.ident_num  # number of identical residues
61
```

These details are format-specific; they may not be present in other formats. To see which details are available for a given sequence search tool, you should check the format's documentation in Bio.SearchIO. Alternatively, you may also use `__dict__.keys()` for a quick list of what’s available:

```python
>>> blast_hsp.__dict__.keys()
['bitscore', 'evalue', 'ident_num', 'gap_num', 'bitscore_raw', 'pos_num', '_items']
```

Finally, you may have noticed that the query and hit attributes of our HSP are not just regular strings:

```python
>>> blast_hsp.query
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG'), id='42291', name='aligned query sequence', description='mystery_seq', dbxrefs=[])  # object

>>> blast_hsp.hit
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG'), id='gi|262205317|ref|NR_030195.1|', name='aligned hit sequence', description='Homo sapiens microRNA 520b (MIR520B), microRNA', dbxrefs=[])  # object
```

They are `SeqRecord` objects you saw earlier in Section 4! This means that you can do all sorts of interesting things you can do with `SeqRecord` objects on `HSP.query` and/or `HSP.hit`.

It should not surprise you now that the HSP object has an `alignment` property which is a `MultipleSeqAlignment` object:

```python
>>> print(blast_hsp.aln)
Alignment with 2 rows and 61 columns
CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG 42291
CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG gi|262205317|ref|NR_030195.1|
```

Having probed the BLAST HSP, let’s now take a look at HSPs from our BLAT results for a different kind of HSP. As usual, we’ll begin by invoking `print` on it:

```python
>>> blat_qresult = SearchIO.read("my_blat.psl", "blat-psl")
>>> blat_hsp = blat_qresult[0][0]  # first hit, first hsp
>>> print(blat_hsp)
Query: mystery_seq <unknown description>
Hit: chr19 <unknown description>
Query range: [0:61] (1)
Hit range: [54204480:54204541] (1)
Quick stats: evalue ?; bitscore ?
Fragments: 1 (? columns)
```

Some of the outputs you may have already guessed. We have the query and hit IDs and descriptions and the sequence coordinates. Values for evalue and bitscore is ‘?’ as BLAT HSPs do not have these attributes. But The biggest difference here is that you don’t see any sequence alignments displayed. If you look closer, PSL formats themselves do not have any hit or query sequences, so Bio.SearchIO won’t create any sequence or alignment objects. What happens if you try to access `HSP.query`, `HSP.hit`, or `HSP.aln`? You’ll get the default values for these attributes, which is `None`.

231
>>> blat_hsp.hit is None
True

>>> blat_hsp.query is None
True

>>> blat_hsp.aln is None
True

This does not affect other attributes, though. For example, you can still access the length of the query or hit alignment. Despite not displaying any attributes, the PSL format still have this information so Bio.SearchIO can extract them:

>>> blat_hsp.query_span  # length of query match
61

>>> blat_hsp.hit_span    # length of hit match
61

Other format-specific attributes are still present as well:

>>> blat_hsp.score       # PSL score
61

>>> blat_hsp.mismatch_num # the mismatch column
0

So far so good? Things get more interesting when you look at another ‘variant’ of HSP present in our BLAT results. You might recall that in BLAT searches, sometimes we get our results separated into ‘blocks’. These blocks are essentially alignment fragments that may have some intervening sequence between them.

Let’s take a look at a BLAT HSP that contains multiple blocks to see how Bio.SearchIO deals with this:

>>> blat_hsp2 = blat_qresult[0][1]  # first hit, second hsp

>>> print(blat_hsp2)
Query: mystery_seq <unknown description>
Hit: chr19 <unknown description>
Query range: [0:61] (1)
Hit range: [54233104:54264463] (1)
Quick stats: evalue ?; bitscore ?
Fragments: --- -------------- ---------------------- ----------------------
# Span Query range Hit range
--- -------------- ---------------------- ----------------------
0 ? [0:18] [54233104:54233122]
1 ? [18:61] [54264420:54264463]

What’s happening here? We still some essential details covered: the IDs and descriptions, the coordinates, and the quick statistics are similar to what you’ve seen before. But the fragments detail is all different. Instead of showing ‘Fragments: 1’, we now have a table with two data rows.

This is how Bio.SearchIO deals with HSPs having multiple fragments. As mentioned before, an HSP alignment may be separated by intervening sequences into fragments. The intervening sequences are not part of the query-hit match, so they should not be considered part of query nor hit sequence. However, they do affect how we deal with sequence coordinates, so we can’t ignore them.

Take a look at the hit coordinate of the HSP above. In the Hit range: field, we see that the coordinate is [54233104:54264463]. But looking at the table rows, we see that not the entire region spanned by this coordinate matches our query. Specifically, the intervening region spans from 54233122 to 54264420.

Why then, is the query coordinates seem to be contiguous, you ask? This is perfectly fine. In this case it means that the query match is contiguous (no intervening regions), while the hit match is not.

All these attributes are accessible from the HSP directly, by the way:
Most of these attributes are not readily available from the PSL file we have, but `Bio.SearchIO` calculates them for you on the fly when you parse the PSL file. All it needs are the start and end coordinates of each fragment.

What about the `query`, `hit`, and `aln` attributes? If the HSP has multiple fragments, you won’t be able to use these attributes as they only fetch single `SeqRecord` or `MultipleSeqAlignment` objects. However, you can use their `_all` counterparts: `query_all`, `hit_all`, and `aln_all`. These properties will return a list containing `SeqRecord` or `MultipleSeqAlignment` objects from each of the HSP fragment. There are other attributes that behave similarly, i.e. they only work for HSPs with one fragment. Check out the HSP documentation for a full list.

Finally, to check whether you have multiple fragments or not, you can use the `is_fragmented` property like so:

```python
>>> blat_hsp2.is_fragmented  # BLAT HSP with 2 fragments
True
>>> blat_hsp.is_fragmented   # BLAT HSP from earlier, with one fragment
False
```

Before we move on, you should also know that we can use the slice notation on HSP objects, just like `QueryResult` or `Hit` objects. When you use this notation, you’ll get an `HSPFragment` object in return, the last component of the object model.

### 11.1.4 HSPFragment

`HSPFragment` represents a single, contiguous match between the query and hit sequences. You could consider it the core of the object model and search result, since it is the presence of these fragments that determine whether your search have results or not.

In most cases, you don’t have to deal with `HSPFragment` objects directly since not that many sequence search tools fragment their HSPs. When you do have to deal with them, what you should remember is that `HSPFragment` objects were written with to be as compact as possible. In most cases, they only contain attributes directly related to sequences: strands, reading frames, molecule types, coordinates, the sequences themselves, and their IDs and descriptions.

These attributes are readily shown when you invoke `print` on an `HSPFragment`. Here’s an example, taken from our BLAST search:

```python
>>> from Bio import SearchIO
>>> blast_qresult = SearchIO.read("my_blast.xml", "blast-xml")
>>> blast_frag = blast_qresult[0][0][0]  # first hit, first hsp, first fragment
>>> print(blast_frag)
Query: 42291 mystery_seq
Hit: gi|262205317|ref|NR_030195.1| Homo sapiens microRNA 520b (MIR520...
```
At this level, the BLAT fragment looks quite similar to the BLAST fragment, save for the query and hit sequences which are not present:

```python
>>> blat_qresult = SearchIO.read("my_blat.psl", "blat-psl")
>>> blat_frag = blat_qresult[0][0][0]  # first hit, first hsp, first fragment
>>> print(blat_frag)
Query: mystery_seq <unknown description>
Hit: chr19 <unknown description>
```

In all cases, these attributes are accessible using our favorite dot notation. Some examples:

```python
>>> blast_frag.query_start  # query start coordinate
0
>>> blast_frag.hit_strand  # hit sequence strand
1
>>> blast_frag.hit  # hit sequence, as a SeqRecord object
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTAGAGGG'), id='gi|262205317|ref|NR_030195.1|', name='aligned hit sequence', description='Homo sapiens microRNA 520b (MIR520B), microRNA', dbxrefs=[])```

### 11.2 A note about standards and conventions

Before we move on to the main functions, there is something you ought to know about the standards Bio.SearchIO uses. If you’ve worked with multiple sequence search tools, you might have had to deal with the many different ways each program deals with things like sequence coordinates. It might not have been a pleasant experience as these search tools usually have their own standards. For example, one tools might use one-based coordinates, while the other uses zero-based coordinates. Or, one program might reverse the start and end coordinates if the strand is minus, while others don’t. In short, these often creates unnecessary mess must be dealt with.

We realize this problem ourselves and we intend to address it in Bio.SearchIO. After all, one of the goals of Bio.SearchIO is to create a common, easy to use interface to deal with various search output files. This means creating standards that extend beyond the object model you just saw.

Now, you might complain, "Not another standard!". Well, eventually we have to choose one convention or the other, so this is necessary. Plus, we’re not creating something entirely new here; just adopting a standard we think is best for a Python programmer (it is Biopython, after all).

There are three implicit standards that you can expect when working with Bio.SearchIO:

- The first one pertains to sequence coordinates. In Bio.SearchIO, all sequence coordinates follows Python’s coordinate style: zero-based and half open. For example, if in a BLAST XML output file the start and end coordinates of an HSP are 10 and 28, they would become 9 and 28 in Bio.SearchIO. The start coordinate becomes 9 because Python indices start from zero, while the end coordinate remains 28 as Python slices omit the last item in an interval.

- The second is on sequence coordinate orders. In Bio.SearchIO, start coordinates are always less than or equal to end coordinates. This isn’t always the case with all sequence search tools, as some of them have larger start coordinates when the sequence strand is minus.
• The last one is on strand and reading frame values. For strands, there are only four valid choices: 1 (plus strand), -1 (minus strand), 0 (protein sequences), and None (no strand). For reading frames, the valid choices are integers from -3 to 3 and None.

Note that these standards only exist in Bio.SearchIO objects. If you write Bio.SearchIO objects into an output format, Bio.SearchIO will use the format’s standard for the output. It does not force its standard over to your output file.

11.3 Reading search output files

There are two functions you can use for reading search output files into Bio.SearchIO objects: read and parse. They’re essentially similar to read and parse functions in other submodules like Bio.SeqIO or Bio.AlignIO. In both cases, you need to supply the search output file name and the file format name, both as Python strings. You can check the documentation for a list of format names Bio.SearchIO recognizes.

Bio.SearchIO.read is used for reading search output files with only one query and returns a QueryResult object. You’ve seen read used in our previous examples. What you haven’t seen is that read may also accept additional keyword arguments, depending on the file format.

Here are some examples. In the first one, we use read just like previously to read a BLAST tabular output file. In the second one, we use a keyword argument to modify so it parses the BLAST tabular variant with comments in it:

```python
>>> from Bio import SearchIO
>>> qresult = SearchIO.read("tab_2226_tblastn_003.txt", "blast-tab")
>>> qresult
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> qresult2 = SearchIO.read("tab_2226_tblastn_007.txt", "blast-tab", comments=True)
>>> qresult2
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
```

These keyword arguments differs among file formats. Check the format documentation to see if it has keyword arguments that modifies its parser’s behavior.

As for the Bio.SearchIO.parse, it is used for reading search output files with any number of queries. The function returns a generator object that yields a QueryResult object in each iteration. Like Bio.SearchIO.read, it also accepts format-specific keyword arguments:

```python
>>> from Bio import SearchIO
>>> qresults = SearchIO.parse("tab_2226_tblastn_001.txt", "blast-tab")
>>> for qresult in qresults:
...     print(qresult.id)
...     print(qresult2.id)
...     print(qresult3.id)
```

random_s00

```python
>>> qresults2 = SearchIO.parse("tab_2226_tblastn_005.txt", "blast-tab", comments=True)
>>> for qresult in qresults2:
...     print(qresult.id)
...     print(qresult2.id)
...     print(qresult3.id)
```

random_s00
11.4 Dealing with large search output files with indexing

Sometimes, you’re handed a search output file containing hundreds or thousands of queries that you need to parse. You can of course use Bio.SearchIO.parse for this file, but that would be grossly inefficient if you need to access only a few of the queries. This is because parse will parse all queries it sees before it fetches your query of interest.

In this case, the ideal choice would be to index the file using Bio.SearchIO.index or Bio.SearchIO.index_db. If the names sound familiar, it’s because you’ve seen them before in Section 5.4.2. These functions also behave similarly to their Bio.SeqIO counterparts, with the addition of format-specific keyword arguments.

Here are some examples. You can use index with just the filename and format name:

```python
>>> from Bio import SearchIO
>>> idx = SearchIO.index("tab_2226_tblastn_001.txt", "blast-tab")
>>> sorted(idx.keys())
['gi|11464971:4-101', 'gi|16080617|ref|NP_391444.1|']
>>> idx["gi|16080617|ref|NP_391444.1"]
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> idx.close()
```

Or also with the format-specific keyword argument:

```python
>>> idx = SearchIO.index("tab_2226_tblastn_005.txt", "blast-tab", comments=True)
>>> sorted(idx.keys())
['gi|11464971:4-101', 'gi|16080617|ref|NP_391444.1|', 'random_s00']
>>> idx["gi|16080617|ref|NP_391444.1"]
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> idx.close()
```

Or with the key_function argument, as in Bio.SeqIO:

```python
>>> key_function = lambda id: id.upper()  # capitalizes the keys
>>> idx = SearchIO.index("tab_2226_tblastn_001.txt", "blast-tab", key_function=key_function)
>>> sorted(idx.keys())
['GI|11464971:4-101', 'GI|16080617|REF|NP_391444.1|']
>>> idx["GI|16080617|REF|NP_391444.1"]
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> idx.close()
```

Bio.SearchIO.index_db works like as index, only it writes the query offsets into an SQLite database file.

11.5 Writing and converting search output files

It is occasionally useful to be able to manipulate search results from an output file and write it again to a new file. Bio.SearchIO provides a write function that lets you do exactly this. It takes as its arguments an iterable returning QueryResult objects, the output filename to write to, the format name to write to, and optionally some format-specific keyword arguments. It returns a four-item tuple, which denotes the number of QueryResult, Hit, HSP, and HSPFragment objects that were written.

```python
>>> from Bio import SearchIO
>>> qresults = SearchIO.parse("mirna.xml", "blast-xml")  # read XML file
>>> SearchIO.write(qresults, "results.tab", "blast-tab")  # write to tabular file
(3, 239, 277, 277)
```
You should note different file formats require different attributes of the `QueryResult`, `Hit`, `HSP` and `HSPFragment` objects. If these attributes are not present, writing won't work. In other words, you can't always write to the output format that you want. For example, if you read a BLAST XML file, you wouldn't be able to write the results to a PSL file as PSL files require attributes not calculated by BLAST (e.g. the number of repeat matches). You can always set these attributes manually, if you really want to write to PSL, though.

Like `read`, `parse`, `index`, and `index_db`, `write` also accepts format-specific keyword arguments. Check out the documentation for a complete list of formats `Bio.SearchIO` can write to and their arguments.

Finally, `Bio.SearchIO` also provides a `convert` function, which is simply a shortcut for `Bio.SearchIO.parse` and `Bio.SearchIO.write`. Using the convert function, our example above would be:

```python
>>> from Bio import SearchIO
>>> SearchIO.convert("mirna.xml", "blast-xml", "results.tab", "blast-tab")
(3, 239, 277, 277)
```

As `convert` uses `write`, it is only limited to format conversions that have all the required attributes. Here, the BLAST XML file provides all the default values a BLAST tabular file requires, so it works just fine. However, other format conversions are less likely to work since you need to manually assign the required attributes first.
Chapter 12

Accessing NCBI’s Entrez databases

Entrez (https://www.ncbi.nlm.nih.gov/Web/Search/entrezfs.html) is a data retrieval system that provides users access to NCBI’s databases such as PubMed, GenBank, GEO, and many others. You can access Entrez from a web browser to manually enter queries, or you can use Biopython’s Bio.Entrez module for programmatic access to Entrez. The latter allows you for example to search PubMed or download GenBank records from within a Python script.

The Bio.Entrez module makes use of the Entrez Programming Utilities (also known as EUtils), consisting of eight tools that are described in detail on NCBI’s page at https://www.ncbi.nlm.nih.gov/books/NBK25501/. Each of these tools corresponds to one Python function in the Bio.Entrez module, as described in the sections below. This module makes sure that the correct URL is used for the queries, and that NCBI’s guidelines for responsible data access are being followed.

The output returned by the Entrez Programming Utilities is typically in XML format. To parse such output, you have several options:

1. Use Bio Entrez’s parser to parse the XML output into a Python object;
2. Use one of the XML parsers available in Python’s standard library;
3. Read the XML output as raw text, and parse it by string searching and manipulation.

See the Python documentation for a description of the XML parsers in Python’s standard library. Here, we discuss the parser in Biopython’s Bio.Entrez module. This parser can be used to parse data as provided through Bio.Entrez’s programmatic access functions to Entrez, but can also be used to parse XML data from NCBI Entrez that are stored in a file. In the latter case, the XML file should be opened in binary mode (e.g. open("myfile.xml", "rb")) for the XML parser in Bio.Entrez to work correctly. Alternatively, you can pass the file name or path to the XML file, and let Bio.Entrez take care of opening and closing the file.

NCBI uses DTD (Document Type Definition) files to describe the structure of the information contained in XML files. Most of the DTD files used by NCBI are included in the Biopython distribution. The Bio.Entrez parser makes use of the DTD files when parsing an XML file returned by NCBI Entrez.

Occasionally, you may find that the DTD file associated with a specific XML file is missing in the Biopython distribution. In particular, this may happen when NCBI updates its DTD files. If this happens, Entrez.read will show a warning message with the name and URL of the missing DTD file. The parser will proceed to access the missing DTD file through the internet, allowing the parsing of the XML file to continue. However, the parser is much faster if the DTD file is available locally. For this purpose, please download the DTD file from the URL in the warning message and place it in the directory ...site-packages/Bio/Entrez/DTDs, containing the other DTD files. If you don’t have write access to this directory, you can also place the DTD file in ~/.biopython/Bio/Entrez/DTDs, where ~ represents your home directory. Since this directory is read before the directory ...site-packages/Bio/Entrez/DTDs, you can also put newer versions of DTD files there if the ones in ...site-packages/Bio/Entrez/DTDs become
outdated. Alternatively, if you installed Biopython from source, you can add the DTD file to the source code’s `Bio/Entrez/DTDs` directory, and reinstall Biopython. This will install the new DTD file in the correct location together with the other DTD files.

The Entrez Programming Utilities can also generate output in other formats, such as the Fasta or GenBank file formats for sequence databases, or the MedLine format for the literature database, discussed in Section 12.13.

The functions in `Bio.Entrez` for programmatic access to Entrez return data either in binary format or in text format, depending on the type of data requested. In most cases, these functions return data in text format by decoding the data obtained from NCBI Entrez to Python strings under the assumption that the encoding is UTF-8. However, XML data are returned in binary format. The reason for this is that the encoding is specified in the XML document itself, which means that we won’t know the correct encoding to use until we start parsing the file. `Bio.Entrez`’s parser therefore accepts data in binary format, extracts the encoding from the XML, and uses it to decode all text in the XML document to Python strings, ensuring that all text (in particular in languages other than English) are interpreted correctly. This is also the reason why you should open an XML file a binary mode when you want to use `Bio.Entrez`’s parser to parse the file.

## 12.1 Entrez Guidelines

Before using Biopython to access the NCBI’s online resources (via `Bio.Entrez` or some of the other modules), please read the NCBI’s Entrez User Requirements. If the NCBI finds you are abusing their systems, they can and will ban your access!

To paraphrase:

- For any series of more than 100 requests, do this at weekends or outside USA peak times. This is up to you to obey.
- Use the `https://eutils.ncbi.nlm.nih.gov` address, not the standard NCBI Web address. Biopython uses this web address.
- If you are using a API key, you can make at most 10 queries per second, otherwise at most 3 queries per second. This is automatically enforced by Biopython. Include `api_key="MyAPIkey"` in the argument list or set it as a module level variable:

```python
>>> from Bio import Entrez
>>> Entrez.api_key = "MyAPIkey"
```

- Use the optional email parameter so the NCBI can contact you if there is a problem. You can either explicitly set this as a parameter with each call to Entrez (e.g. include `email="A.N.Other@example.com"` in the argument list), or you can set a global email address:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"
```

`Bio.Entrez` will then use this email address with each call to Entrez. The `example.com` address is a reserved domain name specifically for documentation (RFC 2606). Please DO NOT use a random email – it’s better not to give an email at all. The email parameter has been mandatory since June 1, 2010. In case of excessive usage, NCBI will attempt to contact a user at the e-mail address provided prior to blocking access to the E-utilities.

- If you are using Biopython within some larger software suite, use the tool parameter to specify this. You can either explicitly set the tool name as a parameter with each call to Entrez (e.g. include `tool="MyLocalScript"` in the argument list), or you can set a global tool name:
The tool parameter will default to Biopython.

- For large queries, the NCBI also recommend using their session history feature (the WebEnv session cookie string, see Section 12.16). This is only slightly more complicated.

In conclusion, be sensible with your usage levels. If you plan to download lots of data, consider other options. For example, if you want easy access to all the human genes, consider fetching each chromosome by FTP as a GenBank file, and importing these into your own BioSQL database (see Section 22.3).

### 12.2 EInfo: Obtaining information about the Entrez databases

EInfo provides field index term counts, last update, and available links for each of NCBI's databases. In addition, you can use EInfo to obtain a list of all database names accessible through the Entrez utilities:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.einfo()
>>> result = stream.read()
>>> stream.close()
```

The variable `result` now contains a list of databases in XML format:

```python
>>> print(result)
<?xml version="1.0"?><!DOCTYPE eInfoResult PUBLIC "-//NLM//DTD eInfoResult, 11 May 2002//EN" "https://www.ncbi.nlm.nih.gov/entrez/query/DTD/eInfo_020511.dtd"><eInfoResult><DbList><DbName>pubmed</DbName><DbName>protein</DbName><DbName>nucleotide</DbName><DbName>nucore</DbName><DbName>nucgss</DbName><DbName>nucest</DbName><DbName>structure</DbName><DbName>genome</DbName><DbName>books</DbName><DbName>cancerchromosomes</DbName><DbName>cdd</DbName><DbName>gap</DbName><DbName>domains</DbName><DbName>gene</DbName><DbName>genomeprj</DbName><DbName>gensat</DbName><DbName>geo</DbName><DbName>gds</DbName><DbName>homologene</DbName><DbName>journal</DbName><DbName>mesh</DbName><DbName>ncbisearch</DbName></DbList></eInfoResult>
```
Since this is a fairly simple XML file, we could extract the information it contains simply by string searching. Using Bio.Entrez's parser instead, we can directly parse this XML file into a Python object:

```python
>>> from Bio import Entrez
>>> stream = Entrez.einfo()
>>> record = Entrez.read(stream)
```

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

```python
>>> record.keys()
dict_keys(['DbList'])
```

The values stored in this key is the list of database names shown in the XML above:

```python
>>> record['DbList']
['pubmed', 'protein', 'nucleotide', 'nuccore', 'nucgss', 'nucest',
 'structure', 'genome', 'books', 'cancerchromosomes', 'cdd', 'gap',
 'domains', 'gene', 'genomemprj', 'gensat', 'geo', 'gds', 'homologene',
 'journals', 'mesh', 'ncbisearch', 'nlmcatalog', 'omia', 'omim', 'pmc',
 'popset', 'probe', 'proteinclusters', 'pcassay', 'pccompound',
 'pccompound', 'snp', 'taxonomy', 'toolkit', 'unigene', 'unists']
```

For each of these databases, we can use EInfo again to obtain more information:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.einfo(db="pubmed")
>>> record = Entrez.read(stream)
```

Try `record["DbInfo"].keys()` for other information stored in this record. One of the most useful is a list of possible search fields for use with ESearch:
for field in record['DbInfo']['FieldList']:
    ... print("%(Name)s, %(FullName)s, %(Description)s" % field)
...
ALL, All Fields, All terms from all searchable fields
UID, UID, Unique number assigned to publication
FILT, Filter, Limits the records
TITL, Title, Words in title of publication
WORD, Text Word, Free text associated with publication
MESH, MeSH Terms, Medical Subject Headings assigned to publication
MAJR, MeSH Major Topic, MeSH terms of major importance to publication
AUTH, Author, Author(s) of publication
JOUR, Journal, Journal abbreviation of publication
AFFL, Affiliation, Author's institutional affiliation and address
...

That's a long list, but indirectly this tells you that for the PubMed database, you can do things like Jones[AUTH] to search the author field, or Sanger[AFFL] to restrict to authors at the Sanger Centre. This can be very handy - especially if you are not so familiar with a particular database.

12.3 ESearch: Searching the Entrez databases

To search any of these databases, we use Bio.Entrez.esearch(). For example, let's search in PubMed for publications that include Biopython in their title:

>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="pubmed", term="biopython[title]", retmax="40")
>>> record = Entrez.read(stream)
>>> "19304878" in record['IdList']
True

>>> print(record['IdList'])
['22909249', '19304878']

In this output, you see PubMed IDs (including 19304878 which is the PMID for the Biopython application note), which can be retrieved by EFetch (see section 12.6).

You can also use ESearch to search GenBank. Here we'll do a quick search for the matK gene in Cypripedioideae orchids (see Section 12.2 about EInfo for one way to find out which fields you can search in each Entrez database):

>>> stream = Entrez.esearch(  
...     db="nucleotide", term="Cypripedioideae[Orgn] AND matK[Gene]", idtype="acc"
... )
>>> record = Entrez.read(stream)
>>> record["Count"]
'348'
>>> record["IdList"]
['JQ660909.1', 'JQ660908.1', 'JQ660907.1', 'JQ660906.1', ..., 'JQ660890.1']

Each of the IDs (JQ660909.1, JQ660908.1, JQ660907.1, ...) is a GenBank identifier (Accession number). See section 12.6 for information on how to actually download these GenBank records.

Note that instead of a species name like Cypripedioideae[Orgn], you can restrict the search using an NCBI taxon identifier, here this would be txid158330[Orgn]. This isn’t currently documented on the
ESearch help page - the NCBI explained this in reply to an email query. You can often deduce the search term formatting by playing with the Entrez web interface. For example, including complete[prop] in a genome search restricts to just completed genomes.

As a final example, let’s get a list of computational journal titles:

```python
>>> stream = Entrez.esearch(db="nlmcatalog", term="computational[Journal]", retmax="20")
>>> record = Entrez.read(stream)
>>> print("{} computational journals found".format(record["Count"]))
117 computational Journals found
>>> print("The first 20 are\n{}".format(record["IdList"]))
['101660833', '101664671', '101661657', '101659814', '101657941', '101653734', '101669877', '101649614', '101647835', '101639023', '101627224', '101647801', '101589678', '101585369', '101645372', '101586429', '101582229', '101574747', '101564639', '101671907']
```

Again, we could use EFetch to obtain more information for each of these journal IDs.

ESearch has many useful options — see the ESearch help page for more information.

### 12.4 EPost: Uploading a list of identifiers

EPost uploads a list of UIs for use in subsequent search strategies; see the EPost help page for more information. It is available from Biopython through the Bio.Entrez.epost() function.

To give an example of when this is useful, suppose you have a long list of IDs you want to download using EFetch (maybe sequences, maybe citations – anything). When you make a request with EFetch your list of IDs, the database etc, are all turned into a long URL sent to the server. If your list of IDs is long, this URL gets long, and long URLs can break (e.g. some proxies don’t cope well).

Instead, you can break this up into two steps, first uploading the list of IDs using EPost (this uses an “HTML post” internally, rather than an “HTML get”, getting round the long URL problem). With the history support, you can then refer to this long list of IDs, and download the associated data with EFetch.

Let’s look at a simple example to see how EPost works – uploading some PubMed identifiers:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> id_list = ['19304878', '18606172', '16403221', '16377612', '14871861', '14630660']
>>> print(Entrez.epost("pubmed", id=",".join(id_list)).read())
<?xml version="1.0"?>
<ePostResult>
  <QueryKey>1</QueryKey>
  <WebEnv>NCID_01_206841095_130.14.22.101_9001_1242061629</WebEnv>
</ePostResult>
```

The returned XML includes two important strings, QueryKey and WebEnv which together define your history session. You would extract these values for use with another Entrez call such as EFetch:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> id_list = ['19304878', '18606172', '16403221', '16377612', '14871861', '14630660']
>>> search_results = Entrez.read(Entrez.epost("pubmed", id=",".join(id_list)))
>>> webenv = search_results["WebEnv"]
>>> query_key = search_results["QueryKey"]
```

Section 12.16 shows how to use the history feature.
12.5 ESummary: Retrieving summaries from primary IDs

ESummary retrieves document summaries from a list of primary IDs (see the ESummary help page for more information). In Biopython, ESummary is available as Bio.Entrez.esummary(). Using the search result above, we can for example find out more about the journal with ID 30367:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esummary(db="nlmcatalog", id="101660833")
>>> record = Entrez.read(stream)
>>> info = record[0]["TitleMainList"]
>>> print("Journal info\n\nid: {}\nTitle: {}").format(record[0]["Id"], info["Title"])
Journal info
id: 101660833
Title: IEEE transactions on computational imaging.
```

12.6 EFetch: Downloading full records from Entrez

EFetch is what you use when you want to retrieve a full record from Entrez. This covers several possible databases, as described on the main EFetch Help page.

For most of their databases, the NCBI support several different file formats. Requesting a specific file format from Entrez using Bio.Entrez.efetch() requires specifying the rettype and/or retmode optional arguments. The different combinations are described for each database type on the pages linked to on NCBI efetch webpage.

One common usage is downloading sequences in the FASTA or GenBank/GenPept plain text formats (which can then be parsed with Bio.SeqIO, see Sections 5.3.1 and 12.6). From the Cypripedioideae example above, we can download GenBank record EU490707 using Bio.Entrez.efetch:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.efetch(db="nucleotide", id="EU490707", rettype="gb", retmode="text")
>>> print(stream.read())
LOCUS EU490707 1302 bp DNA linear PLN 26-JUL-2016
DEFINITION Selenipedium aequinoctiale maturase K (matK) gene, partial cds; chloroplast.
ACCESSION EU490707
VERSION EU490707.1
KEYWORDS .
SOURCE chloroplast Selenipedium aequinoctiale
ORGANISM Selenipedium aequinoctiale
   Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;
   Spermatophyta; Magnoliopsida; Liliopsida; Asparagales; Orchidaceae;
   Cypripedioideae; Selenipedium.
REFERENCE 1 (bases 1 to 1302)
   AUTHORS Neubig,K.M., Whitten,W.M., Carlsward,B.S., Blanco,M.A., Endara,L.,
   Williams,N.H. and Moore,M.
   TITLE Phylogenetic utility of ycf1 in orchids: a plastid gene more
   variable than matK
REFERENCE 2 (bases 1 to 1302)
   AUTHORS Neubig,K.M., Whitten,W.M., Carlsward,B.S., Blanco,M.A.,
   Endara,C.L., Williams,N.H. and Moore,M.J.
```
TITLE Direct Submission
JOURNAL Submitted (14-FEB-2008) Department of Botany, University of Florida, 220 Bartram Hall, Gainesville, FL 32611-8526, USA
FEATURES Location/Qualifiers
  source
    /organism="Selenipedium aequinoctiale"
    /organelle="plastid:chloroplast"
    /mol_type="genomic DNA"
    /specimen_voucher="FLAS:Blanco 2475"
    /db_xref="taxon:256374"
  gene
    /gene="matK"
  CDS
    /gene="matK"
    /codon_start=1
    /translation="IFYEPVEIFGYNKSSLVLVKRLTMRMYQONFLISSVWDSNQKGFWHKHFSSHFSQMVSEFGVILEIPFSSQLVSSLEEKKIPQNYNRLSIFSPFLFDKLHLNVDSDLHIPHIILEILVQLIRICIVPWSLLLLRRLLFHNYNHNLSLSISKFLYFASSKKRKFLWLYNYSYVECEYLFQF4LKRQSSVYLRSTSSGVFLERTHLYVKIEHLL\LVCCNSFQRILCFKPDPMHYVRYQGKAILASKGLTLMKKWKHFLVHNFWYSYFHFWSQPYRHEK13NYSPSLQFSSVLENLVVRNQMLENFIIINLTKKKFDTIAPVILSGLSALKQCTVLGHPISKP1WDSDSILDRCRIFICRHLCRYHSGSKKQVLYRIKYILRLSC\RTALKHKSTVTRTFMRRLGSGLLEEFFMEE"
Please be aware that as of October 2016 GI identifiers are discontinued in favor of accession numbers. You can still fetch sequences based on their GI, but new sequences are no longer given this identifier. You should instead refer to them by the “Accession number” as done in the example.

The arguments `rettype="gb"` and `retmode="text"` let us download this record in the GenBank format.

Note that until Easter 2009, the Entrez EFetch API let you use “genbank” as the return type, however the NCBI now insist on using the official return types of “gb” or “gbwithparts” (or “gp” for proteins) as described online. Also note that until Feb 2012, the Entrez EFetch API would default to returning plain text files, but now defaults to XML.

Alternatively, you could for example use `rettype="fasta"` to get the Fasta-format; see the EFetch Sequences Help page for other options. Remember – the available formats depend on which database you are downloading from - see the main EFetch Help page.

If you fetch the record in one of the formats accepted by Bio.SeqIO (see Chapter 5), you could directly parse it into a SeqRecord:

```python
>>> from Bio import SeqIO
>>> stream = Entrez.efetch(db="nucleotide", id="EU490707", rettype="gb", retmode="text")
>>> record = SeqIO.read(stream, "genbank")
>>> stream.close()
>>> print(record.id)
EU490707.1
>>> print(record.name)
EU490707
>>> print(len(record.features))
3
>>> record.seq
Seq('ATTTTTTACGAACCTGTGGAAATTTTTGGTTATGACAATAAATCTAGTTTAGTA...GAA')
```

Note that a more typical use would be to save the sequence data to a local file, and then parse it with Bio.SeqIO. This can save you having to re-download the same file repeatedly while working on your script, and places less load on the NCBI’s servers. For example:

```python
import os
from Bio import SeqIO
from Bio import Entrez

Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
filename = "EU490707.gbk"
if not os.path.isfile(filename):
    # Downloading...
    stream = Entrez.efetch(db="nucleotide", id="EU490707", rettype="gb", retmode="text")
    output = open(filename, "w")
    output.write(stream.read())
    output.close()
    stream.close()
    print("Saved")
print("Parsing...")
```
record = SeqIO.read(filename, "genbank")
print(record)

To get the output in XML format, which you can parse using the Bio.Entrez.read() function, use retmode="xml":

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.efetch(db="nucleotide", id="EU490707", retmode="xml")
>>> record = Entrez.read(stream)
>>> stream.close()
>>> record[0]["GBSeq_definition"]
'Selenipedium aequinoctiale maturase K (matK) gene, partial cds; chloroplast'
>>> record[0]["GBSeq_source"]
'chloroplast Selenipedium aequinoctiale'
```

So, that dealt with sequences. For examples of parsing file formats specific to the other databases (e.g. the MEDLINE format used in PubMed), see Section 12.13.

If you want to perform a search with Bio.Entrez.esearch(), and then download the records with Bio.Entrez.efetch(), you should use the WebEnv history feature – see Section 12.16.

### 12.7 ELink: Searching for related items in NCBI Entrez

ELink, available from Biopython as Bio.Entrez.elink(), can be used to find related items in the NCBI Entrez databases. For example, you can use this to find nucleotide entries for an entry in the gene database, and other cool stuff.

Let’s use ELink to find articles related to the Biopython application note published in Bioinformatics in 2009. The PubMed ID of this article is 19304878:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> pmid = "19304878"
>>> record = Entrez.read(Entrez.elink(dbfrom="pubmed", id=pmid))
```

The `record` variable consists of a Python list, one for each database in which we searched. Since we specified only one PubMed ID to search for, `record` contains only one item. This item is a dictionary containing information about our search term, as well as all the related items that were found:

```python
>>> record[0]["DbFrom"]
'pubmed'
>>> record[0]["IdList"]
['19304878']
```

The "LinkSetDb" key contains the search results, stored as a list consisting of one item for each target database. In our search results, we only find hits in the PubMed database (although sub-divided into categories):

```python
>>> len(record[0]["LinkSetDb"])
8
```

The exact numbers should increase over time:
>>> for linksetdb in record[0]["LinkSetDb"]:  
...     print(linksetdb["DbTo"], linksetdb["LinkName"], len(linksetdb["Link"]))  
...  
pubmed  pubmed_pubmed 284  
pubmed  pubmed_pubmed_alsoviewed 7  
pubmed  pubmed_pubmed_citedin 926  
pubmed  pubmed_pubmed_combined 6  
pubmed  pubmed_pubmed_five 6  
pubmed  pubmed_pubmed_ref 17  
pubmed  pubmed_pubmed_reviews 12  
pubmed  pubmed_pubmed_reviews_five 6

The actual search results are stored as under the "Link" key.  
Let’s now at the first search result:

>>> record[0]["LinkSetDb"][0]["Link"][0]  
{'Id': '19304878'}

This is the article we searched for, which doesn't help us much, so let’s look at the second search result:

>>> record[0]["LinkSetDb"][0]["Link"][1]  
{'Id': '14630660'}

This paper, with PubMed ID 14630660, is about the Biopython PDB parser.

We can use a loop to print out all PubMed IDs:

>>> for link in record[0]["LinkSetDb"][0]["Link"]:  
...     print(link["Id"])

...  
19304878  
14630660  
18689808  
17121776  
16377612  
12368254  
......

Now that was nice, but personally I am often more interested to find out if a paper has been cited. Well,  
ELink can do that too – at least for journals in Pubmed Central (see Section 12.16.3).

For help on ELink, see the ELink help page. There is an entire sub-page just for the link names, describing  
how different databases can be cross referenced.

12.8 EGQuery: Global Query - counts for search terms

EGQuery provides counts for a search term in each of the Entrez databases (i.e. a global query). This  
is particularly useful to find out how many items your search terms would find in each database without  
actually performing lots of separate searches with ESearch (see the example in 12.15.2 below).

In this example, we use Bio.Entrez.egquery() to obtain the counts for “Biopython”:

>>> from Bio import Entrez  
>>> Entrez.email = "A.N.Other@example.com"  
# Always tell NCBI who you are  
>>> stream = Entrez.egquery(term="biopython")  
>>> record = Entrez.read(stream)
>>> for row in record["eGQueryResult"]:
...    print(row["DbName"], row["Count"])
...  pubmed 6
  pmc 62
  journals 0
...

See the EGQuery help page for more information.

### 12.9 ESpell: Obtaining spelling suggestions

ESpell retrieves spelling suggestions. In this example, we use `Bio.Entrez.espell()` to obtain the correct spelling of Biopython:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.espell(term="biopythoon")
>>> record = Entrez.read(stream)
>>> record["Query"]
'biopythoon'
>>> record["CorrectedQuery"]
'biopython'
```

See the ESpell help page for more information. The main use of this is for GUI tools to provide automatic suggestions for search terms.

### 12.10 Parsing huge Entrez XML files

The `Entrez.read` function reads the entire XML file returned by Entrez into a single Python object, which is kept in memory. To parse Entrez XML files too large to fit in memory, you can use the function `Entrez.parse`. This is a generator function that reads records in the XML file one by one. This function is only useful if the XML file reflects a Python list object (in other words, if `Entrez.read` on a computer with infinite memory resources would return a Python list).

For example, you can download the entire Entrez Gene database for a given organism as a file from NCBI's ftp site. These files can be very large. As an example, on September 4, 2009, the file `Homo_sapiens.ags.gz`, containing the Entrez Gene database for human, had a size of 116576 kB. This file, which is in the ASN format, can be converted into an XML file using NCBI's `gene2xml` program (see NCBI's ftp site for more information):

```bash
$ gene2xml -b T -i Homo_sapiens.ags -o Homo_sapiens.xml
```

The resulting XML file has a size of 6.1 GB. Attempting `Entrez.read` on this file will result in a `MemoryError` on many computers.

The XML file `Homo_sapiens.xml` consists of a list of Entrez gene records, each corresponding to one Entrez gene in human. `Entrez.parse` retrieves these gene records one by one. You can then print out or store the relevant information in each record by iterating over the records. For example, this script iterates over the Entrez gene records and prints out the gene numbers and names for all current genes:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = open("Homo_sapiens.xml", "rb")
>>> records = Entrez.parse(stream)
```
Alternatively, you can use

```
>>> records = Entrez.parse("Homo_sapiens.xml")
```

and let `Bio.Entrez` take care of opening and closing the file. This is safer, as the file will then automatically be closed after parsing it, or if an error occurs.

```
>>> for record in records:
...     status = record["Entrezgene_track-info"]['Gene-track']['Gene-track_status']
...     if status.attributes['value'] == "discontinued":
...         continue
...     geneid = record["Entrezgene_track-info"]['Gene-track']['Gene-track_geneid']
...     genename = record["Entrezgene_gene"]['Gene-ref']['Gene-ref_locus']
...     print(geneid, genename)
...1 A1BG
2 A2M
3 A2MP
8 AA
9 NAT1
10 NAT2
11 AACP
12 SERPINA3
13 AADAC
14 AAMP
15 AANAT
16 AARS
17 AAAS1
...```

### 12.11 HTML escape characters

Pubmed records may contain HTML tags to indicate e.g. subscripts, superscripts, or italic text, as well as mathematical symbols via MathML. By default, the `Bio.Entrez` parser treats all text as plain text without markup; for example, the fragment “\( P < 0.05 \)” in the abstract of a Pubmed record, which is encoded as

\[ \text{i}P < 0.05 \]

in the XML returned by Entrez, is converted to the Python string

 `'\text{i}P < 0.05'`

by the `Bio.Entrez` parser. While this is more human-readable, it is not valid HTML due to the less-than sign, and makes further processing of the text e.g. by an HTML parser impractical. To ensure that all strings returned by the parser are valid HTML, call `Entrez.read` or `Entrez.parse` with the `escape` argument set to `True`:

```
>>> record = Entrez.read(stream, escape=True)
```

The parser will then replace all characters disallowed in HTML by their HTML-escaped equivalent; in the example above, the parser will generate

 `'\text{i}P \text{\&lt;} 0.05'`

which is a valid HTML fragment. By default, `escape` is `False`.  

250
12.12 Handling errors

The file is not an XML file

For example, this error occurs if you try to parse a Fasta file as if it were an XML file:

```python
>>> from Bio import Entrez
>>> stream = open("NC_005816.fna", "rb")  # a Fasta file
>>> record = Entrez.read(stream)
Traceback (most recent call last):
...
Bio.Entrez.Parser.NotXMLError: Failed to parse the XML data (syntax error: line 1, column 0). Please make sure that the input data are in XML format.
```

Here, the parser didn’t find the `<?xml ...` tag with which an XML file is supposed to start, and therefore decides (correctly) that the file is not an XML file.

The file ends prematurely or is otherwise corrupted

When your file is in the XML format but is corrupted (for example, by ending prematurely), the parser will raise a CorruptedXMLError.

Here is an example of an XML file that ends prematurely:

```xml
<?xml version="1.0"?>
<eInfoResult>
(DbList>
  <DbName>pubmed</DbName>
  <DbName>protein</DbName>
  <DbName>nucleotide</DbName>
  <DbName>nuccore</DbName>
  <DbName>nucgss</DbName>
  <DbName>nucest</DbName>
  <DbName>structure</DbName>
  <DbName>genome</DbName>
  <DbName>books</DbName>
  <DbName>cancerchromosomes</DbName>
  <DbName>cdd</DbName>
which will generate the following traceback:

```python
>>> Entrez.read(stream)
Traceback (most recent call last):
...
Bio.Entrez.Parser.CorruptedXMLError: Failed to parse the XML data (no element found: line 16, column 0). Please make sure that the input data are not corrupted.
```

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

The file contains items that are missing from the associated DTD

This is an example of an XML file containing tags that do not have a description in the corresponding DTD file:

```xml
<?xml version="1.0"?>
<eInfoResult>
(DbList>
  <DbName>pubmed</DbName>
  <DbName>protein</DbName>
  <DbName>nucleotide</DbName>
  <DbName>nuccore</DbName>
  <DbName>nucgss</DbName>
  <DbName>nucest</DbName>
  <DbName>structure</DbName>
  <DbName>genome</DbName>
  <DbName>books</DbName>
  <DbName>cancerchromosomes</DbName>
  <DbName>cdd</DbName>
```
In this file, for some reason the tag <DocsumList> (and several others) are not listed in the DTD file eInfo_020511.dtd, which is specified on the second line as the DTD for this XML file. By default, the parser will stop and raise a ValidationError if it cannot find some tag in the DTD:

```python
>>> from Bio import Entrez
>>> stream = open("einfo3.xml", "rb")
>>> record = Entrez.read(stream)
Traceback (most recent call last):
  File "", line 1, in <module>
  File "", line 1, in Entrez.read
  File "", line 1, in read
  File "", line 1, in _validate
Bio.Entrez.Parser.ValidationError: Failed to find tag 'DocsumList' in the DTD. To skip all tags that are not in the DTD, please call `Entrez.read` or `Entrez.parse` with the argument validate equal to False:

Optionally, you can instruct the parser to skip such tags instead of raising a ValidationError. This is done by calling `Entrez.read` or `Entrez.parse` with the argument validate equal to False:

```python
>>> from Bio import Entrez
>>> stream = open("einfo3.xml", "rb")
>>> record = Entrez.read(stream, validate=False)
>>> stream.close()
```

Of course, the information contained in the XML tags that are not in the DTD are not present in the record returned by `Entrez.read`.

The file contains an error message

This may occur, for example, when you attempt to access a PubMed record for a nonexistent PubMed ID. By default, this will raise a RuntimeError:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
```
>>> stream = Entrez.esummary(db="pubmed", id="99999999")
>>> record = Entrez.read(stream)
Traceback (most recent call last):
...
RuntimeError: UID=99999999: cannot get document summary

If you are accessing multiple PubMed records, the RuntimeError would prevent you from receiving results for any of the PubMed records if one of the PubMed IDs is incorrect. To circumvent this, you can set the ignore_errors argument to True. This will return the requested results for the valid PubMed IDs, and an ErrorElement for the incorrect ID:

>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esummary(db="pubmed", id="19304878,99999999,31278684")
>>> record = Entrez.read(stream, ignore_errors=True)
>>> len(record)
3
>>> record[0].tag
'DocSum'
>>> record[0]["Title"]
'Biopython: freely available Python tools for computational molecular biology and bioinformatics.'
>>> record[1].tag
'ERROR'
>>> record[1]
ErrorElement('UID=99999999: cannot get document summary')
>>> record[2].tag
'DocSum'
>>> record[2]["Title"]
'Sharing Programming Resources Between Bio* Projects.'

12.13 Specialized parsers

The Bio.Entrez.read() function can parse most (if not all) XML output returned by Entrez. Entrez typically allows you to retrieve records in other formats, which may have some advantages compared to the XML format in terms of readability (or download size).

To request a specific file format from Entrez using Bio.Entrez.efetch() requires specifying the rettype and/or retmode optional arguments. The different combinations are described for each database type on the NCBI efetch webpage.

One obvious case is you may prefer to download sequences in the FASTA or GenBank/GenPept plain text formats (which can then be parsed with Bio.SeqIO, see Sections 5.3.1 and 12.6). For the literature databases, Biopython contains a parser for the MEDLINE format used in PubMed.

12.13.1 Parsing Medline records

You can find the Medline parser in Bio.Medline. Suppose we want to parse the file pubmed_result1.txt, containing one Medline record. You can find this file in Biopython's Tests\Medline directory. The file looks like this:

PMID= 12230038
OWN = NLM
STAT= MEDLINE
DA = 20020916
Bioinformatics research is often difficult to do with commercial software. The Open Source BioPerl, BioPython and Biojava projects provide toolkits with multiple functionality that make it easier to create customized pipelines or analysis. This review briefly compares the quirks of the underlying languages and the functionality, documentation, utility and relative advantages of the Bio counterparts, particularly from the point of view of the beginning biologist programmer.

The key names used in a Medline record can be rather obscure; use

```python
>>> help(record)
```

for a brief summary.

To parse a file containing multiple Medline records, you can use the `parse` function instead:

```python
>>> from Bio import Medline
>>> with open("pubmed_result2.txt") as stream:
... for record in Medline.parse(stream):
...     print(record["TI"])
```

A high level interface to SCOP and ASTRAL implemented in python.
GenomeDiagram: a python package for the visualization of large-scale genomic data.
Open source clustering software.
PDB file parser and structure class implemented in Python.

Instead of parsing Medline records stored in files, you can also parse Medline records downloaded by `Bio.Entrez.efetch`. For example, let’s look at all Medline records in PubMed related to Biopython:
We now use `Bio.Entrez.efetch` to download these Medline records:

```python
>>> idlist = record["IdList"]
>>> stream = Entrez.efetch(db="pubmed", id=idlist, rettype="medline", retmode="text")
```

Here, we specify `rettype="medline", retmode="text"` to obtain the Medline records in plain-text Medline format. Now we use `Bio.Medline` to parse these records:

```python
>>> from Bio import Medline
>>> records = Medline.parse(stream)
>>> for record in records:
...     print(record["AU"])
...
["Munteanu CR", 'Gonzalez-Diaz H', 'Magalhaes AL']
["Casbon JA", 'Crooks GE', 'Saqi MA']
["Pritchard L", 'White JA', 'Birch PR', 'Toth IK']
["de Hoon MJ", 'Imoto S', 'Nolan J', 'Miyano S']
["Hamelryck T", 'Manderick B']
["Mangalam H"]
```

For comparison, here we show an example using the XML format:

```python
>>> stream = Entrez.efetch(db="pubmed", id=idlist, rettype="medline", retmode="xml")
>>> records = Entrez.read(stream)
>>> for record in records["PubmedArticle"]: ...
...     print(record["MedlineCitation"]["Article"]["ArticleTitle"])
...
```

Biopython: freely available Python tools for computational molecular biology and bioinformatics.
Enzymes/non-enzymes classification model complexity based on composition, sequence, 3D and topological indices.
A high level interface to SCOP and ASTRAL implemented in python.
GenomeDiagram: a python package for the visualization of large-scale genomic data.
Open source clustering software.
PDB file parser and structure class implemented in Python.
The Bio* toolkits--a brief overview.

Note that in both of these examples, for simplicity we have naively combined ESearch and EFetch. In this situation, the NCBI would expect you to use their history feature, as illustrated in Section 12.16.

### 12.13.2 Parsing GEO records

GEO ([Gene Expression Omnibus](https://www.ncbi.nlm.nih.gov/geo/)) is a data repository of high-throughput gene expression and hybridization array data. The `Bio.Geo` module can be used to parse GEO-formatted data.

The following code fragment shows how to parse the example GEO file `GSE16.txt` into a record and print the record:

```python
>>> stream = Entrez.efetch(db="pubmed", id=idlist, rettype="medline", retmode="xml")
>>> records = Entrez.read(stream)
>>> for record in records["PubmedArticle"]: ...
...     print(record["MedlineCitation"]["Article"]["ArticleTitle"])
...
Biopython: freely available Python tools for computational molecular biology and bioinformatics.
Enzymes/non-enzymes classification model complexity based on composition, sequence, 3D and topological indices.
A high level interface to SCOP and ASTRAL implemented in python.
GenomeDiagram: a python package for the visualization of large-scale genomic data.
Open source clustering software.
PDB file parser and structure class implemented in Python.
The Bio* toolkits--a brief overview.

Note that in both of these examples, for simplicity we have naively combined ESearch and EFetch. In this situation, the NCBI would expect you to use their history feature, as illustrated in Section 12.16.

### 12.13.2 Parsing GEO records

GEO ([Gene Expression Omnibus](https://www.ncbi.nlm.nih.gov/geo/)) is a data repository of high-throughput gene expression and hybridization array data. The `Bio.Geo` module can be used to parse GEO-formatted data.

The following code fragment shows how to parse the example GEO file `GSE16.txt` into a record and print the record:
You can search the “gds” database (GEO datasets) with ESearch:

```python
from Bio import Entrez
Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
stream = Entrez.esearch(db="gds", term="GSE16")
record = Entrez.read(stream)
stream.close()
print(record['Count'])
'27'
print(record['IdList'])
['200000016', '100000028', ...]
```

From the Entrez website, UID “200000016” is GDS16 while the other hit “100000028” is for the associated platform, GPL28. Unfortunately, at the time of writing the NCBI don’t seem to support downloading GEO files using Entrez (not as XML, nor in the Simple Omnibus Format in Text (SOFT) format).

However, it is actually pretty straightforward to download the GEO files by FTP from ftp://ftp.ncbi.nih.gov/pub/geo/ instead. In this case you might want ftp://ftp.ncbi.nih.gov/pub/geo/DATA/ SOFT/by_series/GSE16/GSE16_family.soft.gz (a compressed file, see the Python module gzip).

### 12.13.3 Parsing UniGene records

UniGene is an NCBI database of the transcriptome, with each UniGene record showing the set of transcripts that are associated with a particular gene in a specific organism. A typical UniGene record looks like this:

```
ID Hs.2
TITLE N-acetyltransferase 2 (arylamine N-acetyltransferase)
GENE NAT2
CYTOBAND 8p22
GENE_ID 10
LOCUSLINK 10
HOMOL YES
EXPRESS bone| connective tissue| intestine| liver| liver tumor| normal| soft tissue/muscle tissue tumor| adult
RESTR_EXPR adult
CHROMOSOME 8
STS ACC=PMC310725P3 UNISTS=272646
STS ACC=WIAF-2120 UNISTS=44576
STS ACC=G59899 UNISTS=137181
...
STS ACC=GDB:187676 UNISTS=155563
PROTSIM ORG=10090; PROTGI=67547949; PROTID=NP_035004.1; PCT=76.55; ALN=288
PROTSIM ORG=9796; PROTGI=149742490; PROTID=XP_001487907.1; PCT=79.66; ALN=288
PROTSIM ORG=9986; PROTGI=126722851; PROTID=NP_001075655.1; PCT=76.90; ALN=288
...
PROTSIM ORG=9598; PROTGI=114619004; PROTID=XP_519631.2; PCT=98.28; ALN=288
```
This particular record shows the set of transcripts (shown in the `SEQUENCE` lines) that originate from the human gene NAT2, encoding N-acetyltransferase. The `PROTSIM` lines show proteins with significant similarity to NAT2, whereas the `STS` lines show the corresponding sequence-tagged sites in the genome.

To parse UniGene files, use the Bio.UniGene module:

```python
>>> from Bio import UniGene
>>> input = open("myunigenefile.data")
>>> record = UniGene.read(input)
```

The `record` returned by `UniGene.read` is a Python object with attributes corresponding to the fields in the UniGene record. For example,

```python
>>> record.ID
"Hs.2"

>>> record.title
"N-acetyltransferase 2 (arylamine N-acetyltransferase)"
```

The `EXPRESS` and `RESTR_EXPR` lines are stored as Python lists of strings:

```python
[
    "bone",
    "connective tissue",
    "intestine",
    "liver",
    "liver tumor",
    "normal",
    "soft tissue/muscle tissue tumor",
    "adult",
]
```

Specialized objects are returned for the `STS`, `PROTSIM`, and `SEQUENCE` lines, storing the keys shown in each line as attributes:

```python
>>> record.sts[0].acc
'PMC310725P3'
>>> record.sts[0].unists
'272646'
```

and similarly for the `PROTSIM` and `SEQUENCE` lines.

To parse a file containing more than one UniGene record, use the `parse` function in Bio.UniGene:

```python
>>> from Bio import UniGene
>>> input = open("unigenerecords.data")
```
12.14 Using a proxy

Normally you won’t have to worry about using a proxy, but if this is an issue on your network here is how to deal with it. Internally, Bio.Entrez uses the standard Python library urllib for accessing the NCBI servers. This will check an environment variable called http_proxy to configure any simple proxy automatically. Unfortunately this module does not support the use of proxies which require authentication.

You may choose to set the http_proxy environment variable once (how you do this will depend on your operating system). Alternatively you can set this within Python at the start of your script, for example:

```python
import os

os.environ["http_proxy"] = "http://proxyhost.example.com:8080"
```

See the urllib documentation for more details.

12.15 Examples

12.15.1 PubMed and Medline

If you are in the medical field or interested in human issues (and many times even if you are not!), PubMed (https://www.ncbi.nlm.nih.gov/PubMed/) is an excellent source of all kinds of goodies. So like other things, we'd like to be able to grab information from it and use it in Python scripts.

In this example, we will query PubMed for all articles having to do with orchids (see section 2.3 for our motivation). We first check how many of such articles there are:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.egquery(term="orchid")
>>> record = Entrez.read(stream)
>>> for row in record["eGQueryResult"]:
...     if row["DbName"] == "pubmed":
...         print(row["Count"])
... 463
```

Now we use the Bio.Entrez.efetch function to download the PubMed IDs of these 463 articles:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="pubmed", term="orchid", retmax=463)
>>> record = Entrez.read(stream)
>>> stream.close()
>>> idlist = record["IdList"]
```

This returns a Python list containing all of the PubMed IDs of articles related to orchids:
Now that we've got them, we obviously want to get the corresponding Medline records and extract the information from them. Here, we'll download the Medline records in the Medline flat-file format, and use the `Bio.Medline` module to parse them:

```python
>>> from Bio import Medline
>>> stream = Entrez.efetch(db="pubmed", id=idlist, rettype="medline", retmode="text")
>>> records = Medline.parse(stream)
```

NOTE - We've just done a separate search and fetch here, the NCBI much prefer you to take advantage of their history support in this situation. See Section 12.16.

Keep in mind that `records` is an iterator, so you can iterate through the records only once. If you want to save the records, you can convert them to a list:

```python
>>> records = list(records)
```

Let's now iterate over the records to print out some information about each record:

```python
>>> for record in records:
...     print("title:", record.get("TI", "?"))
...     print("authors:", record.get("AU", "?"))
...     print("source:", record.get("SO", "?"))
...     print(""")
```

The output for this looks like:

title: Sex pheromone mimicry in the early spider orchid (ophrys sphegodes): patterns of hydrocarbons as the key mechanism for pollination by sexual deception [In Process Citation]
authors: ['Schiestl FP', 'Ayasse M', 'Paulus HF', 'Lofstedt C', 'Hansson BS', 'Ibarra F', 'Francke W']

Especially interesting to note is the list of authors, which is returned as a standard Python list. This makes it easy to manipulate and search using standard Python tools. For instance, we could loop through a whole bunch of entries searching for a particular author with code like the following:

```python
>>> search_author = "Waits T"
>>> for record in records:
...     if not "AU" in record:
...         continue
...     if search_author in record["AU"]:
...         print("Author %s found: %s" % (search_author, record["SO"]))
...```

Hopefully this section gave you an idea of the power and flexibility of the Entrez and Medline interfaces and how they can be used together.
12.15.2 Searching, downloading, and parsing Entrez Nucleotide records

Here we’ll show a simple example of performing a remote Entrez query. In section 2.3 of the parsing examples, we talked about using NCBI’s Entrez website to search the NCBI nucleotide databases for info on Cypripedioideae, our friends the lady slipper orchids. Now, we’ll look at how to automate that process using a Python script. In this example, we’ll just show how to connect, get the results, and parse them, with the Entrez module doing all of the work.

First, we use EGQuery to find out the number of results we will get before actually downloading them. EGQuery will tell us how many search results were found in each of the databases, but for this example we are only interested in nucleotides:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.egquery(term="Cypripedioideae")
>>> record = Entrez.read(stream)
>>> for row in record["eGQueryResult"]:
...    if row["DbName"] == "nuccore":
...        print(row["Count"])
...        4457
```

So, we expect to find 4457 Entrez Nucleotide records (this increased from 814 records in 2008; it is likely to continue to increase in the future). If you find some ridiculously high number of hits, you may want to reconsider if you really want to download all of them, which is our next step. Let’s use the retmax argument to restrict the maximum number of records retrieved to the number available in 2008:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="nucleotide", term="Cypripedioideae", retmax=814, idtype="acc")
>>> record = Entrez.read(stream)
>>> stream.close()
```

Here, record is a Python dictionary containing the search results and some auxiliary information. Just for information, let’s look at what is stored in this dictionary:

```python
>>> print(record.keys())
['Count', 'RetMax', 'IdList', 'TranslationSet', 'RetStart', 'QueryTranslation']
```

First, let’s check how many results were found:

```python
>>> print(record["Count"])
'4457'
```

You might have expected this to be 814, the maximum number of records we asked to retrieve. However, Count represents the total number of records available for that search, not how many were retrieved. The retrieved records are stored in record["IdList"], which should contain the total number we asked for:

```python
>>> len(record["IdList"])
814
```

Let’s look at the first five results:

```python
>>> record["IdList"][::5]
['KX265015.1', 'KX265014.1', 'KX265013.1', 'KX265012.1', 'KX265011.1']
```
We can download these records using `efetch`. While you could download these records one by one, to reduce the load on NCBI’s servers, it is better to fetch a bunch of records at the same time, shown below. However, in this situation you should ideally be using the history feature described later in Section 12.16.

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

Each of these records corresponds to one GenBank record.

```python
>>> print(records[0].keys())
['GBSeq_moltype', 'GBSeq_source', 'GBSeq_sequence',
 'GBSeq_primary-accession', 'GBSeq_definition', 'GBSeq_accession-version',
 'GBSeq_topology', 'GBSeq_length', 'GBSeq_feature-table',
 'GBSeq_create-date', 'GBSeq_other-seqids', 'GBSeq_division',
 'GBSeq_taxonomy', 'GBSeq_references', 'GBSeq_update-date',
 'GBSeq_organism', 'GBSeq_locus', 'GBSeq_strandedness']
```

```python
>>> print(records[0]['GBSeq_primary-accession'])
DQ110336
```

```python
>>> print(records[0]['GBSeq_other-seqids'])
['gb|DQ110336.1|', 'gi|187237168']
```

```python
>>> print(records[0]['GBSeq_definition'])
Cypripedium calceolus voucher Davis 03-03 A maturase (matR) gene, partial cds; mitochondrial
```

```python
>>> print(records[0]['GBSeq_organism'])
Cypripedium calceolus
```

You could use this to quickly set up searches – but for heavy usage, see Section 12.16.

### 12.15.3 Searching, downloading, and parsing GenBank records

The GenBank record format is a very popular method of holding information about sequences, sequence features, and other associated sequence information. The format is a good way to get information from the NCBI databases at [https://www.ncbi.nlm.nih.gov/](https://www.ncbi.nlm.nih.gov/).

In this example we’ll show how to query the NCBI databases, to retrieve the records from the query, and then parse them using `Bio.SeqIO` - something touched on in Section 5.3.1. For simplicity, this example does not take advantage of the WebEnv history feature – see Section 12.16 for this.

First, we want to make a query and find out the ids of the records to retrieve. Here we’ll do a quick search for one of our favorite organisms, *Opuntia* (prickly-pear cacti). We can do quick search and get back the GIs (GenBank identifiers) for all of the corresponding records. First we check how many records there are:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.egquery(term="Opuntia AND rpl16")
```
Now we download the list of GenBank identifiers:

>>> stream = Entrez.esearch(db="nuccore", term="Opuntia AND rpl16")
>>> record = Entrez.read(stream)
>>> gi_list = record["IdList"]
>>> gi_list
['57240072', '57240071', '6273287', '6273291', '6273290', '6273289', '6273286', '6273285', '6273284']

Now we use these GIs to download the GenBank records - note that with older versions of Biopython
you had to supply a comma separated list of GI numbers to Entrez, as of Biopython 1.59 you can pass a list
and this is converted for you:

>>> gi_str = ",".join(gi_list)
>>> stream = Entrez.efetch(db="nuccore", id=gi_str, rettype="gb", retmode="text")

If you want to look at the raw GenBank files, you can read from this stream and print out the result:

>>> text = stream.read()
>>> print(text)
LOCUS AY851612 892 bp DNA linear PLN 10-APR-2007
DEFINITION Opuntia subulata rpl16 gene, intron; chloroplast.
ACCESSION AY851612
VERSION AY851612.1 GI:57240072
KEYWORDS .
SOURCE chloroplast Austrocylindropuntia subulata
ORGANISM Austrocylindropuntia subulata
  Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;
  Spermatophyta; Magnoliophyta; eudicotyledons; core eudicotyledons;
  Caryophyllales; Cactaceae; Opuntioideae; Austrocylindropuntia.
REFERENCE 1 (bases 1 to 892)
AUTHORS Butterworth,C.A. and Wallace,R.S.
...

In this case, we are just getting the raw records. To get the records in a more Python-friendly form, we
can use Bio.SeqIO to parse the GenBank data into SeqRecord objects, including SeqFeature objects (see
Chapter 5):

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

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

>>> for record in records:
...     print(f"{record.name}, length {len(record)}, with {len(record.features)} features")
...
Using these automated query retrieval functionality is a big plus over doing things by hand. Although the module should obey the NCBI's max three queries per second rule, the NCBI have other recommendations like avoiding peak hours. See Section 12.1. In particular, please note that for simplicity, this example does not use the WebEnv history feature. You should use this for any non-trivial search and download work, see Section 12.16.

Finally, if plan to repeat your analysis, rather than downloading the files from the NCBI and parsing them immediately (as shown in this example), you should just download the records once and save them to your hard disk, and then parse the local file.

12.15.4 Finding the lineage of an organism

Staying with a plant example, let’s now find the lineage of the Cypripedioideae orchid family. First, we search the Taxonomy database for Cypripedioideae, which yields exactly one NCBI taxonomy identifier:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="Taxonomy", term="Cypripedioideae")
>>> record = Entrez.read(stream)
>>> record["IdList"]
['158330']
>>> record["IdList"][0]
'158330'
```

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

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

Again, this record stores lots of information:

```python
>>> records[0].keys()
['Lineage', 'Division', 'ParentTaxId', 'PubDate', 'LineageEx', 'CreateDate', 'TaxId', 'Rank', 'GeneticCode', 'ScientificName', 'MitoGeneticCode', 'UpdateDate']
```

We can get the lineage directly from this record:

```python
>>> records[0]["Lineage"]
'cellular organisms; Eukaryota; Viridiplantae; Streptophyta; Streptophytina; Embryophyta; Tracheophyta; Euphylyophyta; Spermatophyta; Magnoliopsida; Liliopsida; Asparagales; Orchidaceae'
```

The record data contains much more than just the information shown here - for example look under "LineageEx" instead of "Lineage" and you’ll get the NCBI taxon identifiers of the lineage entries too.
12.16 Using the history and WebEnv

Often you will want to make a series of linked queries. Most typically, running a search, perhaps refining the search, and then retrieving detailed search results. You can do this by making a series of separate calls to Entrez. However, the NCBI prefer you to take advantage of their history support - for example combining ESearch and EFetch.

Another typical use of the history support would be to combine EPost and EFetch. You use EPost to upload a list of identifiers, which starts a new history session. You then download the records with EFetch by referring to the session (instead of the identifiers).

12.16.1 Searching for and downloading sequences using the history

Suppose we want to search and download all the *Opuntia* rpl16 nucleotide sequences, and store them in a FASTA file. As shown in Section 12.15.3, we can naively combine Bio.Entrez.esearch() to get a list of Accession numbers, and then call Bio.Entrez.efetch() to download them all.

However, the approved approach is to run the search with the history feature. Then, we can fetch the results by reference to the search results - which the NCBI can anticipate and cache.

To do this, call Bio.Entrez.esearch() as normal, but with the additional argument of usehistory="y",

```python
>>> from Bio import Entrez
>>> Entrez.email = "history.user@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(
...     db="nucleotide", term="Opuntia[orgn] and rpl16", usehistory="y", idtype="acc"
... )
>>> search_results = Entrez.read(stream)
>>> stream.close()
```

As before (see Section 12.15.2), the XML output includes the first retmax search results, with retmax defaulting to 20:

```python
>>> acc_list = search_results["IdList"]
>>> count = int(search_results["Count"])
>>> len(acc_list)
20

>>> count
28
```

You also get given two additional pieces of information, the WebEnv session cookie, and the QueryKey:

```python
>>> webenv = search_results["WebEnv"]
>>> query_key = search_results["QueryKey"]
```

Having stored these values in variables session_cookie and query_key we can use them as parameters to Bio.Entrez.efetch() instead of giving the GI numbers as identifiers.

While for small searches you might be OK downloading everything at once, it is better to download in batches. You use the retstart and retmax parameters to specify which range of search results you want returned (starting entry using zero-based counting, and maximum number of results to return). Note that if Biopython encounters a transient failure like a HTTP 500 response when communicating with NCBI, it will automatically try again a couple of times. For example,

```python
# This assumes you have already run a search as shown above,
# and set the variables count, webenv, query_key
```
batch_size = 3
output = open("orchid_rpl16.fasta", "w")
for start in range(0, count, batch_size):
    end = min(count, start + batch_size)
    print("Going to download record %i to %i" % (start + 1, end))
    stream = Entrez.efetch(
        db="nucleotide",
        rettype="fasta",
        retmode="text",
        retstart=start,
        retmax=batch_size,
        webenv=webenv,
        query_key=query_key,
        idtype="acc",
    )
    data = stream.read()
    stream.close()
    output.write(data)
output.close()

For illustrative purposes, this example downloaded the FASTA records in batches of three. Unless you are downloading genomes or chromosomes, you would normally pick a larger batch size.

12.16.2 Searching for and downloading abstracts using the history

Here is another history example, searching for papers published in the last year about the Opuntia, and then downloading them into a file in MedLine format:

```python
from Bio import Entrez
Entrez.email = "history.user@example.com"
search_results = Entrez.read(
    Entrez.esearch(
        db="pubmed", term="Opuntia[ORGN]", reldate=365, datetype="pdat", usehistory="y"
    )
)
count = int(search_results["Count"])
print("Found %i results" % count)
batch_size = 10
output = open("recent_orchid_papers.txt", "w")
for start in range(0, count, batch_size):
    end = min(count, start + batch_size)
    print("Going to download record %i to %i" % (start + 1, end))
    stream = Entrez.efetch(
        db="pubmed",
        rettype="medline",
        retmode="text",
        retstart=start,
        retmax=batch_size,
        webenv=search_results["WebEnv"],
        query_key=search_results["QueryKey"],
```

265
data = stream.read()
stream.close()
output.write(data)
output.close()

At the time of writing, this gave 28 matches - but because this is a date dependent search, this will of course vary. As described in Section 12.13.1 above, you can then use Bio.Medline to parse the saved records.

12.16.3 Searching for citations

Back in Section 12.7 we mentioned ELink can be used to search for citations of a given paper. Unfortunately this only covers journals indexed for PubMed Central (doing it for all the journals in PubMed would mean a lot more work for the NIH). Let’s try this for the Biopython PDB parser paper, PubMed ID 14630660:

```python
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> pmid = "14630660"
>>> results = Entrez.read(
... Entrez.elink(dbfrom="pubmed", db="pmc", LinkName="pubmed_pmc_refs", id=pmid)
... )
>>> pmc_ids = [link["Id"] for link in results[0]["LinkSetDb"][0]["Link"]]
>>> pmc_ids
[‘2744707’, ‘2705363’, ‘2682512’, ..., ‘1190160’]
```

Great - eleven articles. But why hasn’t the Biopython application note been found (PubMed ID 19304878)? Well, as you might have guessed from the variable names, there are not actually PubMed IDs, but PubMed Central IDs. Our application note is the third citing paper in that list, PMCID 2682512.

So, what if (like me) you’d rather get back a list of PubMed IDs? Well we can call ELink again to translate them. This becomes a two step process, so by now you should expect to use the history feature to accomplish it (Section 12.16).

But first, taking the more straightforward approach of making a second (separate) call to ELink:

```python
>>> results2 = Entrez.read(
... Entrez.elink(dbfrom="pmc", db="pubmed", LinkName="pmc_pubmed", id="\"".join(pmc_ids))
... )
>>> pubmed_ids = [link["Id"] for link in results2[0]["LinkSetDb"][0]["Link"]]
>>> pubmed_ids
[‘19698094’, ‘19450287’, ‘19304878’, ..., ‘15985178’]
```

This time you can immediately spot the Biopython application note as the third hit (PubMed ID 19304878). Now, let’s do that all again but with the history ... TODO.

And finally, don’t forget to include your own email address in the Entrez calls.
Chapter 13

Swiss-Prot and ExPASy

13.1 Parsing Swiss-Prot files

Swiss-Prot ([https://web.expasy.org/docs/swiss-prot_guideline.html](https://web.expasy.org/docs/swiss-prot_guideline.html)) is a hand-curated database of protein sequences. Biopython can parse the “plain text” Swiss-Prot file format, which is still used for the UniProt Knowledgebase which combined Swiss-Prot, TrEMBL and PIR-PSD.

Although in the following we focus on the older human readable plain text format, Bio.SeqIO can read both this and the newer UniProt XML file format for annotated protein sequences.

13.1.1 Parsing Swiss-Prot records

In Section 5.3.2, we described how to extract the sequence of a Swiss-Prot record as a SeqRecord object. Alternatively, you can store the Swiss-Prot record in a Bio.SwissProt.Record object, which in fact stores the complete information contained in the Swiss-Prot record. In this section, we describe how to extract Bio.SwissProt.Record objects from a Swiss-Prot file.

To parse a Swiss-Prot record, we first get a handle to a Swiss-Prot record. There are several ways to do so, depending on where and how the Swiss-Prot record is stored:

- Open a Swiss-Prot file locally:

  ```
  >>> handle = open("SwissProt/F2CXE6.txt")
  ```

- Open a gzipped Swiss-Prot file:

  ```
  >>> import gzip
  >>> handle = gzip.open("myswissprotfile.dat.gz", "rt")
  ```

- Open a Swiss-Prot file over the internet:

  ```
  >>> from urllib.request import urlopen
  >>> url = "https://raw.githubusercontent.com/biopython/biopython/master/Tests/SwissProt/F2CXE6.txt"
  >>> handle = urlopen(url)
  ```

To open the file stored on the Internet before calling read.

- Open a Swiss-Prot file over the internet from the ExPASy database (see section 13.5.1):

  ```
  >>> from Bio import ExPASy
  >>> handle = ExPASy.get_sprot_raw("F2CXE6")
  ```
The key point is that for the parser, it doesn’t matter how the handle was created, as long as it points to data in the Swiss-Prot format. The parser will automatically decode the data as ASCII (the encoding used by Swiss-Prot) if the handle was opened in binary mode.

We can use Bio.SeqIO as described in Section 5.3.2 to get file format agnostic SeqRecord objects. Alternatively, we can use Bio.SwissProt get Bio.SwissProt.Record objects, which are a much closer match to the underlying file format.

To read one Swiss-Prot record from the handle, we use the function read():

```python
>>> from Bio import SwissProt
>>> record = SwissProt.read(handle)
```

This function should be used if the handle points to exactly one Swiss-Prot record. It raises a ValueError if no Swiss-Prot record was found, and also if more than one record was found.

We can now print out some information about this record:

```python
>>> print(record.description)
SubName: Full=Plasma membrane intrinsic protein {ECO:0000313|EMBL:BAN04711.1}; SubName: Full=Predicted protein {ECO:0000313|EMBL:BAJ87517.1};

>>> for ref in record.references:
... print("authors:", ref.authors)
... print("title:", ref.title)
...
title: Comprehensive sequence analysis of 24,783 barley full-length cDNAs derived from 12 clone libraries.

authors: Shibasaka M., Sasano S., Utsugi S., Katsuhara M.
authors: Shibasaka M., Katsuhara M., Sasano S.
title:
```

To parse a file that contains more than one Swiss-Prot record, we use the parse function instead. This function allows us to iterate over the records in the file.

For example, let’s parse the full Swiss-Prot database and collect all the descriptions. You can download this from the ExPASy FTP site as a single gzipped-file uniprot_sprot.dat.gz (about 300MB). This is a compressed file containing a single file, uniprot_sprot.dat (over 1.5GB).

As described at the start of this section, you can use the Python library gzip to open and uncompress a.gz file, like this:

```python
>>> import gzip
>>> handle = gzip.open("uniprot_sprot.dat.gz", "rt")
```

However, uncompressing a large file takes time, and each time you open the file for reading in this way, it has to be decompressed on the fly. So, if you can spare the disk space you’ll save time in the long run if you first decompress the file to disk, to get the uniprot_sprot.dat file inside. Then you can open the file for reading as usual:

```python
>>> handle = open("uniprot_sprot.dat")
```

As of June 2009, the full Swiss-Prot database downloaded from ExPASy contained 468851 Swiss-Prot records. One concise way to build up a list of the record descriptions is with a list comprehension:

```python
>>> from Bio import SwissProt
>>> handle = open("uniprot_sprot.dat")
>>> descriptions = [record.description for record in SwissProt.parse(handle)]
```

>>> len(descriptions)
468851

>>> descriptions[:5]
[{'RecName': Full=Protein MGF 100-1R'},
 'RecName': Full=Protein MGF 100-1R',
 'RecName': Full=Protein MGF 100-1R',
 'RecName': Full=Protein MGF 100-1R',
 'RecName': Full=Protein MGF 100-2L;']

Or, using a for loop over the record iterator:

>>> from Bio import SwissProt
>>> descriptions = []
>>> handle = open("uniprot_sprot.dat")
>>> for record in SwissProt.parse(handle):
...     descriptions.append(record.description)
...

Because this is such a large input file, either way takes about eleven minutes on my new desktop computer (using the uncompressed uniprot_sprot.dat file as input).

It is equally easy to extract any kind of information you’d like from Swiss-Prot records. To see the members of a Swiss-Prot record, use

>>> dir(record)
['__doc__', '__init__', '__module__', 'accessions', 'annotation_update',
 'comments', 'created', 'cross_references', 'data_class', 'description',
 'entry_name', 'features', 'gene_name', 'host_organism', 'keywords',
 'molecule_type', 'organelle', 'organism', 'organism_classification',
 'references', 'seqinfo', 'sequence', 'sequence_length',
 'sequence_update', 'taxonomy_id']

13.1.2 Parsing the Swiss-Prot keyword and category list

Swiss-Prot also distributes a file keywlist.txt, which lists the keywords and categories used in Swiss-Prot. The file contains entries in the following form:

ID  2Fe-2S.
AC  KW-0001
DE  Protein which contains at least one 2Fe-2S iron-sulfur cluster: 2 iron
de  atoms complexed to 2 inorganic sulfides and 4 sulfur atoms of
de  cysteines from the protein.
SY  Fe2S2; [2Fe-2S] cluster; [Fe2S2] cluster; Fe2/S2 (inorganic) cluster;
SY  Di-mu-sulfido-diiron; 2 iron, 2 sulfur cluster binding.
GO  GO:0051537; 2 iron, 2 sulfur cluster binding
HI  Ligand: Iron; Iron-sulfur; 2Fe-2S.
HI  Ligand: Metal-binding; 2Fe-2S.
CA  Ligand.
//
ID  3D-structure.
AC  KW-0002

269
DE Protein, or part of a protein, whose three-dimensional structure has
DE been resolved experimentally (for example by X-ray crystallography or
DE NMR spectroscopy) and whose coordinates are available in the PDB
DE database. Can also be used for theoretical models.
HI Technical term: 3D-structure.
CA Technical term.
//
ID 3Fe-4S.
...

The entries in this file can be parsed by the parse function in the Bio.SwissProt.KeyWList module. Each entry is then stored as a Bio.SwissProt.KeyWList.Record, which is a Python dictionary.

```python
>>> from Bio.SwissProt import KeyWList
>>> handle = open("keylist.txt")
>>> records = KeyWList.parse(handle)
>>> for record in records:
... print(record["ID"])
... print(record["DE"])
...
```

This prints

2Fe-2S.
Protein which contains at least one 2Fe-2S iron-sulfur cluster: 2 iron atoms complexed to 2 inorganic sulfides and 4 sulfur atoms of cysteines from the protein.
...

13.2 Parsing Prosite records

Prosite is a database containing protein domains, protein families, functional sites, as well as the patterns and profiles to recognize them. Prosite was developed in parallel with Swiss-Prot. In Biopython, a Prosite record is represented by the Bio.ExPASy.Prosite.Record class, whose members correspond to the different fields in a Prosite record.

In general, a Prosite file can contain more than one Prosite records. For example, the full set of Prosite records, which can be downloaded as a single file (prosite.dat) from the ExPASy FTP site, contains 2073 records (version 20.24 released on 4 December 2007). To parse such a file, we again make use of an iterator:

```python
>>> from Bio.ExPASy import Prosite
>>> handle = open("myprositefile.dat")
>>> records = Prosite.parse(handle)
```

We can now take the records one at a time and print out some information. For example, using the file containing the complete Prosite database, we’d find

```python
>>> from Bio.ExPASy import Prosite
>>> handle = open("prosite.dat")
>>> records = Prosite.parse(handle)
>>> record = next(records)
>>> record.accession
'PS00001'
>>> record.name
270
```
'ASN_GLYCOSYLATION'

```python
>>> record.pdoc
'PDOC000001'
>>> record = next(records)
>>> record.accession
'PS00004'
>>> record.name
'CAMP_PHOSPHO_SITE'
>>> record.pdoc
'PDOC00004'
>>> record = next(records)
>>> record.accession
'PS00005'
>>> record.name
'PKC_PHOSPHO_SITE'
>>> record.pdoc
'PDOC00005'
```

and so on. If you’re interested in how many Prosite records there are, you could use

```python
>>> from Bio.ExPASy import Prosite
>>> handle = open("prosite.dat")
>>> records = Prosite.parse(handle)
>>> n = 0
>>> for record in records:
...     n += 1
... n
2073
```

To read exactly one Prosite from the handle, you can use the `read()` function:

```python
>>> from Bio.ExPASy import Prosite
>>> handle = open("mysingleprositerecord.dat")
>>> record = Prosite.read(handle)
```

This function raises a ValueError if no Prosite record is found, and also if more than one Prosite record is found.

### 13.3 Parsing Prosite documentation records

In the Prosite example above, the `record.pdoc` accession numbers 'PDOC00001', 'PDOC00004', 'PDOC00005' and so on refer to Prosite documentation. The Prosite documentation records are available from ExPASy as individual files, and as one file (`prosite.doc`) containing all Prosite documentation records.

We use the parser in `Bio.ExPASy.Prodoc` to parse Prosite documentation records. For example, to create a list of all accession numbers of Prosite documentation record, you can use

```python
>>> from Bio.ExPASy import Prodoc
>>> handle = open("prosite.doc")
>>> records = Prodoc.parse(handle)
>>> accessions = [record.accession for record in records]
```

Again a `read()` function is provided to read exactly one Prosite documentation record from the handle.
13.4 Parsing Enzyme records

ExPASy’s Enzyme database is a repository of information on enzyme nomenclature. A typical Enzyme record looks as follows:

ID 3.1.1.34
DE Lipoprotein lipase.
AN Clearing factor lipase.
AN Diacylglycerol lipase.
AN Diglyceride lipase.
CA Triacylglycerol + H(2)O = diacylglycerol + a carboxylate.
CC -!- Hydrolyzes triacylglycerols in chylomicrons and very low-density lipoproteins (VLDL).
CC -!- Also hydrolyzes diacylglycerol.
PR PROSITE; PDOC00110;
DR P11151, LIPL_BOVIN ; P11153, LIPL_CAVPO ; P11602, LIPL_CHICK ;
DR P55031, LIPL_FELCA ; P06858, LIPL_HUMAN ; P11152, LIPL_MOUSE ;
DR 046647, LIPL_MUSVI ; P49060, LIPL_PAPAN ; P49923, LIPL_PIG ;
DR Q06000, LIPL_RAT ; Q29524, LIPL_SHEEP ;

In this example, the first line shows the EC (Enzyme Commission) number of lipoprotein lipase (second line). Alternative names of lipoprotein lipase are "clearing factor lipase", "diacylglycerol lipase", and "diglyceride lipase" (lines 3 through 5). The line starting with "CA" shows the catalytic activity of this enzyme. Comment lines start with "CC". The "PR" line shows references to the Prosite Documentation records, and the "DR" lines show references to Swiss-Prot records. Not all of these entries are necessarily present in an Enzyme record.

In Biopython, an Enzyme record is represented by the Bio.ExPASy.Enzyme.Record class. This record derives from a Python dictionary and has keys corresponding to the two-letter codes used in Enzyme files. To read an Enzyme file containing one Enzyme record, use the read function in Bio.ExPASy.Enzyme:

```python
>>> from Bio.ExPASy import Enzyme
>>> with open("lipoprotein.txt") as handle:
... record = Enzyme.read(handle)
... >>> record["ID"]
'3.1.1.34'
>>> record["DE"]
'Lipoprotein lipase.'
>>> record["AN"]
['Clearing factor lipase.', 'Diacylglycerol lipase.', 'Diglyceride lipase.]
>>> record["CA"]
'Triacylglycerol + H(2)O = diacylglycerol + a carboxylate.'
>>> record["PR"]
['PROSITE; PDOC00110']

>>> record["CC"]
['Hydrolyzes triacylglycerols in chylomicrons and very low-density lipoproteins (VLDL).', 'Also hydrolyzes diacylglycerol.]
>>> record["DR"]
[['P11151', 'LIPL_BOVIN'], ['P11153', 'LIPL_CAVPO'], ['P11602', 'LIPL_CHICK'], ['P55031', 'LIPL_FELCA'], ['P06858', 'LIPL_HUMAN'], ['P11152', 'LIPL_MOUSE'], ['046647', 'LIPL_MUSVI'], ['P49060', 'LIPL_PAPAN'], ['P49923', 'LIPL_PIG'], ['Q06000', 'LIPL_RAT'], ['Q29524', 'LIPL_SHEEP']]
```
The `read` function raises a `ValueError` if no Enzyme record is found, and also if more than one Enzyme record is found.

The full set of Enzyme records can be downloaded as a single file (`enzyme.dat`) from the ExPASy FTP site, containing 4877 records (release of 3 March 2009). To parse such a file containing multiple Enzyme records, use the `parse` function in `Bio.ExPASy.Enzyme` to obtain an iterator:

```python
>>> from Bio.ExPASy import Enzyme
>>> handle = open("enzyme.dat")
>>> records = Enzyme.parse(handle)
```

We can now iterate over the records one at a time. For example, we can make a list of all EC numbers for which an Enzyme record is available:

```python
>>> ecnumbers = [record["ID"] for record in records]
```

### 13.5 Accessing the ExPASy server

Swiss-Prot, Prosite, and Prosite documentation records can be downloaded from the ExPASy web server at [https://www.expasy.org](https://www.expasy.org). Four kinds of queries are available from ExPASy:

- `get_prodoc_entry` To download a Prosite documentation record in HTML format
- `get_prosite_entry` To download a Prosite record in HTML format
- `get_prosite_raw` To download a Prosite or Prosite documentation record in raw format
- `get_sprot_raw` To download a Swiss-Prot record in raw format

To access this web server from a Python script, we use the `Bio.ExPASy` module.

#### 13.5.1 Retrieving a Swiss-Prot record

Let’s say we are looking at chalcone synthases for Orchids (see section 2.3 for some justification for looking for interesting things about orchids). Chalcone synthase is involved in flavanoid biosynthesis in plants, and flavanoids make lots of cool things like pigment colors and UV protectants.

If you do a search on Swiss-Prot, you can find three orchid proteins for Chalcone Synthase, id numbers O23729, O23730, O23731. Now, let’s write a script which grabs these, and parses out some interesting information.

First, we grab the records, using the `get_sprot_raw()` function of `Bio.ExPASy`. This function is very nice since you can feed it an id and get back a handle to a raw text record (no HTML to mess with!). We can use `Bio.SwissProt.read` to pull out the Swiss-Prot record, or `Bio.SeqIO.read` to get a `SeqRecord`. The following code accomplishes what I just wrote:

```python
>>> from Bio import ExPASy
>>> from Bio import SwissProt

>>> accessions = ["O23729", "O23730", "O23731"]
>>> records = []

>>> for accession in accessions:
...     handle = ExPASy.get_sprot_raw(accession)
...     record = SwissProt.read(handle)
...     records.append(record)
...
If the accession number you provided to `ExPASy.get_sprot_raw` does not exist, then `SwissProt.read(handle)` will raise a `ValueError`. You can catch `ValueException` exceptions to detect invalid accession numbers:

```python
generate code for the python code
```

### 13.5.2 Searching Swiss-Prot

Now, you may remark that I knew the records’ accession numbers beforehand. Indeed, `get_sprot_raw()` needs either the entry name or an accession number. When you don’t have them handy, right now you could use [https://www.uniprot.org/](https://www.uniprot.org/) but we do not have a Python wrapper for searching this from a script. Perhaps you could contribute here?

### 13.5.3 Retrieving Prosite and Prosite documentation records

Prosite and Prosite documentation records can be retrieved either in HTML format, or in raw format. To parse Prosite and Prosite documentation records with Biopython, you should retrieve the records in raw format. For other purposes, however, you may be interested in these records in HTML format.

To retrieve a Prosite or Prosite documentation record in raw format, use `get_prosite_raw()`. For example, to download a Prosite record and print it out in raw text format, use

```python
generate code for the python code
```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prosite_entry("PS00001")
>>> html = handle.read()
>>> with open("myprositeresult.html", "w") as out_handle:
...     out_handle.write(html)

... and similarly for a Prosite documentation record:

>>> from Bio import ExPASy
>>> handle = ExPASy.get_prodoc_entry("PDOC00001")
>>> html = handle.read()
>>> with open("myprodocrecord.html", "w") as out_handle:
...     out_handle.write(html)

... For these functions, an invalid accession number returns an error message in HTML format.

13.6 Scanning the Prosite database

ScanProsite allows you to scan protein sequences online against the Prosite database by providing a UniProt
or PDB sequence identifier or the sequence itself. For more information about ScanProsite, please see the
ScanProsite documentation as well as the documentation for programmatic access of ScanProsite.

You can use Biopython's Bio.ExPASy.ScanProsite module to scan the Prosite database from Python. This module both helps you to access ScanProsite programmatically, and to parse the results returned by
ScanProsite. To scan for Prosite patterns in the following protein sequence:

MEHKEVLLLLLFLKSGQGEPLDDYVNTQGASLFSVTKQLGAGSIEEAKEEDEEFT
CRAFQYHKEEQCVIMAENRKSIIIRMREDVVLFEKKVVLSECKTNGKNYRTMSKTKN

you can use the following code:

>>> sequence = ("MEHKEVLLLLLFLKSGQGEPLDDYVNTQGASLFSVTKQLGAGSIEEAKEEDEEFT"
...     "CRAFQYHKEEQCVIMAENRKSIIIRMREDVVLFEKKVVLSECKTNGKNYRTMSKTKN")
>>> from Bio.ExPASy import ScanProsite
>>> handle = ScanProsite.scan(seq=sequence)

By executing handle.read(), you can obtain the search results in raw XML format. Instead, let’s use
Bio.ExPASy.ScanProsite.read to parse the raw XML into a Python object:

>>> result = ScanProsite.read(handle)
>>> type(result)
<class 'Bio.ExPASy.ScanProsite.Record'>

A Bio.ExPASy.ScanProsite.Record object is derived from a list, with each element in the list storing
one ScanProsite hit. This object also stores the number of hits, as well as the number of search sequences,
as returned by ScanProsite. This ScanProsite search resulted in six hits:

>>> result.n_seq
1
>>> result.n_match
275
Other ScanProsite parameters can be passed as keyword arguments; see the documentation for programmatic access of ScanProsite for more information. As an example, passing lowscore=1 to include matches with low level scores lets use find one additional hit:

```python
>>> handle = ScanProsite.scan(seq=sequence, lowscore=1)
>>> result = ScanProsite.read(handle)
>>> result.n_match
```

2
Chapter 14

Going 3D: The PDB module

Bio.PDB is a Biopython module that focuses on working with crystal structures of biological macromolecules. Among other things, Bio.PDB includes a PDBParser class that produces a Structure object, which can be used to access the atomic data in the file in a convenient manner. There is limited support for parsing the information contained in the PDB header. PDB file format is no longer being modified or extended to support new content and PDBx/mmCIF became the standard PDB archive format in 2014. All the Worldwide Protein Data Bank (wwPDB) sites uses the macromolecular Crystallographic Information File (mmCIF) data dictionaries to describe the information content of PDB entries. mmCIF uses a flexible and extensible key-value pair format for representing macromolecular structural data and imposes no limitations for the number of atoms, residues or chains that can be represented in a single PDB entry (no split entries!).

14.1 Reading and writing crystal structure files

14.1.1 Reading an mmCIF file

First create an MMCIFParser object:

```python
>>> from Bio.PDB/MMCIFParser import MMCIFParser
>>> parser = MMCIFParser()
```

Then use this parser to create a structure object from the mmCIF file:

```python
>>> structure = parser.get_structure("1fat", "1fat.cif")
```

To have some more low level access to an mmCIF file, you can use the MMCIF2Dict class to create a Python dictionary that maps all mmCIF tags in an mmCIF file to their values. Whether there are multiple values (like in the case of tag _atom_site.Cartn_y, which holds the y coordinates of all atoms) or a single value (like the initial deposition date), the tag is mapped to a list of values. The dictionary is created from the mmCIF file as follows:

```python
>>> from Bio.PDB/MMCIF2Dict import MMCIF2Dict
>>> mmcif_dict = MMCIF2Dict("1FAT.cif")
```

Example: get the solvent content from an mmCIF file:

```python
>>> sc = mmcif_dict["_exptl_crystal.density_percent_sol"]
```

Example: get the list of the y coordinates of all atoms

```python
>>> y_list = mmcif_dict["_atom_site.Cartn_y"]
```
14.1.2 Reading files in the MMTF format

You can use the direct MMTFParser to read a structure from a file:

```python
>>> from Bio.PDB.mmtf import MMTFParser
>>> structure = MMTFParser.get_structure("PDB/4CUP.mmtf")
```

Or you can use the same class to get a structure by its PDB ID:

```python
>>> structure = MMTFParser.get_structure_from_url("4CUP")
```

This gives you a Structure object as if read from a PDB or mmCIF file.

You can also have access to the underlying data using the external MMTF library which Biopython is using internally:

```python
>>> from mmtf import fetch
>>> decoded_data = fetch("4CUP")
```

For example you can access just the X-coordinate.

```python
>>> print(decoded_data.x_coord_list)
```

14.1.3 Reading a PDB file

First we create a PDBParser object:

```python
>>> from Bio.PDB.PDBParser import PDBParser
>>> parser = PDBParser(PERMISSIVE=1)
```

The PERMISSIVE flag indicates that a number of common problems (see 14.7.1) associated with PDB files will be ignored (but note that some atoms and/or residues will be missing). If the flag is not present a PDBConstructionException will be generated if any problems are detected during the parse operation.

The Structure object is then produced by letting the PDBParser object parse a PDB file (the PDB file in this case is called pdb1fat.ent, 1fat is a user defined name for the structure):

```python
>>> structure_id = "1fat"
>>> filename = "pdb1fat.ent"
>>> structure = parser.get_structure(structure_id, filename)
```

You can extract the header and trailer (simple lists of strings) of the PDB file from the PDBParser object with the get_header and get_trailer methods. Note however that many PDB files contain headers with incomplete or erroneous information. Many of the errors have been fixed in the equivalent mmCIF files. Hence, if you are interested in the header information, it is a good idea to extract information from mmCIF files using the MMCIF2Dict tool described above, instead of parsing the PDB header.

Now that is clarified, let’s return to parsing the PDB header. The structure object has an attribute called header which is a Python dictionary that maps header records to their values.

Example:

```python
>>> resolution = structure.header["resolution"]
>>> keywords = structure.header["keywords"]
```

The available keys are name, head, deposition_date, release_date, structure_method, resolution, structure_reference (which maps to a list of references), journal_reference, author, compound (which maps to a dictionary with various information about the crystallized compound), has_missing_residues, missing_residues, and astral (which maps to dictionary with additional information about the domain if present).
has_missing_residues maps to a bool that is True if at least one non-empty REMARK 465 header line was found. In this case you should assume that the molecule used in the experiment has some residues for which no ATOM coordinates could be determined. missing_residues maps to a list of dictionaries with information about the missing residues. The list of missing residues will be empty or incomplete if the PDB header does not follow the template from the PDB specification.

The dictionary can also be created without creating a Structure object, i.e. directly from the PDB file:

```python
>>> from Bio.PDB import parse_pdb_header
>>> with open(filename, "r") as handle:
...    header_dict = parse_pdb_header(handle)
...```

14.1.4 Reading a PQR file

In order to parse a PQR file, proceed in a similar manner as in the case of PDB files:

Create a PDBParser object, using the is_pqr flag:

```python
>>> from Bio.PDB.PDBParser import PDBParser
>>> pqr_parser = PDBParser(PERMISSIVE=1, is_pqr=True)
```

The is_pqr flag set to True indicates that the file to be parsed is a PQR file, and that the parser should read the atomic charge and radius fields for each atom entry. Following the same procedure as for PQR files, a Structure object is then produced, and the PQR file is parsed.

```python
>>> structure_id = "1fat"
>>> filename = "pdb1fat.ent"
>>> structure = parser.get_structure(structure_id, filename, is_pqr=True)
```

14.1.5 Reading files in the PDB XML format

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

14.1.6 Writing mmCIF files

The MMCIFIO class can be used to write structures to the mmCIF file format:

```python
>>> io = MMCIFIO()
>>> io.set_structure(s)
>>> io.save("out.cif")
```

The Select class can be used in a similar way to PDBIO below. mmCIF dictionaries read using MMCIF2Dict can also be written:

```python
>>> io = MMCIFIO()
>>> io.set_dict(d)
>>> io.save("out.cif")
```

14.1.7 Writing PDB files

Use the PDBIO class for this. It’s easy to write out specific parts of a structure too, of course.

Example: saving a structure
>>> io = PDBIO()
>>> io.set_structure(s)
>>> io.save("out.pdb")

If you want to write out a part of the structure, make use of the `Select` class (also in `PDBIO`). Select has four methods:

- `accept_model(model)`
- `accept_chain(chain)`
- `accept_residue(residue)`
- `accept_atom(atom)`

By default, every method returns 1 (which means the model/chain/residue/atom is included in the output). By subclassing `Select` and returning 0 when appropriate you can exclude models, chains, etc. from the output. Cumbersome maybe, but very powerful. The following code only writes out glycine residues:

```python
>>> class GlySelect(Select):
...     def accept_residue(self, residue):
...         if residue.get_name() == "GLY":
...             return True
...         else:
...             return False
...
>>> io = PDBIO()
>>> io.set_structure(s)
>>> io.save("gly_only.pdb", GlySelect())
```

If this is all too complicated for you, the Dice module contains a handy `extract` function that writes out all residues in a chain between a start and end residue.

### 14.1.8 Writing PQR files

Use the `PDBIO` class as you would for a PDB file, with the flag `is_pqr=True`. The PDBIO methods can be used in the case of PQR files as well.

Example: writing a PQR file

```python
>>> io = PDBIO(is_pqr=True)
>>> io.set_structure(s)
>>> io.save("out.pdb")
```

### 14.1.9 Writing MMTF files

To write structures to the MMTF file format:

```python
>>> from Bio.PDB.mmtf import MMTFIO
>>> io = MMTFIO()
>>> io.set_structure(s)
>>> io.save("out.mmtf")
```

The `Select` class can be used as above. Note that the bonding information, secondary structure assignment and some other information contained in standard MMTF files is not written out as it is not easy to determine from the structure object. In addition, molecules that are grouped into the same entity in standard MMTF files are treated as separate entities by `MMTFIO`.

280
14.2 Structure representation

The overall layout of a Structure object follows the so-called SMCRA (Structure/Model/Chain/Residue/Atom) architecture:

- A structure consists of models
- A model consists of chains
- A chain consists of residues
- A residue consists of atoms

This is the way many structural biologists/bioinformaticians think about structure, and provides a simple but efficient way to deal with structure. Additional stuff is essentially added when needed. A UML diagram of the Structure object (forget about the Disordered classes for now) is shown in Figure 14.1. Such a data structure is not necessarily best suited for the representation of the macromolecular content of a structure, but it is absolutely necessary for a good interpretation of the data present in a file that describes the structure (typically a PDB or MMCIF file). If this hierarchy cannot represent the contents of a structure file, it is fairly certain that the file contains an error or at least does not describe the structure unambiguously. If a SMCRA data structure cannot be generated, there is reason to suspect a problem. Parsing a PDB file can thus be used to detect likely problems. We will give several examples of this in section 14.7.1.

Structure, Model, Chain and Residue are all subclasses of the Entity base class. The Atom class only (partly) implements the Entity interface (because an Atom does not have children).

For each Entity subclass, you can extract a child by using a unique id for that child as a key (e.g. you can extract an Atom object from a Residue object by using an atom name string as a key, you can extract a Chain object from a Model object by using its chain identifier as a key).

Disordered atoms and residues are represented by DisorderedAtom and DisorderedResidue classes, which are both subclasses of the DisorderedEntityWrapper base class. They hide the complexity associated with disorder and behave exactly as Atom and Residue objects.

In general, a child Entity object (i.e. Atom, Residue, Chain, Model) can be extracted from its parent (i.e. Residue, Chain, Model, Structure, respectively) by using an id as a key.

```python
>>> child_entity = parent_entity[child_id]
```

You can also get a list of all child Entities of a parent Entity object. Note that this list is sorted in a specific way (e.g. according to chain identifier for Chain objects in a Model object).

```python
>>> child_list = parent_entity.get_list()
```

You can also get the parent from a child:

```python
>>> parent_entity = child_entity.get_parent()
```

At all levels of the SMCRA hierarchy, you can also extract a full id. The full id is a tuple containing all id's starting from the top object (Structure) down to the current object. A full id for a Residue object e.g. is something like:

```python
>>> full_id = residue.get_full_id()
>>> print(full_id)
("1abc", 0, "A", ("", 10, "A"))
```

This corresponds to:

- The Structure with id "1abc"
Figure 14.1: UML diagram of SMCRA architecture of the \texttt{Structure} class used to represent a macromolecular structure. Full lines with diamonds denote aggregation, full lines with arrows denote referencing, full lines with triangles denote inheritance and dashed lines with triangles denote interface realization.
- The Model with id 0
- The Chain with id "A"
- The Residue with id ("", 10, "A")

The Residue id indicates that the residue is not a hetero-residue (nor a water) because it has a blank hetero field, that its sequence identifier is 10 and that its insertion code is "A".

To get the entity's id, use the `get_id` method:

```python
>>> entity.get_id()
```

You can check if the entity has a child with a given id by using the `has_id` method:

```python
>>> entity.has_id(entity_id)
```

The length of an entity is equal to its number of children:

```python
>>> nr_children = len(entity)
```

It is possible to delete, rename, add, etc. child entities from a parent entity, but this does not include any sanity checks (e.g. it is possible to add two residues with the same id to one chain). This really should be done via a nice Decorator class that includes integrity checking, but you can take a look at the code (Entity.py) if you want to use the raw interface.

### 14.2.1 Structure

The Structure object is at the top of the hierarchy. Its id is a user given string. The Structure contains a number of Model children. Most crystal structures (but not all) contain a single model, while NMR structures typically consist of several models. Disorder in crystal structures of large parts of molecules can also result in several models.

### 14.2.2 Model

The id of the Model object is an integer, which is derived from the position of the model in the parsed file (they are automatically numbered starting from 0). Crystal structures generally have only one model (with id 0), while NMR files usually have several models. Whereas many PDB parsers assume that there is only one model, the `Structure` class in Bio.PDB is designed such that it can easily handle PDB files with more than one model.

As an example, to get the first model from a Structure object, use

```python
>>> first_model = structure[0]
```

The Model object stores a list of Chain children.

### 14.2.3 Chain

The id of a Chain object is derived from the chain identifier in the PDB/mmCIF file, and is a single character (typically a letter). Each Chain in a Model object has a unique id. As an example, to get the Chain object with identifier “A” from a Model object, use

```python
>>> chain_A = model["A"]
```

The Chain object stores a list of Residue children.
14.2.4 Residue

A residue id is a tuple with three elements:

- The **hetero-field** (hetfield): this is
  - 'W' in the case of a water molecule;
  - 'H_' followed by the residue name for other hetero residues (e.g. 'H_GLC' in the case of a glucose molecule);
  - blank for standard amino and nucleic acids.

  This scheme is adopted for reasons described in section 14.4.1.

- The **sequence identifier** (resseq), an integer describing the position of the residue in the chain (e.g., 100);

- The **insertion code** (icode); a string, e.g. 'A'. The insertion code is sometimes used to preserve a certain desirable residue numbering scheme. A Ser 80 insertion mutant (inserted e.g. between a Thr 80 and an Asn 81 residue) could e.g. have sequence identifiers and insertion codes as follows: Thr 80 A, Ser 80 B, Asn 81. In this way the residue numbering scheme stays in tune with that of the wild type structure.

The id of the above glucose residue would thus be ('H_GLC', 100, 'A'). If the hetero-flag and insertion code are blank, the sequence identifier alone can be used:

```python
# Full id
>>> residue = chain[(' ', 100, ' ')]
# Shortcut id
>>> residue = chain[100]
```

Unsurprisingly, a Residue object stores a set of Atom children. It also contains a string that specifies the residue name (e.g. "ASN") and the segment identifier of the residue (well known to X-PLOR users, but not used in the construction of the SMCRA data structure).

Let’s look at some examples. Asn 10 with a blank insertion code would have residue id (' ', 10, ' '). Water 10 would have residue id ('W', 10, ' '). A glucose molecule (a hetero residue with residue name GLC) with sequence identifier 10 would have residue id ('H_GLC', 10, ' '). In this way, the three residues (with the same insertion code and sequence identifier) can be part of the same chain because their residue id’s are distinct.

In most cases, the hetflag and insertion code fields will be blank, e.g. (' ', 10, ' '). In these cases, the sequence identifier can be used as a shortcut for the full id:

```python
# use full id
>>> res10 = chain[(' ', 10, ' ')]
# use shortcut
>>> res10 = chain[10]
```

Each Residue object in a Chain object should have a unique id. However, disordered residues are dealt with in a special way, as described in section 14.3.3.

A Residue object has a number of additional methods:

```python
>>> residue.get_resname()  # returns the residue name, e.g. "ASN"
>>> residue.is_disordered() # returns 1 if the residue has disordered atoms
>>> residue.get_segid()    # returns the SEGID, e.g. "CHN1"
>>> residue.has_id(name)   # test if a residue has a certain atom
```

You can use `is_aa(residue)` to test if a Residue object is an amino acid.
14.2.5 Atom

The Atom object stores the data associated with an atom, and has no children. The id of an atom is its atom name (e.g. "OG" for the side chain oxygen of a Ser residue). An Atom id needs to be unique in a Residue. Again, an exception is made for disordered atoms, as described in section 14.3.2.

The atom id is simply the atom name (e.g. 'CA'). In practice, the atom name is created by stripping all spaces from the atom name in the PDB file.

However, in PDB files, a space can be part of an atom name. Often, calcium atoms are called 'CA..' in order to distinguish them from Cα atoms (which are called '.CA.'). In cases were stripping the spaces would create problems (i.e. two atoms called 'CA' in the same residue) the spaces are kept.

In a PDB file, an atom name consists of 4 chars, typically with leading and trailing spaces. Often these spaces can be removed for ease of use (e.g. an amino acid Cα atom is labeled “.CA.” in a PDB file, where the dots represent spaces). To generate an atom name (and thus an atom id) the spaces are removed, unless this would result in a name collision in a Residue (i.e. two Atom objects with the same atom name and id). In the latter case, the atom name including spaces is tried. This situation can e.g. happen when one residue contains atoms with names “.CA.” and “CA..”, although this is not very likely.

The atomic data stored includes the atom name, the atomic coordinates (including standard deviation if present), the B factor (including anisotropic B factors and standard deviation if present), the altloc specifier and the full atom name including spaces. Less used items like the atom element number or the atomic charge sometimes specified in a PDB file are not stored.

To manipulate the atomic coordinates, use the transform method of the Atom object. Use the set_coord method to specify the atomic coordinates directly.

An Atom object has the following additional methods:

>>> a.get_name()  # atom name (spaces stripped, e.g. "CA")
>>> a.get_id()    # id (equals atom name)
>>> a.get_coord() # atomic coordinates
>>> a.get_vector()# atomic coordinates as Vector object
>>> a.get_bfactor()# isotropic B factor
>>> a.get_occupancy()# occupancy
>>> a.get_altloc()# alternative location specifier
>>> a.get_sigatm()# standard deviation of atomic parameters
>>> a.get_siguij()# standard deviation of anisotropic B factor
>>> a.get_anisou()# anisotropic B factor
>>> a.get_fullname()# atom name (with spaces, e.g. ".CA.")

To represent the atom coordinates, siguij, anisotropic B factor and sigatm Numpy arrays are used.

The get_vector method returns a Vector object representation of the coordinates of the Atom object, allowing you to do vector operations on atomic coordinates. Vector implements the full set of 3D vector operations, matrix multiplication (left and right) and some advanced rotation-related operations as well.

As an example of the capabilities of Bio.PDB’s Vector module, suppose that you would like to find the position of a Gly residue’s Cβ atom, if it had one. Rotating the N atom of the Gly residue along the Co-C bond over -120 degrees roughly puts it in the position of a virtual Cβ atom. Here’s how to do it, making use of the rotaxis method (which can be used to construct a rotation around a certain axis) of the Vector module:

# get atom coordinates as vectors
>>> n = residue['N'].get_vector()
>>> c = residue['C'].get_vector()
>>> ca = residue['CA'].get_vector()
# center at origin
>>> n = n - ca
>>> c = c - ca
# find rotation matrix that rotates n
# -120 degrees along the ca-c vector
>>> rot = rotaxis(-pi * 120.0 / 180.0, c)
# apply rotation to ca-n vector
>>> cb_at_origin = n.left_multiply(rot)
# put on top of ca atom
>>> cb = cb_at_origin + ca

This example shows that it’s possible to do some quite nontrivial vector operations on atomic data, which
can be quite useful. In addition to all the usual vector operations (cross (use \( \times \)), and dot (use \( \cdot \)) product,
angle, norm, etc.) and the above mentioned rotaxis function, the Vector module also has methods to
rotate (rotmat) or reflect (refmat) one vector on top of another.

14.2.6 Extracting a specific Atom/Residue/Chain/Model from a Structure

These are some examples:

>>> model = structure[0]
>>> chain = model["A"]
>>> residue = chain[100]
>>> atom = residue["CA"]

Note that you can use a shortcut:

>>> atom = structure[0]["A"][100]["CA"]

14.3 Disorder

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

14.3.1 General approach

Disorder should be dealt with from two points of view: the atom and the residue points of view. In general,
we have tried to encapsulate all the complexity that arises from disorder. If you just want to loop over all
\( \alpha \) atoms, you do not care that some residues have a disordered side chain. On the other hand it should also
be possible to represent disorder completely in the data structure. Therefore, disordered atoms or residues
are stored in special objects that behave as if there is no disorder. This is done by only representing a subset
of the disordered atoms or residues. Which subset is picked (e.g. which of the two disordered OG side chain
atom positions of a Ser residue is used) can be specified by the user.

14.3.2 Disordered atoms

Disordered atoms are represented by ordinary Atom objects, but all Atom objects that represent the same
physical atom are stored in a DisorderedAtom object (see Figure 14.1). Each Atom object in a Disordered-
Atom object can be uniquely indexed using its altloc specifier. The DisorderedAtom object forwards all
uncaught method calls to the selected Atom object, by default the one that represents the atom with the
highest occupancy. The user can of course change the selected Atom object, making use of its altloc specifier.
In this way atom disorder is represented correctly without much additional complexity. In other words, if
you are not interested in atom disorder, you will not be bothered by it.

Each disordered atom has a characteristic altloc identifier. You can specify that a DisorderedAtom object
should behave like the Atom object associated with a specific altloc identifier:
14.3.3 Disordered residues

Common case

The most common case is a residue that contains one or more disordered atoms. This is evidently solved by using DisorderedAtom objects to represent the disordered atoms, and storing the DisorderedAtom object in a Residue object just like ordinary Atom objects. The DisorderedAtom will behave exactly like an ordinary atom (in fact the atom with the highest occupancy) by forwarding all uncaught method calls to one of the Atom objects (the selected Atom object) it contains.

Point mutations

A special case arises when disorder is due to a point mutation, i.e. when two or more point mutants of a polypeptide are present in the crystal. An example of this can be found in PDB structure 1EN2. Since these residues belong to a different residue type (e.g. let’s say Ser 60 and Cys 60) they should not be stored in a single Residue object as in the common case. In this case, each residue is represented by one Residue object, and both Residue objects are stored in a single DisorderedResidue object (see Figure 14.1).

The DisorderedResidue object forwards all uncaught methods to the selected Residue object (by default the last Residue object added), and thus behaves like an ordinary residue. Each Residue object in a DisorderedResidue object can be uniquely identified by its residue name. In the above example, residue Ser 60 would have id “SER” in the DisorderedResidue object, while residue Cys 60 would have id “CYS”. The user can select the active Residue object in a DisorderedResidue object via this id.

Example: suppose that a chain has a point mutation at position 10, consisting of a Ser and a Cys residue. Make sure that residue 10 of this chain behaves as the Cys residue.

```python
>>> residue = chain[10]
>>> residue.disordered_select("CYS")
```

In addition, you can get a list of all Atom objects (ie. all DisorderedAtom objects are 'unpacked' to their individual Atom objects) using the get_unpacked_list method of a (Disordered)Residue object.

14.4 Hetero residues

14.4.1 Associated problems

A common problem with hetero residues is that several hetero and non-hetero residues present in the same chain share the same sequence identifier (and insertion code). Therefore, to generate a unique id for each hetero residue, waters and other hetero residues are treated in a different way.

Remember that Residue object have the tuple (hetfield, resseq, icode) as id. The hetfield is blank (“ ”) for amino and nucleic acids, and a string for waters and other hetero residues. The content of the hetfield is explained below.

14.4.2 Water residues

The hetfield string of a water residue consists of the letter “W”. So a typical residue id for a water is (“W”, 1, “ ”).
14.4.3 Other hetero residues

The hetfield string for other hetero residues starts with “H,” followed by the residue name. A glucose molecule e.g. with residue name “GLC” would have hetfield “H,GLC”. Its residue id could e.g. be ("H,GLC", 1, "").

14.5 Navigating through a Structure object

Parse a PDB file, and extract some Model, Chain, Residue and Atom objects

```python
>>> from Bio.PDB.PDBParser import PDBParser
>>> parser = PDBParser()
>>> structure = parser.get_structure("test", "1fat.pdb")
>>> model = structure[0]
>>> chain = model["A"]
>>> residue = chain[1]
>>> atom = residue["CA"]
```

Iterating through all atoms of a structure

```python
>>> p = PDBParser()
>>> structure = p.get_structure("X", "pdb1fat.ent")
>>> for model in structure:
...     for chain in model:
...         for residue in chain:
...             for atom in residue:
...                 print(atom)
...
```

There is a shortcut if you want to iterate over all atoms in a structure:

```python
>>> atoms = structure.get_atoms()
>>> for atom in atoms:
...     print(atom)
...
```

Similarly, to iterate over all atoms in a chain, use

```python
>>> atoms = chain.get_atoms()
>>> for atom in atoms:
...     print(atom)
...
```

Iterating over all residues of a model

or if you want to iterate over all residues in a model:

```python
>>> residues = model.get_residues()
>>> for residue in residues:
...     print(residue)
...
```

You can also use the `Selection.unfold_entities` function to get all residues from a structure:

```python
>>> res_list = Selection.unfold_entities(structure, "R")
```
or to get all atoms from a chain:

```python
>>> atom_list = Selection.unfold_entities(chain, "A")
```

Obviously, \(A=\text{atom}, \ R=\text{residue}, \ C=\text{chain}, \ M=\text{model}, \ S=\text{structure}\). You can use this to go up in the hierarchy, e.g. to get a list of (unique) Residue or Chain parents from a list of Atoms:

```python
>>> residue_list = Selection.unfold_entities(atom_list, "R")
>>> chain_list = Selection.unfold_entities(atom_list, "C")
```

For more info, see the API documentation.

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

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

Print all hetero residues in chain

```python
>>> for residue in chain.get_list():
...    residue_id = residue.get_id()
...    hetfield = residue_id[0]
...    if hetfield[0] == "H":
...        print(residue_id)
...```

Print out coordinates of all CA atoms in structure with B factor over 50

```python
>>> for model in structure.get_list():
...    for chain in model.get_list():
...        for residue in chain.get_list():
...            if residue.has_id("CA"):
...                ca = residue["CA"]
...                if ca.get_bfactor() > 50.0:
...                    print(ca.get_coord())
...```

Print out all the residues that contain disordered atoms

```python
>>> for model in structure.get_list():
...    for chain in model.get_list():
...        for residue in chain.get_list():
...            if residue.is_disordered():
...                resseq = residue.get_id()[1]
...                resname = residue.get_resname()
...                model_id = model.get_id()
...                chain_id = chain.get_id()
...                print(model_id, chain_id, resname, resseq)
...```
Loop over all disordered atoms, and select all atoms with altloc A (if present)

This will make sure that the SMCRA data structure will behave as if only the atoms with altloc A are present.

```python
>>> for model in structure.get_list():
...   for chain in model.get_list():
...     for residue in chain.get_list():
...       if residue.is_disordered():
...         for atom in residue.get_list():
...           if atom.is_disordered():
...             if atom.disordered_has_id("A"):
...               atom.disordered_select("A")
...```

Extracting polypeptides from a Structure object

To extract polypeptides from a structure, construct a list of Polypeptide objects from a Structure object using PolypeptideBuilder as follows:

```python
>>> model_nr = 1
>>> polypeptide_list = build_peptides(structure, model_nr)
>>> for polypeptide in polypeptide_list:
...   print(polypeptide)
...
```

A Polypeptide object is simply a UserList of Residue objects, and is always created from a single Model (in this case model 1). You can use the resulting Polypeptide object to get the sequence as a Seq object or to get a list of Cα atoms as well. Polypeptides can be built using a C-N or a Cα-Cα distance criterion.

Example:

```
# Using C-N
>>> ppb = PPBuilder()
>>> for pp in ppb.build_peptides(structure):
...   print(pp.get_sequence())
...
# Using CA-CA
>>> ppb = CaPPBuilder()
>>> for pp in ppb.build_peptides(structure):
...   print(pp.get_sequence())
...
```

Note that in the above case only model 0 of the structure is considered by PolypeptideBuilder. However, it is possible to use PolypeptideBuilder to build Polypeptide objects from Model and Chain objects as well.

Obtaining the sequence of a structure

The first thing to do is to extract all polypeptides from the structure (as above). The sequence of each polypeptide can then easily be obtained from the Polypeptide objects. The sequence is represented as a Biopython Seq object.

Example:

```python
>>> seq = polypeptide.get_sequence()
>>> seq
Seq('SNDIYFNFQFRNLTNLQRDASVSSGQLRLTLN')
```
14.6 Analyzing structures

14.6.1 Measuring distances

The minus operator for atoms has been overloaded to return the distance between two atoms.

# Get some atoms
>>> ca1 = residue1["CA"]
>>> ca2 = residue2["CA"]
# Simply subtract the atoms to get their distance
>>> distance = ca1 - ca2

14.6.2 Measuring angles

Use the vector representation of the atomic coordinates, and the calc_angle function from the Vector module:

>>> vector1 = atom1.get_vector()
>>> vector2 = atom2.get_vector()
>>> vector3 = atom3.get_vector()
>>> angle = calc_angle(vector1, vector2, vector3)

14.6.3 Measuring torsion angles

Use the vector representation of the atomic coordinates, and the calc_dihedral function from the Vector module:

>>> vector1 = atom1.get_vector()
>>> vector2 = atom2.get_vector()
>>> vector3 = atom3.get_vector()
>>> vector4 = atom4.get_vector()
>>> angle = calc_dihedral(vector1, vector2, vector3, vector4)

14.6.4 Internal coordinates - distances, angles, torsion angles, distance plots, etc

Protein structures are normally supplied in 3D XYZ coordinates relative to a fixed origin, as in a PDB or mmCIF file. The internal_coords module facilitates converting this system to and from bond lengths, angles and dihedral angles. In addition to supporting standard psi, phi, chi, etc. calculations on protein structures, this representation is invariant to translation and rotation, and the implementation exposes multiple benefits for structure analysis.

First load up some modules here for later examples:

```python
>>> from Bio.PDB.PDBParser import PDBParser
>>> from Bio.PDB.Chain import Chain
>>> from Bio.PDB.internal_coords import *
>>> from Bio.PDB.PICIO import write_PIC, read_PIC, read_PIC_seq
>>> from Bio.PDB.ic_rebuild import write_PDB, IC_duplicate, structure_rebuild_test
>>> from Bio.PDB.SCADIO import write_SCAD
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.PDB.PDBIO import PDBIO
>>> import numpy as np
```
14.6.4.1 Accessing dihedrals, angles and bond lengths

We start with the simple case of computing internal coordinates for a structure:

```
>>> # load a structure as normal, get first chain
>>> parser = PDBParser()
>>> myProtein = parser.get_structure("1a8o", "1A8O.pdb")
>>> myChain = myProtein[0]["A"]

>>> # compute bond lengths, angles, dihedral angles
>>> myChain.atom_to_internal_coordinates(verbose=True)
chain break at THR 186 due to MaxPeptideBond (1.4 angstroms) exceeded
chain break at THR 216 due to MaxPeptideBond (1.4 angstroms) exceeded
```

The chain break warnings for 1A8O are suppressed by removing the `verbose=True` option above. To avoid the creation of a break and instead allow unrealistically long N-C bonds, override the class variable `MaxPeptideBond`, e.g.:

```
>>> IC_Chain.MaxPeptideBond = 4.0
>>> myChain.internal_coord = None  # force re-loading structure data with new cutoff
>>> myChain.atom_to_internal_coordinates(verbose=True)
```

At this point the values are available at both the chain and residue level. The first residue of 1A8O is HETATM MSE (selenomethionine), so we investigate residue 2 below using either canonical names or atom specifiers. Here we obtain the \( \chi_1 \) dihedral and \( \tau \) angles by name and by atom sequence, and the \( \text{C}_\alpha-\text{C}_\beta \) distance by specifying the atom pair:

```
>>> r2 = myChain.child_list[1]
>>> r2
<Residue ASP het= resseq=152 icode= >
>>> r2ic = r2.internal_coord
>>> print(r2ic, "", r2ic.pretty_str(), "", r2ic.rbase, "", r2ic.lc)
('1a8o', 0, 'A', (' ', 152, ' ')) : ASP 152 : (152, None, 'D') : D

>>> r2chi1 = r2ic.get_angle("chi1")
>>> print(round(r2chi1, 2))
-144.86
>>> r2ic.get_angle("chi1") == r2ic.get_angle("N:CA:CB:CG")
True
>>> print(round(r2ic.get_angle("tau"), 2))
113.45
>>> r2ic.get_angle("tau") == r2ic.get_angle("N:CA:C")
True
>>> print(round(r2ic.get_length("CA:CB"), 2))
1.53
```

The `Chain.internal_coord` object holds arrays and dictionaries of hedra (3 bonded atoms) and dihedra (4 bonded atoms) objects. The dictionaries are indexed by tuples of `AtomKey` objects; `AtomKey` objects capture residue position, insertion code, 1 or 3-character residue name, atom name, altloc and occupancy.

Below we obtain the same \( \chi_1 \) and \( \tau \) angles as above by indexing the `Chain` arrays directly, using `AtomKeys` to index the `Chain` arrays:

```
>>> myCic = myChain.internal_coord
```
>>> r2chi1_object = r2ic.pick_angle("chi1")
>>> # or same thing (as for get_angle() above):
>>> r2chi1_object == r2ic.pick_angle("N:CA:CB:CG")
True
>>> r2chi1_key = r2chi1_object.atomkeys
>>> r2chi1_key # r2chi1_key is tuple of AtomKeys
(152_D_N, 152_D_CA, 152_D_CB, 152_D_CG)
>>> r2chi1_index = myCic.dihedraNdx[r2chi1_key]
>>> # or same thing:
>>> r2chi1_index == r2chi1_object.ndx
True
>>> print(round(myCic.dihedraAngle[r2chi1_index], 2))
-144.86
>>> # also:
>>> r2chi1_object == myCic.dihedra[r2chi1_key]
True

>>> # hedra angles are similar:
>>> r2tau = r2ic.pick_angle("tau")
>>> print(round(myCic.hedraAngle[r2tau.ndx], 2))
113.45

Obtaining bond length data at the Chain level is more complicated (and not recommended). As shown here, multiple hedra will share a single bond in different positions:

>>> r2CaCb = r2ic.pick_length("CA:CB") # returns list of hedra containing bond
>>> r2CaCb[0][0].atomkeys
(152_D_CB, 152_D_CA, 152_D_C)
>>> print(round(myCic.hedraL12[r2CaCb[0][0].ndx], 2)) # position 1-2
1.53
>>> r2CaCb[0][1].atomkeys
(152_D_N, 152_D_CA, 152_D_CB)
>>> print(round(myCic.hedraL23[r2CaCb[0][1].ndx], 2)) # position 2-3
1.53
>>> r2CaCb[0][2].atomkeys
(152_D_CA, 152_D_CB, 152_D_CG)
>>> print(round(myCic.hedraL12[r2CaCb[0][2].ndx], 2)) # position 1-2
1.53

Please use the Residue level set_length function instead.

14.6.4.2 Testing structures for completeness

Missing atoms and other issues can cause problems when rebuilding a structure. Use structure_rebuild_test to determine quickly if a structure has sufficient data for a clean rebuild. Add verbose=True and/or inspect the result dictionary for more detail:

>>> # check myChain makes sense (can get angles and rebuild same structure)
>>> resultDict = structure_rebuild_test(myChain)
>>> resultDict["pass"]
True

14.6.4.3 Modifying and rebuilding structures

It’s preferable to use the residue level set_angle and set_length facilities for modifying internal coordinates rather than directly accessing the Chain structures. While directly modifying hedra angles is safe, bond lengths appear in multiple overlapping hedra as noted above, and this is handled by set_length. When applied to a dihedral angle, set_angle will wrap the result to +/-180 and rotate adjacent dihedra as well (such as both bonds for an isoleucine chi1 angle - which is probably what you want).

```python
>>> # rotate residue 2 chi1 angle by -120 degrees
>>> r2ic.set_angle("chi1", r2chi1 - 120.0)
```

```
>>> print(round(r2ic.get_angle("chi1"), 2))
95.14
```

```
>>> r2ic.set_length("CA:CB", 1.49)
```

```
>>> print(round(myCic.hedraL12[r2CaCb[0][0].ndx], 2))  # Cb-Ca-C position 1-2
1.49
```

Rebuilding a structure from internal coordinates is a simple call to internal_to_atom_coordinates():

```python
>>> myChain.internal_to_atom_coordinates()
```

```
>>> # just for proof:
>>> myChain.internal_coord = None  # all internal_coord data removed, only atoms left
>>> myChain.atom_to_internal_coordinates()  # re-generate internal coordinates
```

```
>>> print(round(r2ic.get_angle("chi1"), 2))  # show measured values match what was set above
95.14
```

```
>>> print(round(myCic.hedraL23[r2CaCb[0][1].ndx], 2))  # N-Ca-Cb position 2-3
1.49
```

The generated structure can be written with PDBIO, as normal:

```python
write_PDB(myProtein, "myChain.pdb")
# or just the ATOM records without headers:
io = PDBIO()
io.set_structure(myProtein)
io.save("myChain2.pdb")
```

14.6.4.4 Protein Internal Coordinate (.pic) files and default values

A file format is defined in the PICIO module to describe protein chains as hedra and dihedra relative to initial coordinates. All parts of the file other than the residue sequence information (e.g. ('1A8O', 0, 'A', (' ', 153, ' ')) ILE) are optional, and will be filled in with default values if not specified and read_PIC is called with the defaults=True option. Default values are calculated from Sep 2019 Dunbrack cullpdb pc20 res2.2 R1.0.

Here we write ‘myChain’ as a .pic file of internal coordinate specifications and then read it back in as ‘myProtein2’.

```python
# write chain as 'protein internal coordinates' (.pic) file
write_PIC(myProtein, "myChain.pic")
# read .pic file
myProtein2 = read_PIC("myChain.pic")
```

As all internal coordinate values can be replaced with defaults, PICIO.read_PIC_seq is supplied as a utility function to create a valid (mostly helical) default structure from an input sequence:
# create default structure for random sequence by reading as .pic file
myProtein3 = read_PIC_seq(
    SeqRecord(
        Seq("GAVLIMFPSTCNQYWDEHKR"),
        id="1RND",
        description="my random sequence",
    ),
)

myProtein3.internal_to_atom_coordinates()
write_PDB(myProtein3, "myRandom.pdb")

It may be of interest to explore the accuracy required in e.g. *omega* angles (180.0), hedra angles and/or bond lengths when generating structures from internal coordinates. The picFlags option to write_PIC enables this, allowing the selection of data to be written to the .pic file vs. left unspecified to get default values.

Various combinations are possible and some presets are supplied, for example classic will write only *psi*, *phi*, *tau*, proline *omega* and sidechain *chi* angles to the .pic file:

write_PIC(myProtein, "myChain.pic", picFlags=IC_Residue.pic_flags.classic)

myProtein2 = read_PIC("myChain.pic", defaults=True)

14.6.4.5 Accessing the all-atom AtomArray

All 3D XYZ coordinates in Biopython Atom objects are moved to a single large array in the Chain class and replaced by Numpy ‘views’ into this array in an early step of atom_to_internal_coordinates. Software accessing Biopython Atom coordinates is not affected, but the new array may offer efficiencies for future work.

Unlike the Atom XYZ coordinates, AtomArray coordinates are homogeneous, meaning they are arrays like \[ x \ y \ z \ 1.0 \] with 1.0 as the fourth element. This facilitates efficient transformation using combined translation and rotation matrices throughout the internal_coords module. There is a corresponding AtomArrayIndex dictionary, mapping AtomKeys to their coordinates.

Here we demonstrate reading coordinates for a specific C\(\beta\) atom from the array, then show that modifying the array value modifies the Atom object at the same time:

```python
>>> # access the array of all atoms for the chain, e.g. r2 above is residue 152 C-beta
>>> r2_cBeta_index = myChain.internal_coord.atomArrayIndex[AtomKey("152_D_CB")]
>>> r2_cBeta_coords = myChain.internal_coord.atomArray[r2_cBeta_index]
>>> print(np.round(r2_cBeta_coords, 2))
[-0.75 -1.18 -0.51 1. ]

>>> # the Biopython Atom coord array is now a view into atomArray, so
>>> assert r2_cBeta_coords[1] == r2["CB"].coord[1]
>>> r2_cBeta_coords[1] += 1.0  # change the Y coord 1 angstrom
>>> assert r2_cBeta_coords[1] == r2["CB"].coord[1]
>>> # they are always the same (they share the same memory)
>>> r2_cBeta_coords[1] -= 1.0  # restore

Note that it is easy to ‘break’ the view linkage between the Atom coord arrays and the chain atomArray. When modifying Atom coordinates directly, use syntax for an element-by-element copy to avoid this:

# use these:
myAtom1.coord[:] = myAtom2.coord
myAtom1.coord[...] = myAtom2.coord
```
myAtom1.coord[:] = [1, 2, 3]
for i in range(3):
    myAtom1.coord[i] = myAtom2.coord[i]

# do not use:
myAtom1.coord = myAtom2.coord
myAtom1.coord = [1, 2, 3]

Using the atomArrayIndex and knowledge of the AtomKey class enables us to create Numpy ‘selectors’, as shown below to extract an array of only the Cα atom coordinates:

```python
>>> # create a selector to filter just the C-alpha atoms from the all atom array
>>> atmNameNdx = AtomKey.fields.atm
>>> aaI = myChain.internal_coord.atomArrayIndex
>>> CaSelect = [aaI.get(k) for k in aaI.keys() if k.akl[atmNameNdx] == "CA"]
>>> # now the ordered array of C-alpha atom coordinates is:
>>> CA_coords = myChain.internal_coord.atomArray[CaSelect]
>>> # (if you modify it, the original atomArray is unaffected)
```

### 14.6.4.6 Distance Plots

A benefit of the atomArray is that generating a distance plot from it is a single line of Numpy code:

```python
np.linalg.norm(atomArray[:, None, :] - atomArray[None, :, :], axis=-1)
```

Despite its briefness, the idiom can be difficult to remember and in the form above generates all-atom distances rather than the classic Cα plot as may be desired. The distance_plot method wraps the line above and accepts an optional selector like CaSelect defined in the previous section. See Figure 14.2.

```python
# create a C-alpha distance plot
caDistances = myChain.internal_coord.distance_plot(CaSelect)
# display with e.g. MatPlotLib:
import matplotlib.pyplot as plt
plt.imshow(caDistances, cmap="hot", interpolation="nearest")
plt.show()
```

![Figure 14.2: Cα distance plot for PDB file 1A8O (HIV capsid C-terminal domain)](image-url)
14.6.4.7 Building a structure from a distance plot

The all-atom distance plot is another representation of a protein structure, also invariant to translation and rotation but lacking in chirality information (a mirror-image structure will generate the same distance plot). By combining the distance matrix with the signs of each dihedral angle, it is possible to regenerate the internal coordinates.

This work uses equations developed by Blue, the Hedronometer, discussed in https://math.stackexchange.com/a/49340/409 and further in http://daylateanddollarshort.com/mathdocs/Heron-like-Results-for-Tetrahedral.pdf.

To begin, we extract the distances and chirality values from ‘myChain’:

```python
>>> # build structure from distance plot:

>>> ## create the all-atom distance plot
>>> distances = myCic.distance_plot()

>>> ## get the signs of the dihedral angles
>>> chirality = myCic.dihedral_signs()
```

We need a valid data structure matching ‘myChain’ to correctly rebuild it; using read_PIC_seq above would work in the general case, but the 1A8O example used here has some ALTLOC complexity which the sequence alone would not generate. For demonstration the easiest approach is to simply duplicate the ‘myChain’ structure, but we set all the atom and internal coordinate chain arrays to 0s (only for demonstration) just to be certain there is no data coming through from the original structure:

```python
>>> ## get new, empty data structure : copy data structure from myChain
>>> myChain2 = IC_duplicate(myChain)[0]['A']

>>> ## clear the new atomArray and di/hedra value arrays, just for proof
>>> cic2 = myChain2.internal_coord

>>> cic2.atomArray = np.zeros((cic2.AAsiz, 4), dtype=np.float64)

>>> cic2.dihedraAngle[:] = 0.0

>>> cic2.hedraAngle[:] = 0.0

>>> cic2.hedraL12[:] = 0.0

>>> cic2.hedraL23[:] = 0.0
```

The approach is to regenerate the internal coordinates from the distance plot data, then generate the atom coordinates from the internal coordinates as shown above. To place the final generated structure in the same coordinate space as the starting structure, we copy just the coordinates for the first three N-Ca-C atoms from the chain start of ‘myChain’ to the ‘myChain2’ structure (this is only needed to demonstrate equivalence at end):

```python
>>> ## copy just the first N-Ca-C coords so structures will superimpose:

>>> cic2.copy_initNCaCs(myChain.internal_coord)
```

The distance_to_internal_coordinates routine needs arrays of the six inter-atom distances for each dihedron for the target structure. The convenience routine distplot_to_dh_arrays extracts these values from the previously generated distance matrix as needed, and may be replaced by a user method to write these data to the arrays in the Chain.internal_coords object.

```python
>>> ## copy distances to chain arrays:

>>> cic2.distplot_to_dh_arrays(distances, chirality)

>>> ## compute angles and dihedral angles from distances:

>>> cic2.distance_to_internal_coordinates()
```
The steps below generate the atom coordinates from the newly generated ‘myChain2’ internal coordinates, then use the Numpy allclose routine to confirm that all values match to better than PDB file resolution:

```python
>>> # generate XYZ coordinates from internal coordinates:
>>> myChain2.internal_to_atom_coordinates()

>>> # confirm result atomArray matches original structure:
>>> np.allclose(cic2.atomArray, myCic.atomArray)
True
```

Note that this procedure does not use the entire distance matrix, but only the six local distances between the four atoms of each dihedral angle.

### 14.6.4.8 Superimposing residues and their neighborhoods

The `internal_coords` module relies on transforming atom coordinates between different coordinate spaces for both calculation of torsion angles and reconstruction of structures. Each dihedron has a coordinate space transform placing its first atom on the XZ plane, second atom at the origin, and third atom on the +Z axis, as well as a corresponding reverse transform which will return it to the coordinates in the original structure. These transform matrices are available to use as shown below. By judicious choice of a reference dihedron, pairwise and higher order residue interactions can be investigated and visualized across multiple protein structures, e.g. Figure 14.3.

![Figure 14.3: Neighboring phenylalanine sidechains in PDB file 3PBL (human dopamine D3 receptor)](image)

This example superimposes each PHE residue in a chain on its N-Co-Cβ atoms, and presents all PHEs in the chain in the respective coordinate space as a simple demonstration. A more realistic exploration of pairwise sidechain interactions would examine a dataset of structures and filter for interaction classes as discussed in the relevant literature.

```python
# superimpose all phe-phe pairs - quick hack just to demonstrate concept
# for analyzing pairwise residue interactions. Generates PDB ATOM records
```
# placing each PHE at origin and showing all other PHEs in environment

### shorthand for key variables:
```python
cic = myChain.internal_coord
resNameNdx = AtomKey.fields.resname
aaNdx = cic.atomArrayIndex
```

### select just PHE atoms:
```python
pheAtomSelect = [aaNdx.get(k) for k in aaNdx.keys() if k.a[k][resNameNdx] == "F"]
aaF = cic.atomArray[pheAtomSelect]  # numpy fancy indexing makes COPY not view
```

```python
for ric in cic.ordered_aa_ic_list:  # internal_coords version of get_residues()
    if ric.lc == "F":  # if PHE, get transform matrices for chi1 dihedral
        ch1 = ric.pick_angle("ch1")  # N:CA:CB:CG space has C-alpha at origin
        cst = np.transpose(ch1.cst)  # transform TO chi1 space
        # rcst = np.transpose(ch1.rcst)  # transform FROM chi1 space (not needed here)
        cic.atomArray[pheAtomSelect] = aaF.dot(cst)  # transform just the PHEs
    for res in myChain.get_residues():  # print PHEs in new coordinate space
        if res.resname in ["PHE"]:  # print PHEs in new coordinate space
            print(res.internal_coord.pdb_residue_string())

        cic.atomArray[pheAtomSelect] = aaF  # restore coordinate space from copy
```

### 3D printing protein structures

OpenSCAD ([https://openscad.org](https://openscad.org)) is a language for creating solid 3D CAD objects. The algorithm to construct a protein structure from internal coordinates is supplied in OpenSCAD with data describing a structure, such that a model can be generated suitable for 3D printing. While other software can generate STL data as a rendering option for 3D printing (e.g. Chimera, [https://www.cgl.ucsf.edu/chimera/](https://www.cgl.ucsf.edu/chimera/)), this approach generates spheres and cylinders as output and is therefore more amenable to modifications relevant to 3D printing protein structures. Individual residues and bonds can be selected in the OpenSCAD code for special handling, such as highlighting by size or adding rotatable bonds in specific positions (see [https://www.thingiverse.com/thing:3957471](https://www.thingiverse.com/thing:3957471) for an example).

```python
# write OpenSCAD program of spheres and cylinders to 3d print myChain backbone
## set atom load filter to accept backbone only:
IC_Residue.accept_atoms = IC_Residue.accept_backbone
## set chain break cutoff very high to bridge missing residues with long bonds
IC_Chain.MaxPeptideBond = 4.0
## delete existing data to force re-read of all atoms with attributes set above:
myChain.internal_coord = None
write_SCAD(myChain, "myChain.scad", scale=10.0)
```

### internal_coords control attributes

A few control attributes are available in the `internal_coords` classes to modify or filter data as internal coordinates are calculated. These are listed in Table 14.1:

### Determining atom-atom contacts

Use `NeighborSearch` to perform neighbor lookup. The neighbor lookup is done using a KD tree module written in C (see the `KDTree` class in module `Bio.PDB.kdtrees`), making it very fast. It also includes a fast method to find all point pairs within a certain distance of each other.
Table 14.1: Control attributes in Bio.PDB.internal_coords.

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

14.6.6 Superimposing two structures

Use a Superimposer object to superimpose two coordinate sets. This object calculates the rotation and translation matrix that rotates two lists of atoms on top of each other in such a way that their RMSD is minimized. Of course, the two lists need to contain the same number of atoms. The Superimposer object can also apply the rotation/translation to a list of atoms. The rotation and translation are stored as a tuple in the rotran attribute of the Superimposer object (note that the rotation is right multiplying!). The RMSD is stored in the rmsd attribute.

The algorithm used by Superimposer comes from [15, Golub & Van Loan] and makes use of singular value decomposition (this is implemented in the general Bio.SVDSuperimposer module).

Example:

```python
>>> sup = Superimposer()
# Specify the atom lists
# 'fixed' and 'moving' are lists of Atom objects
# The moving atoms will be put on the fixed atoms
>>> sup.set_atoms(fixed, moving)
# Print rotation/translation/rmsd
>>> print(sup.rotran)
>>> print(sup.rms)
# Apply rotation/translation to the moving atoms
>>> sup.apply(moving)
```

To superimpose two structures based on their active sites, use the active site atoms to calculate the rotation/translation matrices (as above), and apply these to the whole molecule.

14.6.7 Mapping the residues of two related structures onto each other

First, create an alignment file in FASTA format, then use the StructureAlignment class. This class can also be used for alignments with more than two structures.

14.6.8 Calculating the Half Sphere Exposure

Half Sphere Exposure (HSE) is a new, 2D measure of solvent exposure [18]. Basically, it counts the number of Cα atoms around a residue in the direction of its side chain, and in the opposite direction (within a radius of 13 Å. Despite its simplicity, it outperforms many other measures of solvent exposure.

HSE comes in two flavors: HSEα and HSEβ. The former only uses the Cα atom positions, while the latter uses the Cα and Cβ atom positions. The HSE measure is calculated by the HSEExposure class, which can also calculate the contact number. The latter class has methods which return dictionaries that map a Residue object to its corresponding HSEα, HSEβ and contact number values.

Example:

```python
>>> model = structure[0]
>>> hse = HSEExposure()
```
DSSP codes in Bio.PDB.

<table>
<thead>
<tr>
<th>Code</th>
<th>Secondary structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>α-helix</td>
</tr>
<tr>
<td>B</td>
<td>Isolated β-bridge residue</td>
</tr>
<tr>
<td>E</td>
<td>Strand</td>
</tr>
<tr>
<td>G</td>
<td>3-10 helix</td>
</tr>
<tr>
<td>I</td>
<td>H-helix</td>
</tr>
<tr>
<td>T</td>
<td>Turn</td>
</tr>
<tr>
<td>S</td>
<td>Bend</td>
</tr>
<tr>
<td>-</td>
<td>Other</td>
</tr>
</tbody>
</table>

Table 14.2: DSSP codes in Bio.PDB.

```python
# Calculate HSEalpha
>>> exp_ca = hse.calc_hs_exposure(model, option="CA3")
# Calculate HSEbeta
>>> exp_cb = hse.calc_hs_exposure(model, option="CB")
# Calculate classical coordination number
>>> exp_fs = hse.calc_fs_exposure(model)
# Print HSEalpha for a residue
>>> print(exp_ca[some_residue])
```

### 14.6.9 Determining the secondary structure

For this functionality, you need to install DSSP (and obtain a license for it — free for academic use, see [https://swift.cmbi.umcn.nl/gv/dssp/](https://swift.cmbi.umcn.nl/gv/dssp/)). Then use the DSSP class, which maps Residue objects to their secondary structure (and accessible surface area). The DSSP codes are listed in Table 14.2. Note that DSSP (the program, and thus by consequence the class) cannot handle multiple models!

The DSSP class can also be used to calculate the accessible surface area of a residue. But see also section 14.6.10.

### 14.6.10 Calculating the residue depth

Residue depth is the average distance of a residue’s atoms from the solvent accessible surface. It’s a fairly new and very powerful parameterization of solvent accessibility. For this functionality, you need to install Michel Sanner’s MSMS program ([https://www.scripps.edu/sanner/html/msms_home.html](https://www.scripps.edu/sanner/html/msms_home.html)). Then use the ResidueDepth class. This class behaves as a dictionary which maps Residue objects to corresponding (residue depth, Cα depth) tuples. The Cα depth is the distance of a residue’s Cα atom to the solvent accessible surface.

Example:

```python
>>> model = structure[0]
>>> rd = ResidueDepth(model, pdb_file)
>>> residue_depth, ca_depth = rd[some_residue]
```

You can also get access to the molecular surface itself (via the get_surface function), in the form of a Numeric Python array with the surface points.

### 14.7 Common problems in PDB files

It is well known that many PDB files contain semantic errors (not the structures themselves, but their representation in PDB files). Bio.PDB tries to handle this in two ways. The PDBParser object can behave in two ways: a restrictive way and a permissive way, which is the default.
Example:

```python
# Permissive parser
>>> parser = PDBParser(PERMISSIVE=1)
>>> parser = PDBParser()  # The same (default)
# Strict parser
>>> strict_parser = PDBParser(PERMISSIVE=0)
```

In the permissive state (DEFAULT), PDB files that obviously contain errors are “corrected” (i.e. some residues or atoms are left out). These errors include:

- Multiple residues with the same identifier
- Multiple atoms with the same identifier (taking into account the altloc identifier)

These errors indicate real problems in the PDB file (for details see [16, Hamelryck and Manderick, 2003]).

In the restrictive state, PDB files with errors cause an exception to occur. This is useful to find errors in PDB files.

Some errors however are automatically corrected. Normally each disordered atom should have a non-blank altloc identifier. However, there are many structures that do not follow this convention, and have a blank and a non-blank identifier for two disordered positions of the same atom. This is automatically interpreted in the right way.

Sometimes a structure contains a list of residues belonging to chain A, followed by residues belonging to chain B, and again followed by residues belonging to chain A, i.e. the chains are 'broken'. This is also correctly interpreted.

### 14.7.1 Examples

The PDBParser/Structure class was tested on about 800 structures (each belonging to a unique SCOP superfamily). This takes about 20 minutes, or on average 1.5 seconds per structure. Parsing the structure of the large ribosomal subunit (1FKK), which contains about 64000 atoms, takes 10 seconds on a 1000 MHz PC.

Three exceptions were generated in cases where an unambiguous data structure could not be built. In all three cases, the likely cause is an error in the PDB file that should be corrected. Generating an exception in these cases is much better than running the chance of incorrectly describing the structure in a data structure.

#### 14.7.1.1 Duplicate residues

One structure contains two amino acid residues in one chain with the same sequence identifier (resseq 3) and icode. Upon inspection it was found that this chain contains the residues Thr A3, . . . , Gly A202, Leu A3, Glu A204. Clearly, Leu A3 should be Leu A203. A couple of similar situations exist for structure 1FFK (which e.g. contains Gly B64, Met B65, Glu B65, Thr B67, i.e. residue Glu B65 should be Glu B66).

#### 14.7.1.2 Duplicate atoms

Structure 1EJG contains a Ser/Pro point mutation in chain A at position 22. In turn, Ser 22 contains some disordered atoms. As expected, all atoms belonging to Ser 22 have a non-blank altloc specifier (B or C). All atoms of Pro 22 have altloc A, except the N atom which has a blank altloc. This generates an exception, because all atoms belonging to two residues at a point mutation should have non-blank altloc. It turns out that this atom is probably shared by Ser and Pro 22, as Ser 22 misses the N atom. Again, this points to a problem in the file: the N atom should be present in both the Ser and the Pro residue, in both cases associated with a suitable altloc identifier.
14.7.2 Automatic correction

Some errors are quite common and can be easily corrected without much risk of making a wrong interpretation. These cases are listed below.

14.7.2.1 A blank altloc for a disordered atom

Normally each disordered atom should have a non-blank altloc identifier. However, there are many structures that do not follow this convention, and have a blank and a non-blank identifier for two disordered positions of the same atom. This is automatically interpreted in the right way.

14.7.2.2 Broken chains

Sometimes a structure contains a list of residues belonging to chain A, followed by residues belonging to chain B, and again followed by residues belonging to chain A, i.e. the chains are “broken”. This is correctly interpreted.

14.7.3 Fatal errors

Sometimes a PDB file cannot be unambiguously interpreted. Rather than guessing and risking a mistake, an exception is generated, and the user is expected to correct the PDB file. These cases are listed below.

14.7.3.1 Duplicate residues

All residues in a chain should have a unique id. This id is generated based on:

- The sequence identifier (resseq).
- The insertion code (icode).
- The hetfield string (“W” for waters and “H” followed by the residue name for other hetero residues)
- The residue names of the residues in the case of point mutations (to store the Residue objects in a DisorderedResidue object).

If this does not lead to a unique id something is quite likely wrong, and an exception is generated.

14.7.3.2 Duplicate atoms

All atoms in a residue should have a unique id. This id is generated based on:

- The atom name (without spaces, or with spaces if a problem arises).
- The altloc specifier.

If this does not lead to a unique id something is quite likely wrong, and an exception is generated.

14.8 Accessing the Protein Data Bank

14.8.1 Downloading structures from the Protein Data Bank

Structures can be downloaded from the PDB (Protein Data Bank) by using the `retrieve_pdb_file` method on a `PDBList` object. The argument for this method is the PDB identifier of the structure.

```python
>>> pdbl = PDBList()
>>> pdbl.retrieve_pdb_file("1FAT")
```
The PDBList class can also be used as a command-line tool:

```python
python PDBList.py 1fat
```

The downloaded file will be called `pdb1fat.ent` and stored in the current working directory. Note that the `retrieve_pdb_file` method also has an optional argument `pdir` that specifies a specific directory in which to store the downloaded PDB files.

The `retrieve_pdb_file` method also has some options to specify the compression format used for the download, and the program used for local decompression (default `.Z` format and `gunzip`). In addition, the PDB ftp site can be specified upon creation of the PDBList object. By default, the server of the Worldwide Protein Data Bank (`ftp://ftp.wwpdb.org/pub/pdb/data/structures/divided/pdb/`) is used. See the API documentation for more details. Thanks again to Kristian Rother for donating this module.

### 14.8.2 Downloading the entire PDB

The following commands will store all PDB files in the `/data/pdb` directory:

```python
python PDBList.py all /data/pdb
```

```python
python PDBList.py all /data/pdb -d
```

The API method for this is called `download_entire_pdb`. Adding the `-d` option will store all files in the same directory. Otherwise, they are sorted into PDB-style subdirectories according to their PDB ID's. Depending on the traffic, a complete download will take 2-4 days.

### 14.8.3 Keeping a local copy of the PDB up to date

This can also be done using the PDBList object. One simply creates a PDBList object (specifying the directory where the local copy of the PDB is present) and calls the `update_pdb` method:

```python
>>> pl = PDBList(pdb="/data/pdb")
>>> pl.update_pdb()
```

One can of course make a weekly cronjob out of this to keep the local copy automatically up-to-date. The PDB ftp site can also be specified (see API documentation).

PDBList has some additional methods that can be of use. The `get_allObsolete` method can be used to get a list of all obsolete PDB entries. The `changedThisWeek` method can be used to obtain the entries that were added, modified or obsoleted during the current week. For more info on the possibilities of PDBList, see the API documentation.

### 14.9 General questions

#### 14.9.1 How well tested is Bio.PDB?

Pretty well, actually. Bio.PDB has been extensively tested on nearly 5500 structures from the PDB - all structures seemed to be parsed correctly. More details can be found in the Bio.PDB Bioinformatics article. Bio.PDB has been used/is being used in many research projects as a reliable tool. In fact, I’m using Bio.PDB almost daily for research purposes and continue working on improving it and adding new features.

#### 14.9.2 How fast is it?

The PDBParser performance was tested on about 800 structures (each belonging to a unique SCOP superfamily). This takes about 20 minutes, or on average 1.5 seconds per structure. Parsing the structure of the large ribosomal subunit (1FKK), which contains about 64000 atoms, takes 10 seconds on a 1000 MHz PC. In short: it’s more than fast enough for many applications.
14.9.3 Is there support for molecular graphics?

Not directly, mostly since there are quite a few Python based/Python aware solutions already, that can potentially be used with Bio.PDB. My choice is Pymol, BTW (I’ve used this successfully with Bio.PDB, and there will probably be specific PyMol modules in Bio.PDB soon/some day). Python based/aware molecular graphics solutions include:

- PyMol: https://pymol.org/
- Chimera: https://www.cgl.ucsf.edu/chimera/
- PMV: http://www.scripps.edu/~sanner/python/
- Coot: https://ww2.mrc-lmb.cam.ac.uk/personal/pemsley/coot/
- CCP4mg: http://www.ccp4.ac.uk/CCP4/
- mmLib: http://pymmlib.sourceforge.net/
- VMD: https://www.ks.uiuc.edu/Research/vmd/
- MMTK: http://dirac.cnrs-orleans.fr/MMTK/

14.9.4 Who’s using Bio.PDB?

Bio.PDB was used in the construction of DISEMBL, a web server that predicts disordered regions in proteins (http://dis.embl.de/). Bio.PDB has also been used to perform a large scale search for active sites similarities between protein structures in the PDB [17, Hamelryck, 2003], and to develop a new algorithm that identifies linear secondary structure elements [31, Majumdar et al., 2005].

Judging from requests for features and information, Bio.PDB is also used by several LPCs (Large Pharmaceutical Companies :-).
Chapter 15

Bio.PopGen: Population genetics

Bio.PopGen is a Biopython module supporting population genetics, available in Biopython 1.44 onwards. The objective for the module is to support widely used data formats, applications and databases.

15.1 GenePop

GenePop (http://genepop.curtin.edu.au/) is a popular population genetics software package supporting Hardy-Weinberg tests, linkage disequilibrium, population differentiation, basic statistics, $F_{st}$ and migration estimates, among others. GenePop does not supply sequence based statistics as it doesn’t handle sequence data. The GenePop file format is supported by a wide range of other population genetic software applications, thus making it a relevant format in the population genetics field.

Bio.PopGen provides a parser and generator of GenePop file format. Utilities to manipulate the content of a record are also provided. Here is an example on how to read a GenePop file (you can find example GenePop data files in the Test/PopGen directory of Biopython):

```python
from Bio.PopGen import GenePop

with open("example.gen") as handle:
    rec = GenePop.read(handle)
```

This will read a file called example.gen and parse it. If you do print rec, the record will be output again, in GenePop format.

The most important information in rec will be the loci names and population information (but there is more – use help(GenePop.Record) to check the API documentation). Loci names can be found on rec.loci_list. Population information can be found on rec.populations. Populations is a list with one element per population. Each element is itself a list of individuals, each individual is a pair composed by individual name and a list of alleles (2 per marker), here is an example for rec.populations:

```python
[
    [
        ("Ind1", [(1, 2), (3, 3), (200, 201)]),
        ("Ind2", [(2, None), (3, 3), (None, None)]),
    ],
    [
        ("Other1", [(1, 1), (4, 3), (200, 200)]),
    ],
]```
So we have two populations, the first with two individuals, the second with only one. The first individual of the first population is called Ind1, allelic information for each of the 3 loci follows. Please note that for any locus, information might be missing (see as an example, Ind2 above).

A few utility functions to manipulate GenePop records are made available, here is an example:

```python
from Bio.PopGen import GenePop

# Imagine that you have loaded rec, as per the code snippet above...

rec.remove_population(pos)
# Removes a population from a record, pos is the population position in
# rec.populations, remember that it starts on position 0.
# rec is altered.

rec.remove_locus_by_position(pos)
# Removes a locus by its position, pos is the locus position in
# rec.loci_list, remember that it starts on position 0.
# rec is altered.

rec.remove_locus_by_name(name)
# Removes a locus by its name, name is the locus name as in
# rec.loci_list. If the name doesn't exist the function fails
# silently.
# rec is altered.

rec_loci = rec.split_in_loci()
# Splits a record in loci, that is, for each loci, it creates a new
# record, with a single loci and all populations.
# The result is returned in a dictionary, being each key the locus name.
# The value is the GenePop record.
# rec is not altered.

rec_pops = rec.split_in_pops(pop_names)
# Splits a record in populations, that is, for each population, it creates
# a new record, with a single population and all loci.
# The result is returned in a dictionary, being each key
# the population name. As population names are not available in GenePop,
# they are passed in array (pop_names).
# The value of each dictionary entry is the GenePop record.
# rec is not altered.
```

GenePop does not support population names, a limitation which can be cumbersome at times. Functionality to enable population names is currently being planned for Biopython. These extensions won’t break compatibility in any way with the standard format. In the medium term, we would also like to support the GenePop web service.
Chapter 16
Phylogenetics with Bio.Phylo

The Bio.Phylo module was introduced in Biopython 1.54. Following the lead of SeqIO and AlignIO, it aims to provide a common way to work with phylogenetic trees independently of the source data format, as well as a consistent API for I/O operations.

Bio.Phylo is described in an open-access journal article [45, Talevich et al., 2012], which you might also find helpful.

16.1 Demo: What’s in a Tree?

To get acquainted with the module, let’s start with a tree that we’ve already constructed, and inspect it a few different ways. Then we’ll colorize the branches, to use a special phyloXML feature, and finally save it.

Create a simple Newick file named simple.dnd using your favorite text editor, or use simple.dnd provided with the Biopython source code:

(((A,B),(C,D)),(E,F,G));

This tree has no branch lengths, only a topology and labeled terminals. (If you have a real tree file available, you can follow this demo using that instead.)

Launch the Python interpreter of your choice:

$ ipython -pylab

For interactive work, launching the IPython interpreter with the -pylab flag enables matplotlib integration, so graphics will pop up automatically. We’ll use that during this demo.

Now, within Python, read the tree file, giving the file name and the name of the format.

>>> from Bio import Phylo
>>> tree = Phylo.read("simple.dnd", "newick")

Printing the tree object as a string gives us a look at the entire object hierarchy.

>>> print(tree)
Tree(rooted=False, weight=1.0)
  Clade()
    Clade()
      Clade()
        Clade(name='A')
        Clade(name='B')
      Clade()
The `Tree` object contains global information about the tree, such as whether it’s rooted or unrooted. It has one root clade, and under that, it’s nested lists of clades all the way down to the tips.

The function `draw_ascii` creates a simple ASCII-art (plain text) dendrogram. This is a convenient visualization for interactive exploration, in case better graphical tools aren’t available.

```python
>>> from Bio import Phylo
>>> tree = Phylo.read("simple.dnd", "newick")
>>> Phylo.draw_ascii(tree)
```

If you have `matplotlib` or `pylab` installed, you can create a graphical tree using the `draw` function.

```python
>>> tree.rooted = True
>>> Phylo.draw(tree)
```

See Figure 16.1.

### 16.1.1 Coloring branches within a tree

The function `draw` supports the display of different colors and branch widths in a tree. As of Biopython 1.59, the `color` and `width` attributes are available on the basic `Clade` object and there’s nothing extra required to use them. Both attributes refer to the branch leading the given clade, and apply recursively; so all descendent branches will also inherit the assigned width and color values during display.

In earlier versions of Biopython, these were special features of PhyloXML trees, and using the attributes required first converting the tree to a subclass of the basic tree object called `Phylogeny`, from the Bio.Phylo.PhyloXML module.

In Biopython 1.55 and later, this is a convenient tree method:

```python
>>> tree = tree.as_phyloxml()
```

In Biopython 1.54, you can accomplish the same thing with one extra import:

```python
>>> from Bio.Phylo.PhyloXML import Phylogeny
>>> tree = Phylogeny.from_tree(tree)
```
Note that the file formats Newick and Nexus don’t support branch colors or widths, so if you use these attributes in Bio.Phylo, you will only be able to save the values in PhyloXML format. (You can still save a tree as Newick or Nexus, but the color and width values will be skipped in the output file.)

Now we can begin assigning colors. First, we’ll color the root clade gray. We can do that by assigning the 24-bit color value as an RGB triple, an HTML-style hex string, or the name of one of the predefined colors.

```python
>>> tree.root.color = (128, 128, 128)
```

Or:

```python
>>> tree.root.color = "#808080"
```

Or:

```python
>>> tree.root.color = "gray"
```

Colors for a clade are treated as cascading down through the entire clade, so when we colorize the root here, it turns the whole tree gray. We can override that by assigning a different color lower down on the tree.

Let’s target the most recent common ancestor (MRCA) of the nodes named “E” and “F”. The `common_ancestor` method returns a reference to that clade in the original tree, so when we color that clade “salmon”, the color will show up in the original tree.

```python
>>> mrca = tree.common_ancestor({"name": "E"}, {"name": "F"})
>>> mrca.color = "salmon"
```

If we happened to know exactly where a certain clade is in the tree, in terms of nested list entries, we can jump directly to that position in the tree by indexing it. Here, the index `[0, 1]` refers to the second child of the first child of the root.
>>> tree.clade[0, 1].color = "blue"

Finally, show our work:

>>> Phylo.draw(tree)

![A colorized tree drawn with Phylo.draw.](image)

Figure 16.2: A colorized tree drawn with Phylo.draw.

See Figure 16.2.

Note that a clade’s color includes the branch leading to that clade, as well as its descendents. The common ancestor of E and F turns out to be just under the root, and with this coloring we can see exactly where the root of the tree is.

My, we’ve accomplished a lot! Let’s take a break here and save our work. Call the `write` function with a file name or handle — here we use standard output, to see what would be written — and the format `phyloxml`. PhyloXML saves the colors we assigned, so you can open this phyloXML file in another tree viewer like Archaeopteryx, and the colors will show up there, too.

```python
>>> import sys
>>> n = Phylo.write(tree, sys.stdout, "phyloxml")  # doctest:+ELLIPSIS
<phyloxml ...
<phylogeny rooted="true">
  <clade>
    <color>
      <red>128</red>
      <green>128</green>
      <blue>128</blue>
    </color>
  </clade>
```

311
The rest of this chapter covers the core functionality of Bio.Phylo in greater detail. For more examples of using Bio.Phylo, see the cookbook page on Biopython.org:

http://biopython.org/wiki/Phylo_cookbook

16.2 I/O functions

Like SeqIO and AlignIO, Phylo handles file input and output through four functions: `parse`, `read`, `write` and `convert`, all of which support the tree file formats Newick, NEXUS, phyloXML and NeXML, as well as the Comparative Data Analysis Ontology (CDAO).

The `read` function parses a single tree in the given file and returns it. Careful; it will raise an error if the file contains more than one tree, or no trees.

```python
>>> from Bio import Phylo
>>> tree = Phylo.read("Tests/Nexus/int_node_labels.nwk", "newick")
>>> print(tree)  # doctest:+ELLIPSIS
Tree(rooted=False, weight=1.0)
  Clade(branch_length=75.0, name='gymnosperm')
    Clade(branch_length=25.0, name='Coniferales')
      Clade(branch_length=25.0)
        Clade(branch_length=10.0, name='Tax+nonSci')
          Clade(branch_length=90.0, name='Taxaceae')
            Clade(branch_length=125.0, name='Cephalotaxus')
...}

(Example files are available in the Tests/Nexus/ and Tests/PhyloXML/ directories of the Biopython distribution.)

To handle multiple (or an unknown number of) trees, use the `parse` function iterates through each of the trees in the given file:
>>> trees = Phylo.parse("Tests/PhyloXML/phyloxml_examples.xml", "phyloxml")

>>> for tree in trees:
...     print(tree)  # doctest:+ELLIPSIS
...
Phylogeny(description='phyloXML allows to use either a "branch_length" attribute...', name='example from Prof. Joe Felsenstein’s book “Inferring Phyl...', rooted=True)
    Clade()
        Clade(branch_length=0.06)
            Clade(branch_length=0.102, name='A')

Write a tree or iterable of trees back to file with the write function:

>>> trees = Phylo.parse("Tests/PhyloXML/phyloxml_examples.xml", "phyloxml")
>>> tree1 = next(trees)
>>> Phylo.write(tree1, "tree1.nwk", "newick")
1
>>> Phylo.write(trees, "other_trees.xml", "phyloxml")  # write the remaining trees
12

Convert files between any of the supported formats with the convert function:

>>> Phylo.convert("tree1.nwk", "newick", "tree1.xml", "nexml")
1
>>> Phylo.convert("other_trees.xml", "phyloxml", "other_trees.nex", "nexus")
12

To use strings as input or output instead of actual files, use StringIO as you would with SeqIO and AlignIO:

>>> from Bio import Phylo
>>> from io import StringIO
>>> handle = StringIO("(((A,B),(C,D)),(E,F,G));")
>>> tree = Phylo.read(handle, "newick")

16.3 View and export trees

The simplest way to get an overview of a Tree object is to print it:

>>> from Bio import Phylo
>>> tree = Phylo.read("PhyloXML/example.xml", "phyloxml")
>>> print(tree)
Phylogeny(description='phyloXML allows to use either a "branch_length" attribute...', name='example from Prof. Joe Felsenstein’s book “Inferring Phyl...
    Clade()
        Clade(branch_length=0.06)
            Clade(branch_length=0.102, name='A')
        Clade(branch_length=0.23, name='B')
    Clade(branch_length=0.4, name='C')

This is essentially an outline of the object hierarchy Biopython uses to represent a tree. But more likely, you’d want to see a drawing of the tree. There are three functions to do this.
As we saw in the demo, draw_ascii prints an ascii-art drawing of the tree (a rooted phylogram) to standard output, or an open file handle if given. Not all of the available information about the tree is shown, but it provides a way to quickly view the tree without relying on any external dependencies.
Figure 16.3: A simple rooted tree plotted with the `draw` function.

```python
>>> tree = Phylo.read("PhyloXML/example.xml", "phyloxml")
>>> Phylo.draw_ascii(tree)
                      ___________ A
                     |                  |
                     |                  |
                     |                  |
                     |                  |
                     B
                     |
                     C
                    <BLANKLINE>
```

The `draw` function draws a more attractive image using the matplotlib library. See the API documentation for details on the arguments it accepts to customize the output.

```python
>>> Phylo.draw(tree, branch_labels=lambda c: c.branch_length)
```

See Figure 16.3 for example.

See the Phylo page on the Biopython wiki (http://biopython.org/wiki/Phylo) for descriptions and examples of the more advanced functionality in `draw_ascii`, `draw_graphviz` and `to_networkx`.

### 16.4 Using Tree and Clade objects

The `Tree` objects produced by `parse` and `read` are containers for recursive sub-trees, attached to the `Tree` object at the `root` attribute (whether or not the phylogenetic tree is actually considered rooted). A `Tree` has globally applied information for the phylogeny, such as rootedness, and a reference to a single `Clade`; a `Clade` has node- and clade-specific information, such as branch length, and a list of its own descendant `Clade` instances, attached at the `clades` attribute.

So there is a distinction between `tree` and `tree.root`. In practice, though, you rarely need to worry about it. To smooth over the difference, both `Tree` and `Clade` inherit from `TreeMixin`, which contains the implementations for methods that would be commonly used to search, inspect or modify a tree or any of its clades. This means that almost all of the methods supported by `tree` are also available on `tree.root` and any clade below it. (`Clade` also has a `root` property, which returns the clade object itself.)
16.4.1 Search and traversal methods

For convenience, we provide a couple of simplified methods that return all external or internal nodes directly as a list:

**get_terminals** makes a list of all of this tree’s terminal (leaf) nodes.

**get_nonterminals** makes a list of all of this tree’s nonterminal (internal) nodes.

These both wrap a method with full control over tree traversal, **find_clades**. Two more traversal methods, **find_elements** and **find_any**, rely on the same core functionality and accept the same arguments, which we’ll call a “target specification” for lack of a better description. These specify which objects in the tree will be matched and returned during iteration. The first argument can be any of the following types:

- A **TreeElement instance**, which tree elements will match by identity — so searching with a Clade instance as the target will find that clade in the tree;

- A **string**, which matches tree elements’ string representation — in particular, a clade’s name *(added in Biopython 1.56)*;

- A **class** or **type**, where every tree element of the same type (or sub-type) will be matched;

- A **dictionary** where keys are tree element attributes and values are matched to the corresponding attribute of each tree element. This one gets even more elaborate:
  - If an **int** is given, it matches numerically equal attributes, e.g. 1 will match 1 or 1.0
  - If a boolean is given (True or False), the corresponding attribute value is evaluated as a boolean and checked for the same
  - None matches None
  - If a string is given, the value is treated as a regular expression (which must match the whole string in the corresponding element attribute, not just a prefix). A given string without special regex characters will match string attributes exactly, so if you don’t use regexes, don’t worry about it. For example, in a tree with clade names Foo1, Foo2 and Foo3, `tree.find_clades(\{"name": "Foo1"\})` matches Foo1, `{"name": "Foo.*"}` matches all three clades, and `{"name": "Foo"}` doesn’t match anything.

Since floating-point arithmetic can produce some strange behavior, we don’t support matching **floats** directly. Instead, use the boolean True to match every element with a nonzero value in the specified attribute, then filter on that attribute manually with an inequality (or exact number, if you like living dangerously).

If the dictionary contains multiple entries, a matching element must match each of the given attribute values — think “and”, not “or”.

- A **function** taking a single argument (it will be applied to each element in the tree), returning True or False. For convenience, LookupError, AttributeError and ValueError are silenced, so this provides another safe way to search for floating-point values in the tree, or some more complex characteristic.

After the target, there are two optional keyword arguments:

**terminal** — A boolean value to select for or against terminal clades (a.k.a. leaf nodes): True searches for only terminal clades, False for non-terminal (internal) clades, and the default, None, searches both terminal and non-terminal clades, as well as any tree elements lacking the is_terminal method.

**order** — Tree traversal order: "preorder" (default) is depth-first search, "postorder" is DFS with child nodes preceding parents, and "level" is breadth-first search.
Finally, the methods accept arbitrary keyword arguments which are treated the same way as a dictionary target specification: keys indicate the name of the element attribute to search for, and the argument value (string, integer, None or boolean) is compared to the value of each attribute found. If no keyword arguments are given, then any TreeElement types are matched. The code for this is generally shorter than passing a dictionary as the target specification: `tree.find_clades({"name": "Foo1"})` can be shortened to `tree.find_clades(name="Foo1")`.

(In Biopython 1.56 or later, this can be even shorter: `tree.find_clades("Foo1")`)

Now that we've mastered target specifications, here are the methods used to traverse a tree:

- **find_clades** Find each clade containing a matching element. That is, find each element as with find_elements, but return the corresponding clade object. (This is usually what you want.)

  The result is an iterable through all matching objects, searching depth-first by default. This is not necessarily the same order as the elements appear in the Newick, Nexus or XML source file!

- **find_elements** Find all tree elements matching the given attributes, and return the matching elements themselves. Simple Newick trees don’t have complex sub-elements, so this behaves the same as find_clades on them. PhyloXML trees often do have complex objects attached to clades, so this method is useful for extracting those.

- **find_any** Return the first element found by find_elements(), or None. This is also useful for checking whether any matching element exists in the tree, and can be used in a conditional.

Two more methods help navigating between nodes in the tree:

- **get_path** List the clades directly between the tree root (or current clade) and the given target. Returns a list of all clade objects along this path, ending with the given target, but excluding the root clade.

- **trace** List of all clade object between two targets in this tree. Excluding start, including finish.

### 16.4.2 Information methods

These methods provide information about the whole tree (or any clade).

- **common_ancestor** Find the most recent common ancestor of all the given targets. (This will be a Clade object). If no target is given, returns the root of the current clade (the one this method is called from); if 1 target is given, this returns the target itself. However, if any of the specified targets are not found in the current tree (or clade), an exception is raised.

- **count_terminals** Counts the number of terminal (leaf) nodes within the tree.

- **depths** Create a mapping of tree clades to depths. The result is a dictionary where the keys are all of the Clade instances in the tree, and the values are the distance from the root to each clade (including terminals). By default the distance is the cumulative branch length leading to the clade, but with the `unit_branch_lengths=True` option, only the number of branches (levels in the tree) is counted.

- **distance** Calculate the sum of the branch lengths between two targets. If only one target is specified, the other is the root of this tree.

- **total_branch_length** Calculate the sum of all the branch lengths in this tree. This is usually just called the “length” of the tree in phylogenetics, but we use a more explicit name to avoid confusion with Python terminology.

The rest of these methods are boolean checks:
is_bifurcating  True if the tree is strictly bifurcating; i.e. all nodes have either 2 or 0 children (internal or external, respectively). The root may have 3 descendents and still be considered part of a bifurcating tree.

is_monophyletic Test if all of the given targets comprise a complete subclade — i.e., there exists a clade such that its terminals are the same set as the given targets. The targets should be terminals of the tree. For convenience, this method returns the common ancestor (MCRA) of the targets if they are monophyletic (instead of the value True), and False otherwise.

is_parent_of  True if target is a descendent of this tree — not required to be a direct descendent. To check direct descendents of a clade, simply use list membership testing: if subclade in clade: ...

is_preterminal  True if all direct descendents are terminal; False if any direct descendent is not terminal.

16.4.3 Modification methods

These methods modify the tree in-place. If you want to keep the original tree intact, make a complete copy of the tree first, using Python’s copy module:

tree = Phylo.read("example.xml", "phyloxml")
import copy
newtree = copy.deepcopy(tree)

collapse Deletes the target from the tree, relinking its children to its parent.
collapse_all Collapse all the descendents of this tree, leaving only terminals. Branch lengths are preserved, i.e. the distance to each terminal stays the same. With a target specification (see above), collapses only the internal nodes matching the specification.
ladderize Sort clades in-place according to the number of terminal nodes. Deepest clades are placed last by default. Use reverse=True to sort clades deepest-to-shallowest.

prune Prunes a terminal clade from the tree. If taxon is from a bifurcation, the connecting node will be collapsed and its branch length added to remaining terminal node. This might no longer be a meaningful value.

root_with_outgroup Reroot this tree with the outgroup clade containing the given targets, i.e. the common ancestor of the outgroup. This method is only available on Tree objects, not Clades.

If the outgroup is identical to self.root, no change occurs. If the outgroup clade is terminal (e.g. a single terminal node is given as the outgroup), a new bifurcating root clade is created with a 0-length branch to the given outgroup. Otherwise, the internal node at the base of the outgroup becomes a trifurcating root for the whole tree. If the original root was bifurcating, it is dropped from the tree. In all cases, the total branch length of the tree stays the same.

root_at_midpoint Reroot this tree at the calculated midpoint between the two most distant tips of the tree.

(This uses root_with_outgroup under the hood.)

split Generate n (default 2) new descendants. In a species tree, this is a speciation event. New clades have the given branch_length and the same name as this clade’s root plus an integer suffix (counting from 0) — for example, splitting a clade named “A” produces the sub-clades “A0” and “A1”.

See the Phylo page on the Biopython wiki (http://biopython.org/wiki/Phylo) for more examples of using the available methods.
16.4.4 Features of PhyloXML trees

The phyloXML file format includes fields for annotating trees with additional data types and visual cues.

See the PhyloXML page on the Biopython wiki (http://biopython.org/wiki/PhyloXML) for descriptions and examples of using the additional annotation features provided by PhyloXML.

16.5 Running external applications

While Bio.Phylo doesn’t infer trees from alignments itself, there are third-party programs available that do. These can be accessed from within python by using the subprocess module.

Below is an example on how to use a python script to interact with PhyML (http://www.atgc-montpellier.fr/phyml/). The program accepts an input alignment in phylip-relaxed format (that’s Phylip format, but without the 10-character limit on taxon names) and a variety of options.

```python
>>> import subprocess
>>> cmd = "phyml -i Tests/Phylip/random.phy"
>>> results = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, text=True)
```

The ‘stdout = subprocess.PIPE‘ argument makes the output of the program accessible through ‘results.stdout‘ for debugging purposes, (the same can be done for ‘stderr‘), and ‘text=True‘ makes the returned information be a python string, instead of a ‘bytes‘ object.

This generates a tree file and a stats file with the names [input filename]_phyml_tree.txt and [input filename]_phyml_stats.txt. The tree file is in Newick format:

```python
>>> from Bio import Phylo
>>> tree = Phylo.read("Tests/Phylip/random.phy_phyml_tree.txt", "newick")
>>> Phylo.draw_ascii(tree)
```

The subprocess module can also be used for interacting with any other programs that provide a command line interface such as RAxML (https://sco.h-its.org/exelixis/software.html), FastTree (http://www.microbesonline.org/fasttree/), dnaml and protml.
16.6 PAML integration

Biopython 1.58 brought support for PAML (http://abacus.gene.ucl.ac.uk/software/paml.html), a suite of programs for phylogenetic analysis by maximum likelihood. Currently the programs codeml, baseml and yu00 are implemented. Due to PAML’s usage of control files rather than command line arguments to control runtime options, usage of this wrapper strays from the format of other application wrappers in Biopython.

A typical workflow would be to initialize a PAML object, specifying an alignment file, a tree file, an output file and a working directory. Next, runtime options are set via the set_options() method or by reading an existing control file. Finally, the program is run via the run() method and the output file is automatically parsed to a results dictionary.

Here is an example of typical usage of codeml:

```python
>>> from Bio.Phylo.PAML import codeml
>>> cml = codeml.Codeml()
>>> cml.alignment = "Tests/PAML/Alignments/alignment.phylip"
>>> cml.tree = "Tests/PAML/Trees/species.tree"
>>> cml.out_file = "results.out"
>>> cml.working_dir = ".\scratch"
>>> cml.set_options(
...   seqtype=1,
...   verbose=0,
...   noisy=0,
...   RateAncestor=0,
...   model=0,
...   NSsites=[0, 1, 2],
...   CodonFreq=2,
...   cleandata=1,
...   fix_alpha=1,
...   kappa=4.54006,
...   )
>>> results = cml.run()
>>> ns_sites = results.get("NSsites")
>>> m0 = ns_sites.get(0)
>>> m0_params = m0.get("parameters")
>>> print(m0_params.get("omega"))
```

Existing output files may be parsed as well using a module’s read() function:

```python
>>> results = codeml.read("Tests/PAML/Results/codeml/codeml_NSsites_all.out")
>>> print(results.get("lnL max"))
```

Detailed documentation for this new module currently lives on the Biopython wiki: http://biopython.org/wiki/PAML

16.7 Future plans

Bio.Phylo is under active development. Here are some features we might add in future releases:

**New methods** Generally useful functions for operating on Tree or Clade objects appear on the Biopython wiki first, so that casual users can test them and decide if they’re useful before we add them to Bio.Phylo:

http://biopython.org/wiki/Phylo_cookbook
**Bio.Nexus port** Much of this module was written during Google Summer of Code 2009, under the auspices of NESCent, as a project to implement Python support for the phyloXML data format (see 16.4.4). Support for Newick and Nexus formats was added by porting part of the existing Bio.Nexus module to the new classes used by Bio.Phylo.

Currently, Bio.Nexus contains some useful features that have not yet been ported to Bio.Phylo classes — notably, calculating a consensus tree. If you find some functionality lacking in Bio.Phylo, try poking through Bio.Nexus to see if it’s there instead.

We’re open to any suggestions for improving the functionality and usability of this module; just let us know on the mailing list or our bug database.

Finally, if you need additional functionality not yet included in the Phylo module, check if it’s available in another of the high-quality Python libraries for phylogenetics such as DendroPy ([https://dendropy.org/](https://dendropy.org/)) or PyCogent ([http://pycogent.org/](http://pycogent.org/)). Since these libraries also support standard file formats for phylogenetic trees, you can easily transfer data between libraries by writing to a temporary file or StringIO object.
Chapter 17

Sequence motif analysis using Bio.motifs

This chapter gives an overview of the functionality of the Bio.motifs package included in Biopython. It is intended for people who are involved in the analysis of sequence motifs, so I'll assume that you are familiar with basic notions of motif analysis. In case something is unclear, please look at Section 17.10 for some relevant links.

Most of this chapter describes the new Bio.motifs package included in Biopython 1.61 onwards, which is replacing the older Bio.Motif package introduced with Biopython 1.50, which was in turn based on two older former Biopython modules, Bio.AlignAce and Bio.MEME. It provides most of their functionality with a unified motif object implementation.

Speaking of other libraries, if you are reading this you might be interested in TAMO, another python library designed to deal with sequence motifs. It supports more de-novo motif finders, but it is not a part of Biopython and has some restrictions on commercial use.

17.1 Motif objects

Since we are interested in motif analysis, we need to take a look at Motif objects in the first place. For that we need to import the Bio.motifs library:

```python
>>> from Bio import motifs
```

and we can start creating our first motif objects. We can either create a Motif object from a list of instances of the motif, or we can obtain a Motif object by parsing a file from a motif database or motif finding software.

17.1.1 Creating a motif from instances

Suppose we have these instances of a DNA motif:

```python
>>> from Bio.Seq import Seq

>>> instances = [  
...     Seq("TACAA"),  
...     Seq("TACGC"),  
...     Seq("TACAC"),  
...     Seq("TACCC"),  
...     Seq("AACCC"),  
...     Seq("AATGC"),
...     ]
```

```bash
321
```
then we can create a Motif object as follows:

```python
>>> m = motifs.create(instances)
```

The instances from which this motif was created is stored in the `.alignment` property:

```python
>>> print(m.alignment.sequences)
[Seq('TACAA'), Seq('TACGC'), Seq('TACAC'), Seq('TACCC'), Seq('AACCC'), Seq('AATGC'), Seq('AATGC')]
```

Printing the Motif object shows the instances from which it was constructed:

```python
>>> print(m)
TACAA
TACGC
TACAC
TACCC
AACCC
AATGC
AATGC
```

The length of the motif is defined as the sequence length, which should be the same for all instances:

```python
>>> len(m)
5
```

The Motif object has an attribute `.counts` containing the counts of each nucleotide at each position. Printing this counts matrix shows it in an easily readable format:

```python
>>> print(m.counts)

          0  1  2  3  4
A:  3.00  7.00  0.00  2.00  1.00
C:  0.00  0.00  5.00  2.00  6.00
G:  0.00  0.00  0.00  3.00  0.00
T:  4.00  0.00  2.00  0.00  0.00

```

You can access these counts as a dictionary:

```python
>>> m.counts["A"]
[3.0, 7.0, 0.0, 2.0, 1.0]
```

but you can also think of it as a 2D array with the nucleotide as the first dimension and the position as the second dimension:

```python
>>> m.counts["T", 0]
4.0
>>> m.counts["T", 2]
2.0
>>> m.counts["T", 3]
0.0
```

You can also directly access columns of the counts matrix.
Instead of the nucleotide itself, you can also use the index of the nucleotide in the alphabet of the motif:

```python
>>> m.alphabet
'ACGT'
```

```python
>>> m.counts['A', :]
(3.0, 7.0, 0.0, 2.0, 1.0)
```

```python
>>> m.counts[0, :]
(3.0, 7.0, 0.0, 2.0, 1.0)
```

### 17.1.2 Obtaining a consensus sequence

The consensus sequence of a motif is defined as the sequence of letters along the positions of the motif for which the largest value in the corresponding columns of the `.counts` matrix is obtained:

```python
>>> m.consensus
Seq('TACGC')
```

Conversely, the anticonsensus sequence corresponds to the smallest values in the columns of the `.counts` matrix:

```python
>>> m.anticonsensus
Seq('CCATG')
```

Note that there is some ambiguity in the definition of the consensus and anticonsensus sequence if in some columns multiple nucleotides have the maximum or minimum count.

For DNA sequences, you can also ask for a degenerate consensus sequence, in which ambiguous nucleotides are used for positions where there are multiple nucleotides with high counts:

```python
>>> m.degenerate_consensus
Seq('WACVC')
```

Here, W and R follow the IUPAC nucleotide ambiguity codes: W is either A or T, and V is A, C, or G [7]. The degenerate consensus sequence is constructed following the rules specified by Cavener [3].

The `motif.counts.calculate_consensus` method lets you specify in detail how the consensus sequence should be calculated. This method largely follows the conventions of the EMBOSS program `cons`, and takes the following arguments:

- `substitution_matrix` The scoring matrix used when comparing sequences. By default, it is `None`, in which case we simply count the frequency of each letter. Instead of the default value, you can use the substitution matrices available in `Bio.Align.substitution_matrices`. Common choices are BLOSUM62 (also known as EBLOSUM62) for protein, and NUC.4.4 (also known as EDNAFULL) for nucleotides. **NOTE:** Currently, this method has not yet been implemented for values other than the default value `None`.

- `plurality` Threshold value for the number of positive matches, divided by the total count in a column, required to reach consensus. If `substitution_matrix` is `None`, then this argument must also be `None`, and is ignored; a `ValueError` is raised otherwise. If `substitution_matrix` is not `None`, then the default value of the plurality is 0.5.

- `identity` Number of identities, divided by the total count in a column, required to define a consensus value. If the number of identities is less than `identity` multiplied by the total count in a column, then the undefined character (N for nucleotides and X for amino acid sequences) is used in the consensus sequence. **NOTE:** For DNA sequences, if `identity` is 1.0, then only columns of identical letters contribute to the consensus. Default value is zero.
**setcase** threshold for the positive matches, divided by the total count in a column, above which the consensus is upper-case and below which the consensus is in lower-case. By default, this is equal to 0.5.

This is an example:

```python
>>> m.counts.calculate_consensus(identity=0.5, setcase=0.7)
'tACNC'
```

### 17.1.3 Reverse-complementing a motif

We can get the reverse complement of a motif by calling the `reverse_complement` method on it:

```python
>>> r = m.reverse_complement()
>>> r.consensus
Seq('GCGTA')
>>> r.degenerate_consensus
Seq('GBGTW')
>>> print(r)
TTGTA
GCGTA
GTGTA
GGGTA
GGGTT
GCATT
GCATT
```

The reverse complement is only defined for DNA motifs.

### 17.1.4 Slicing a motif

You can slice the motif to obtain a new `Motif` object for the selected positions:

```python
>>> m_sub = m[2:-1]
>>> print(m_sub)
CA
CG
CA
CC
CC
TG
TG
```

```python
>>> m_sub.consensus
Seq('CG')
>>> m_sub.degenerate_consensus
Seq('CV')
```

### 17.1.5 Relative entropy

The relative entropy (or Kullback-Leibler distance) $H_j$ of column $j$ of the motif is defined as $[39, 11]$

$$H_j = \sum_{i=1}^{M} p_{ij} \log \left( \frac{p_{ij}}{b_i} \right)$$

where:
• \( M \) – The number of letters in the alphabet (given by `len(m.alphabet)`);
• \( p_{ij} \) – The observed frequency of letter \( i \), normalized, in the \( j \)-th column (see below);
• \( b_i \) – The background probability of letter \( i \) (given by `m.background[i]`).

The observed frequency \( p_{ij} \) is computed as follows:

\[
p_{ij} = \frac{c_{ij} + k_i}{C_j + k}
\]

where:

• \( c_{ij} \) – the number of times letter \( i \) appears in column \( j \) of the alignment (given by `m.counts[i, j]`);
• \( C_j \) – The total number of letters in column \( j \): \( C_j = \sum_{i=1}^{M} c_{ij} \) (given by `sum(m.counts[:, j])`);
• \( k_i \) – the pseudocount of letter \( i \) (given by `m.pseudocounts[i]`);
• \( k \) – the total pseudocount: \( k = \sum_{i=1}^{M} k_i \) (given by `sum(m.pseudocounts.values())`).

With these definitions, both \( p_{ij} \) and \( b_i \) are normalized to 1:

\[
\sum_{i=1}^{M} p_{ij} = 1
\]

\[
\sum_{i=1}^{M} b_i = 1
\]

The relative entropy is the same as the information content if the background distribution is uniform.

The relative entropy for each column of motif \( m \) can be obtained using the `relative_entropy` property:

```python
>>> m.relative_entropy
array([1.01477186, 2., 1.13687943, 0.44334329, 1.40832722])
```

These values are calculated using the base-2 logarithm, and are therefore in units of bits. The second column (which consists of \( A \) nucleotides only) has the highest relative entropy; the fourth column (which consists of \( A, C, \) or \( G \) nucleotides) has the lowest relative entropy). The relative entropy of the motif can be calculated by summing over the columns:

```python
>>> sum(m.relative_entropy)  # doctest:+ELLIPSIS
6.003321...
```

### 17.1.6 Creating a sequence logo

If we have internet access, we can create a weblogo:

```python
>>> m.weblogo("mymotif.png")
```

We should get our logo saved as a PNG in the specified file.

### 17.2 Reading motifs

Creating motifs from instances by hand is a bit boring, so it’s useful to have some I/O functions for reading and writing motifs. There are not any really well established standards for storing motifs, but there are a couple of formats that are more used than others.
17.2.1 JASPAR

One of the most popular motif databases is JASPAR. In addition to the motif sequence information, the JASPAR database stores a lot of meta-information for each motif. The module Bio.motifs contains a specialized class jaspar.Motif in which this meta-information is represented as attributes:

- **matrix_id** - the unique JASPAR motif ID, e.g. 'MA0004.1'
- **name** - the name of the TF, e.g. 'Arnt'
- **collection** - the JASPAR collection to which the motif belongs, e.g. 'CORE'
- **tf_class** - the structural class of this TF, e.g. 'Zipper-Type'
- **tf_family** - the family to which this TF belongs, e.g. 'Helix-Loop-Helix'
- **species** - the species to which this TF belongs, may have multiple values, these are specified as taxonomy IDs, e.g. 10090
- **tax_group** - the taxonomic supergroup to which this motif belongs, e.g. 'vertebrates'
- **acc** - the accession number of the TF protein, e.g. 'P53762'
- **data_type** - the type of data used to construct this motif, e.g. 'SELEX'
- **medline** - the Pubmed ID of literature supporting this motif, may be multiple values, e.g. 7592839
- **pazar_id** - external reference to the TF in the PAZAR database, e.g. 'TF0000003'
- **comment** - free form text containing notes about the construction of the motif

The jaspar.Motif class inherits from the generic Motif class and therefore provides all the facilities of any of the motif formats — reading motifs, writing motifs, scanning sequences for motif instances etc.

JASPAR stores motifs in several different ways including three different flat file formats and as an SQL database. All of these formats facilitate the construction of a counts matrix. However, the amount of meta information described above that is available varies with the format.

The JASPAR sites format

The first of the three flat file formats contains a list of instances. As an example, these are the beginning and ending lines of the JASPAR Arnt.sites file showing known binding sites of the mouse helix-loop-helix transcription factor Arnt.

```
>MA0004 ARNT 1
CACGTGatgtcctc
>MA0004 ARNT 2
CACGTGgaggtac
>MA0004 ARNT 3
CACGTGccgcgcgc
...
>MA0004 ARNT 18
AACGTGacagccctcc
>MA0004 ARNT 19
AACGTGcacatcgtcc
>MA0004 ARNT 20
aggaatCGCGTGc
```
The parts of the sequence in capital letters are the motif instances that were found to align to each other.

We can create a **Motif** object from these instances as follows:

```python
>>> from Bio import motifs
>>> with open("Arnt.sites") as handle:
...     arnt = motifs.read(handle, "sites")
...     ...
The instances from which this motif was created is stored in the `.alignment` property:

```python
>>> print(arnt.alignment.sequences[:3])
[Seq('CACGTG'), Seq('CACGTG'), Seq('CACGTG')]
>>> for sequence in arnt.alignment.sequences:
...     print(sequence)
...     ...
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
AACGTG
AACGTG
AACGTG
AACGTG
CGCGTG
```

The counts matrix of this motif is automatically calculated from the instances:

```python
>>> print(arnt.counts)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>4.00</td>
<td>19.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>C:</td>
<td>16.00</td>
<td>0.00</td>
<td>20.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>G:</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>20.00</td>
<td>0.00</td>
</tr>
<tr>
<td>T:</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>20.00</td>
</tr>
</tbody>
</table>

```

This format does not store any meta information.

**The JASPAR pfm format**

JASPAR also makes motifs available directly as a count matrix, without the instances from which it was created. This **pfm** format only stores the counts matrix for a single motif. For example, this is the JASPAR file `SRF.pfm` containing the counts matrix for the human SRF transcription factor:
We can create a motif for this count matrix as follows:

```python
>>> with open("SRF.pfm") as handle:
...     srf = motifs.read(handle, "pfm")
...
>>> print(srf.counts)
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>2.00</td>
<td>9.00</td>
<td>0.00</td>
<td>1.00</td>
<td>32.00</td>
<td>3.00</td>
<td>46.00</td>
<td>1.00</td>
<td>43.00</td>
<td>15.00</td>
<td>2.00</td>
</tr>
<tr>
<td>C:</td>
<td>1.00</td>
<td>33.00</td>
<td>45.00</td>
<td>45.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>G:</td>
<td>39.00</td>
<td>2.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>44.00</td>
</tr>
<tr>
<td>T:</td>
<td>4.00</td>
<td>2.00</td>
<td>0.00</td>
<td>0.00</td>
<td>13.00</td>
<td>42.00</td>
<td>0.00</td>
<td>45.00</td>
<td>3.00</td>
<td>30.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

As this motif was created from the counts matrix directly, it has no instances associated with it:

```python
>>> print(srf.alignment)
None
```

We can now ask for the consensus sequence of these two motifs:

```python
>>> print(arnt.counts.consensus)
CACGTG
>>> print(srf.counts.consensus)
GCCCATATATGG
```

As with the instances file, no meta information is stored in this format.

### The JASPAR format `jaspar`

The `jaspar` file format allows multiple motifs to be specified in a single file. In this format each of the motif records consist of a header line followed by four lines defining the counts matrix. The header line begins with a `>` character (similar to the Fasta file format) and is followed by the unique JASPAR matrix ID and the TF name. The following example shows a `jaspar` formatted file containing the three motifs Arnt, RUNX1 and MEF2A:

```
>MA0004.1 Arnt
A [ 4 19 0 0 0 0 ]
C [16 0 20 0 0 0 ]
G [ 0 1 0 20 0 20 ]
T [ 0 0 0 0 20 0 ]

>MA0002.1 RUNX1
A [10 12 4 1 2 2 0 0 0 8 13 ]
C [ 2 2 7 1 0 8 0 0 1 2 2 ]
G [ 3 1 1 0 23 0 26 26 0 0 4 ]
T [11 11 14 24 1 16 0 0 25 16 7 ]

>MA0052.1 MEF2A
A [ 1 0 57 2 9 6 37 2 56 6 ]
C [50 0 1 1 0 0 0 0 0 0 0 ]
G [ 0 0 0 0 0 0 0 2 50 0 ]
T [ 7 58 0 55 49 52 21 56 0 2 ]
```
The motifs are read as follows:

```python
>>> fh = open("jaspar_motifs.txt")
>>> for m in motifs.parse(fh, "jaspar"):
...     print(m)
...
TF name Arnt
Matrix ID MA0004.1
Matrix:
   0 1 2 3 4 5
A:  4.00 19.00 0.00 0.00 0.00 0.00
C: 16.00  0.00 20.00 0.00 0.00 0.00
G:  0.00  1.00 20.00 0.00 20.00 0.00
T:  0.00  0.00 20.00 0.00  0.00 0.00
```

```
TF name RUNX1
Matrix ID MA0002.1
Matrix:
   0 1 2 3 4 5 6 7 8 9 10
A: 10.00 12.00 4.00 1.00 2.00 2.00 0.00 0.00 0.00 8.00 13.00
C:  2.00  2.00 7.00 1.00 0.00 8.00 0.00 0.00 1.00 2.00  2.00
G:  3.00  1.00 1.00 0.00 23.00 0.00 26.00 26.00 0.00 0.00  4.00
T: 11.00 11.00 14.00 24.00 1.00 16.00 0.00 0.00 25.00 16.00  7.00
```

```
TF name MEF2A
Matrix ID MA0052.1
Matrix:
   0 1 2 3 4 5 6 7 8 9
A: 1.00  0.00 57.00 2.00 9.00 6.00 37.00 2.00 56.00 6.00
C: 50.00  0.00 1.00 1.00 0.00 0.00 0.00 0.00 0.00 0.00
G:  0.00  0.00 0.00 0.00 0.00 0.00 0.00 0.00  2.00 50.00
T:  7.00 58.00  0.00 55.00 49.00 52.00 21.00 56.00 0.00  2.00
```

Note that printing a JASPAR motif yields both the counts data and the available meta-information.

**Accessing the JASPAR database**

In addition to parsing these flat file formats, we can also retrieve motifs from a JASPAR SQL database. Unlike the flat file formats, a JASPAR database allows storing of all possible meta information defined in the JASPAR Motif class. It is beyond the scope of this document to describe how to set up a JASPAR database (please see the main JASPAR website). Motifs are read from a JASPAR database using the Bio.motifs.jaspar.db module. First connect to the JASPAR database using the JASPAR5 class which models the the latest JASPAR schema:

```python
>>> from Bio.motifs.jaspar.db import JASPAR5
>>> JASPAR_DB_HOST = "yourhostname"  # fill in these values
>>> JASPAR_DB_NAME = "yourdatabase"
>>> JASPAR_DB_USER = "yourusername"
```
>>> JASPAR_DB_PASS = "yourpassword"

>>> jdb = JASPAR5(
...     host=JASPAR_DB_HOST,
...     name=JASPAR_DB_NAME,
...     user=JASPAR_DB_USER,
...     password=JASPAR_DB_PASS,
... )

Now we can fetch a single motif by its unique JASPAR ID with the `fetch_motif_by_id` method. Note that a JASPAR ID consists of a base ID and a version number separated by a decimal point, e.g. 'MA0004.1'. The `fetch_motif_by_id` method allows you to use either the fully specified ID or just the base ID. If only the base ID is provided, the latest version of the motif is returned.

>>> arnt = jdb.fetch_motif_by_id("MA0004")

Printing the motif reveals that the JASPAR SQL database stores much more meta-information than the flat files:

```python
>>> print(arnt)
TF name Arnt
Matrix ID MA0004.1
Collection CORE
TF class Zipper-Type
TF family Helix-Loop-Helix
Species 10090
Taxonomic group vertebrates
Accession ['P53762']
Data type used SELEX
Medline 7592839
PAZAR ID TF0000003
Comments -
Matrix:
    0  1  2  3  4  5
A: 4.00 19.00 0.00 0.00 0.00 0.00
C: 16.00 0.00 20.00 0.00 0.00 0.00
G: 0.00 1.00 0.00 20.00 0.00 20.00
T: 0.00 0.00 0.00 0.00 20.00 0.00
```

We can also fetch motifs by name. The name must be an exact match (partial matches or database wildcards are not currently supported). Note that as the name is not guaranteed to be unique, the `fetch_motifs_by_name` method actually returns a list.

```python
>>> motifs = jdb.fetch_motifs_by_name("Arnt")

>>> print(motifs[0])
TF name Arnt
Matrix ID MA0004.1
Collection CORE
TF class Zipper-Type
TF family Helix-Loop-Helix
Species 10090
Taxonomic group vertebrates
```
The fetch_motifs method allows you to fetch motifs which match a specified set of criteria. These criteria include any of the above described metadata as well as certain matrix properties such as the minimum information content (min_ic in the example below), the minimum length of the matrix or the minimum number of sites used to construct the matrix. Only motifs which pass ALL the specified criteria are returned. Note that selection criteria which correspond to metadata which allow for multiple values may be specified as either a single value or a list of values, e.g. tax_group and tf_family in the example below.

```python
>>> motifs = jdb.fetch_motifs(
...     collection="CORE",
...     tax_group=['vertebrates', 'insects'],
...     tf_class="Winged Helix-Turn-Helix",
...     tf_family=['Forkhead', 'Ets'],
...     min_ic=12,
... )
>>> for motif in motifs:
...     pass  # do something with the motif
...```

Compatibility with Perl TFBS modules

An important thing to note is that the JASPAR Motif class was designed to be compatible with the popular Perl TFBS modules. Therefore some specifics about the choice of defaults for background and pseudocounts as well as how information content is computed and sequences searched for instances is based on this compatibility criteria. These choices are noted in the specific subsections below.

- **Choice of background:**
  The Perl TFBS modules appear to allow a choice of custom background probabilities (although the documentation states that uniform background is assumed). However the default is to use a uniform background. Therefore it is recommended that you use a uniform background for computing the position-specific scoring matrix (PSSM). This is the default when using the Biopython motifs module.

- **Choice of pseudocounts:**
  By default, the Perl TFBS modules use a pseudocount equal to $\sqrt{N} \times bg[nucleotide]$, where $N$ represents the total number of sequences used to construct the matrix. To apply this same pseudocount formula, set the motif pseudocounts attribute using the jaspar.calculate_pseudocounts() function:

```python
>>> motif.pseudocounts = motifs.jaspar.calculate_pseudocounts(motif)
```
Note that it is possible for the counts matrix to have an unequal number of sequences making up the columns. The pseudocount computation uses the average number of sequences making up the matrix. However, when `normalize` is called on the counts matrix, each count value in a column is divided by the total number of sequences making up that specific column, not by the average number of sequences. This differs from the Perl TFBS modules because the normalization is not done as a separate step and so the average number of sequences is used throughout the computation of the pssm. Therefore, for matrices with unequal column counts, the PSSM computed by the `motifs` module will differ somewhat from the pssm computed by the Perl TFBS modules.

- **Computation of matrix information content:**
  The information content (IC) or specificity of a matrix is computed using the `mean` method of the `PositionSpecificScoringMatrix` class. However of note, in the Perl TFBS modules the default behavior is to compute the IC without first applying pseudocounts, even though by default the PSSMs are computed using pseudocounts as described above.

- **Searching for instances:**
  Searching for instances with the Perl TFBS motifs was usually performed using a relative score threshold, i.e. a score in the range 0 to 1. In order to compute the absolute PSSM score corresponding to a relative score one can use the equation:

  ```
  >>> abs_score = (pssm.max - pssm.min) * rel_score + pssm.min
  ```

  To convert the absolute score of an instance back to a relative score, one can use the equation:

  ```
  >>> rel_score = (abs_score - pssm.min) / (pssm.max - pssm.min)
  ```

  For example, using the Arnt motif before, let’s search a sequence with a relative score threshold of 0.8.

  ```
  >>> test_seq = Seq("TAAGCGTGAGCGAAGCAGTGATTA")
  >>> arnt.pseudocounts = motifs.jaspar.calculate_pseudocounts(arnt)
  >>> pssm = arnt.pssm
  >>> max_score = pssm.max
  >>> min_score = pssm.min
  >>> abs_score_threshold = (max_score - min_score) * 0.8 + min_score
  >>> for pos, score in pssm.search(test_seq, threshold=abs_score_threshold):
  ...     rel_score = (score - min_score) / (max_score - min_score)
  ...     print(f"Position {pos}: score = {score:5.3f}, rel. score = {rel_score:5.3f}")
  ...
  Position 2: score = 5.362, rel. score = 0.801
  Position 8: score = 6.112, rel. score = 0.831
  Position -20: score = 7.103, rel. score = 0.870
  Position 17: score = 10.351, rel. score = 1.000
  Position -11: score = 10.351, rel. score = 1.000
  ```

### 17.2.2 MEME

MEME [2] is a tool for discovering motifs in a group of related DNA or protein sequences. It takes as input a group of DNA or protein sequences and outputs as many motifs as requested. Therefore, in contrast to JASPAR files, MEME output files typically contain multiple motifs. This is an example.

At the top of an output file generated by MEME shows some background information about the MEME and the version of MEME used:

```
********************************************************************************
MEME - Motif discovery tool
********************************************************************************
```
Further down, the input set of training sequences is recapitulated:

********************************************************************************
TRAINING SET
********************************************************************************
DATAFILE= INO_up800.s
ALPHABET= ACGT

<table>
<thead>
<tr>
<th>Sequence name</th>
<th>Weight</th>
<th>Length</th>
<th>Sequence name</th>
<th>Weight</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH01</td>
<td>1.0000</td>
<td>800</td>
<td>CH02</td>
<td>1.0000</td>
<td>800</td>
</tr>
<tr>
<td>FAS1</td>
<td>1.0000</td>
<td>800</td>
<td>FAS2</td>
<td>1.0000</td>
<td>800</td>
</tr>
<tr>
<td>ACC1</td>
<td>1.0000</td>
<td>800</td>
<td>INO1</td>
<td>1.0000</td>
<td>800</td>
</tr>
<tr>
<td>OPI3</td>
<td>1.0000</td>
<td>800</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

and the exact command line that was used:

********************************************************************************
COMMAND LINE SUMMARY
********************************************************************************
This information can also be useful in the event you wish to report a problem with the MEME software.

command: meme -mod oops -dna -revcomp -nmotifs 2 -bfile yeast.nc.6.freq INO_up800.s

Next is detailed information on each motif that was found:

********************************************************************************
MOTIF 1 width = 12 sites = 7 llr = 95 E-value = 2.0e-001
********************************************************************************

Motif 1 Description

Simplified  A ::9:a::3:
pos.-specific  C ::a:9:11691a
probability  G ::::1::94:4:
matrix       T aa:1::9::11:

To parse this file (stored as meme.dna.oops.txt), use

```python
>>> with open("meme.INO_up800.classic.oops.xml") as handle:
...     record = motifs.parse(handle, "meme")
...```

The `motifs.parse` command reads the complete file directly, so you can close the file after calling `motifs.parse`. The header information is stored in attributes:

```python
>>> record.version
'5.0.1'
>>> record.datafile
'common/INO_up800.s'
```
>>> record.command
'meme common/INO_up800.s -oc results/meme10 -mod oops -dna -revcomp -bfile common/yeast.nc.6.freq -nmotifs 2 -objfun classic -minw 8 -nostatus '

>>> record.alphabet
'ACGT'

>>> record.sequences
['sequence_0', 'sequence_1', 'sequence_2', 'sequence_3', 'sequence_4', 'sequence_5', 'sequence_6']

The record is an object of the Bio.motifs.meme.Record class. The class inherits from list, and you can think of record as a list of Motif objects:

>>> len(record)
2

>>> motif = record[0]
>>> print(motif.consensus)
GCGGCATGTGAAA

>>> print(motif.degenerate_consensus)
GSKGCATGTGAAA

In addition to these generic motif attributes, each motif also stores its specific information as calculated by MEME. For example,

>>> motif.num_occurrences
7

>>> motif.length
13

>>> evalue = motif.evalue
>>> print("%3.1g" % evalue)
0.2

>>> motif.name
'GSKGCATGTGAAA'

>>> motif.id
'motif_1'

In addition to using an index into the record, as we did above, you can also find it by its name:

>>> motif = record['GSKGCATGTGAAA']

Each motif has an attribute .alignment with the sequence alignment in which the motif was found, providing some information on each of the sequences:

>>> len(motif.alignment)
7

>>> motif.alignment.sequences[0]
Instance('GCGGCATGTGAAA')

>>> motif.alignment.sequences[0].motif_name
'GSKGCATGTGAAA'

>>> motif.alignment.sequences[0].sequence_name
'INO1'

>>> motif.alignment.sequences[0].sequence_id
'sequence_5'

>>> motif.alignment.sequences[0].start
620

>>> motif.alignment.sequences[0].strand
334
>>> motif.alignment.sequences[0].length
13
>>> pvalue = motif.alignment.sequences[0].pvalue
>>> print("%5.3g" % pvalue)
1.21e-08

MAST

17.2.3 TRANSFAC

TRANSFAC is a manually curated database of transcription factors, together with their genomic binding sites and DNA binding profiles [32]. While the file format used in the TRANSFAC database is nowadays also used by others, we will refer to it as the TRANSFAC file format.

A minimal file in the TRANSFAC format looks as follows:

```
ID motif1
P0  A  C  G  T
 01  1  2  2  0  S
 02  2  1  2  0  R
 03  3  0  1  1  A
 04  0  5  0  0  C
 05  5  0  0  0  A
 06  0  0  4  1  G
 07  0  1  4  0  G
 08  0  0  0  5  T
 09  0  0  5  0  G
10  0  1  2  2  K
11  0  2  0  3  Y
12  1  0  3  1  G
//
```

This file shows the frequency matrix of motif `motif1` of 12 nucleotides. In general, one file in the TRANSFAC format can contain multiple motifs. For example, this is the contents of the example TRANSFAC file `transfac.dat`:

```
VV EXAMPLE January 15, 2013
XX
//
ID motif1
P0  A  C  G  T
 01  1  2  2  0  S
 02  2  1  2  0  R
 03  3  0  1  1  A
... 11  0  2  0  3  Y
12  1  0  3  1  G
//
ID motif2
P0  A  C  G  T
 01  2  1  2  0  R
 02  1  2  2  0  S
...```

335
To parse a TRANSFAC file, use

```python
>>> with open("transfac.dat") as handle:
...     record = motifs.parse(handle, "TRANSFAC")
...     ...
```

If any discrepancies between the file contents and the TRANSFAC file format are detected, a `ValueError` is raised. Note that you may encounter files that do not follow the TRANSFAC format strictly. For example, the number of spaces between columns may be different, or a tab may be used instead of spaces. Use `strict=False` to enable parsing such files without raising a `ValueError`:

```python
>>> record = motifs.parse(handle, "TRANSFAC", strict=False)
```

When parsing a non-compliant file, we recommend to check the record returned by `motif.parse` to ensure that it is consistent with the file contents.

The overall version number, if available, is stored as `record.version`:

```python
>>> record.version
'EXAMPLE January 15, 2013'
```

Each motif in `record` is in instance of the `Bio.motifs.transfac.Motif` class, which inherits both from the `Bio.motifs.Motif` class and from a Python dictionary. The dictionary uses the two-letter keys to store any additional information about the motif:

```python
>>> motif = record[0]
>>> motif.degenerate_consensus  # Using the Bio.motifs.Motif property
Seq('SRACAGGTGKYG')
>>> motif["ID"]  # Using motif as a dictionary
'motif1'
```

TRANSFAC files are typically much more elaborate than this example, containing lots of additional information about the motif. Table 17.2.3 lists the two-letter field codes that are commonly found in TRANSFAC files:

Each motif also has an attribute `.references` containing the references associated with the motif, using these two-letter keys:

Printing the motifs writes them out in their native TRANSFAC format:

```python
>>> print(record)
VV EXAMPLE January 15, 2013
XX
//
ID motif1
XX
P0  A  C  G  T
01  1  2  2  0  S
02  2  1  2  0  R
03  3  0  1  1  A
04  0  5  0  0  C
05  5  0  0  0  A
06  0  0  4  1  G
```

336
Table 17.1: Fields commonly found in TRANSFAC files

| AC | Accession number          |
|    |                          |
| AS | Accession numbers, secondary |
| BA | Statistical basis        |
| BF | Binding factors          |
| BS | Factor binding sites underlying the matrix |
| CC | Comments                 |
| CD | Copyright notice         |
| DE | Short factor description |
| DR | External databases       |
| DT | Date created/updated     |
| HC | Subfamilies              |
| HP | Superfamilies            |
| ID | Identifier               |
| NA | Name of the binding factor |
| OC | Taxonomic classification |
| OS | Species/Taxon            |
| OV | Older version            |
| PV | Preferred version        |
| TY | Type                     |
| XX | Empty line; these are not stored in the Record. |

Table 17.2: Fields used to store references in TRANSFAC files

| RN | Reference number |
|    |                  |
| RA | Reference authors |
| RL | Reference data   |
| RT | Reference title  |
| RX | PubMed ID        |

|07  | 0  | 1  | 4  | 0  | G |
|08  | 0  | 0  | 0  | 5  | T |
|09  | 0  | 0  | 5  | 0  | G |
|10  | 0  | 1  | 2  | 2  | K |
|11  | 0  | 2  | 0  | 3  | Y |
|12  | 1  | 0  | 3  | 1  | G |

//

ID  motif2

XX

XX

P0  A  C  G  T
00  2  1  2  0  R
01  1  2  2  0  S
02  0  5  0  0  C
03  3  0  1  1  A
04  0  0  4  1  G
05  5  0  0  0  A
06  0  1  4  0  G
07  0  0  5  0  G
You can export the motifs in the TRANSFAC format by capturing this output in a string and saving it in a file:

```python
>>> text = str(record)
>>> with open("mytransfacfile.dat", "w") as out_handle:
...    out_handle.write(text)
...```

17.3 Writing motifs

Speaking of exporting, let’s look at export functions in general. We can use the `format` built-in function to write the motif in the simple JASPAR pfm format:

```python
>>> print(format(arnt, "pfm"))
4.00 19.00 0.00 0.00 0.00 0.00
16.00 0.00 20.00 0.00 0.00 0.00
0.00 1.00 0.00 20.00 0.00 20.00
0.00 0.00 0.00 0.00 20.00 0.00
```

Similarly, we can use `format` to write the motif in the JASPAR jaspar format:

```python
>>> print(format(arnt, "jaspar"))
>MA0004.1 Arnt
A [ 4.00 19.00 0.00 0.00 0.00 0.00]
C [ 16.00 0.00 20.00 0.00 0.00 0.00]
G [ 0.00 1.00 0.00 20.00 0.00 20.00]
T [ 0.00 0.00 0.00 0.00 20.00 0.00]
```

To write the motif in a TRANSFAC-like matrix format, use

```python
>>> print(format(m, "transfac"))
P0 A C G T
01 4 16 0 0 C
```

To write out multiple motifs, you can use `motifs.write`. This function can be used regardless of whether the motifs originated from a TRANSFAC file. For example,

```python
>>> two_motifs = [arnt, srf]
>>> print(motifs.write(two_motifs, "transfac"))
P0 A C G T
01 4 16 0 0 C
```
Or, to write multiple motifs in the jaspar format:

```python
>>> two_motifs = [arnt, mef2a]
>>> print(motifs.write(two_motifs, "jaspar"))
>MA0004.1 Arnt
A [ 4.00 19.00 0.00 0.00 0.00 0.00]
C [16.00 20.00 0.00 0.00 0.00 0.00]
G [ 0.00  1.00 0.00 20.00 0.00 20.00]
T [ 0.00  0.00 0.00  0.00 20.00 0.00]
>MA0052.1 MEF2A
A [ 1.00  0.00 57.00  2.00  9.00 37.00]
C [ 50.00  0.00 1.00  1.00  0.00  0.00]
G [ 0.00  0.00  0.00  0.00  0.00 50.00]
T [ 7.00 58.00  0.00 55.00  49.00 52.00]
```

17.4 Position-Weight Matrices

The `.counts` attribute of a Motif object shows how often each nucleotide appeared at each position along the alignment. We can normalize this matrix by dividing by the number of instances in the alignment, resulting in the probability of each nucleotide at each position along the alignment. We refer to these probabilities as the position-weight matrix. However, beware that in the literature this term may also be used to refer to the position-specific scoring matrix, which we discuss below.

Usually, pseudocounts are added to each position before normalizing. This avoids overfitting of the position-weight matrix to the limited number of motif instances in the alignment, and can also prevent probabilities from becoming zero. To add a fixed pseudocount to all nucleotides at all positions, specify a number for the `pseudocounts` argument:

```python
>>> pwm = m.counts.normalize(pseudocounts=0.5)
>>> print(pwm)
```
Alternatively, *pseudocounts* can be a dictionary specifying the pseudocounts for each nucleotide. For example, as the GC content of the human genome is about 40%, you may want to choose the pseudocounts accordingly:

```python
>>> pwm = m.counts.normalize(pseudocounts={"A": 0.6, "C": 0.4, "G": 0.4, "T": 0.6})
>>> print(pwm)
0 1 2 3 4
A: 0.40 0.84 0.07 0.29 0.18
C: 0.04 0.04 0.60 0.27 0.71
G: 0.04 0.04 0.04 0.38 0.04
T: 0.51 0.07 0.29 0.07 0.07
```

The position-weight matrix has its own methods to calculate the consensus, anticonsensus, and degenerate consensus sequences:

```python
>>> pwm.consensus
Seq('TACGC')
>>> pwm.anticonsensus
Seq('CCGTG')
>>> pwm.degenerate_consensus
Seq('WACNC')
```

Note that due to the pseudocounts, the degenerate consensus sequence calculated from the position-weight matrix is slightly different from the degenerate consensus sequence calculated from the instances in the motif:

```python
>>> m.degenerate_consensus
Seq('WACVC')
```

The reverse complement of the position-weight matrix can be calculated directly from the `pwm`:

```python
>>> rpwm = pwm.reverse_complement()
>>> print(rpwm)
0 1 2 3 4
A: 0.07 0.07 0.29 0.07 0.51
C: 0.04 0.38 0.04 0.04 0.04
G: 0.71 0.27 0.60 0.04 0.04
T: 0.18 0.29 0.07 0.84 0.40
```

### 17.5 Position-Specific Scoring Matrices

Using the background distribution and PWM with pseudo-counts added, it’s easy to compute the log-odds ratios, telling us what are the log odds of a particular symbol to be coming from a motif against the background. We can use the `.log_odds()` method on the position-weight matrix:
Here we can see positive values for symbols more frequent in the motif than in the background and negative for symbols more frequent in the background. 0.0 means that it's equally likely to see a symbol in the background and in the motif.

This assumes that A, C, G, and T are equally likely in the background. To calculate the position-specific scoring matrix against a background with unequal probabilities for A, C, G, T, use the background argument. For example, against a background with a 40% GC content, use

```python
>>> background = {"A": 0.3, "C": 0.2, "G": 0.2, "T": 0.3}
>>> pssm = pwm.log_odds(background)
>>> print(pssm)

A:  0.42 1.49 -2.17 -0.05 -0.75
C: -2.17 -2.17 1.58  0.42  1.83
G: -2.17 -2.17 -2.17  0.92 -2.17
T:  0.77 -2.17 -0.05 -2.17 -2.17

The maximum and minimum score obtainable from the PSSM are stored in the .max and .min properties:

```python
>>> print("%4.2f" % pssm.max)
6.59
>>> print("%4.2f" % pssm.min)
-10.85
```

The mean and standard deviation of the PSSM scores with respect to a specific background are calculated by the .mean and .std methods.

```python
>>> mean = pssm.mean(background)
>>> std = pssm.std(background)
>>> print("mean = %0.2f, standard deviation = %0.2f" % (mean, std))
mean = 3.21, standard deviation = 2.59
```

A uniform background is used if background is not specified. The mean is equal to the Kullback-Leibler divergence or relative entropy described in Section 17.1.5.

The .reverse_complement, .consensus, .anticonsensus, and .degenerate_consensus methods can be applied directly to PSSM objects.

### 17.6 Searching for instances

The most frequent use for a motif is to find its instances in some sequence. For the sake of this section, we will use an artificial sequence like this:

```python
>>> test_seq = Seq("TACACTGCATTACAACCCAAGCATTA")
>>> len(test_seq)
26
```
17.6.1 Searching for exact matches

The simplest way to find instances, is to look for exact matches of the true instances of the motif:

```python
>>> for pos, seq in test_seq.search(m.alignment):
...     print("%i %s" % (pos, seq))
...
0  TACAC
10 TACAA
13 AACCC
```

We can do the same with the reverse complement (to find instances on the complementary strand):

```python
>>> for pos, seq in test_seq.search(r.alignment):
...     print("%i %s" % (pos, seq))
...
6  GCATT
20 GCATT
```

17.6.2 Searching for matches using the PSSM score

It’s just as easy to look for positions, giving rise to high log-odds scores against our motif:

```python
>>> for position, score in pssm.search(test_seq, threshold=3.0):
...     print("Position %d: score = %5.3f" % (position, score))
...
Position 0: score = 5.622
Position -20: score = 4.601
Position 10: score = 3.037
Position 13: score = 5.738
Position -6: score = 4.601
```

The negative positions refer to instances of the motif found on the reverse strand of the test sequence, and follow the Python convention on negative indices. Therefore, the instance of the motif at pos is located at `test_seq[pos:pos+len(m)]` both for positive and for negative values of pos.

You may notice the threshold parameter, here set arbitrarily to 3.0. This is in log2, so we are now looking only for words, which are eight times more likely to occur under the motif model than in the background. The default threshold is 0.0, which selects everything that looks more like the motif than the background.

You can also calculate the scores at all positions along the sequence:

```python
>>> pssm.calculate(test_seq)
```

```
array([ 5.62230396, -5.6796999 , -3.43177247, 0.93827754,
         -6.84962511, -2.04066086, -10.84962463, -3.65614533,
         -0.03370807, -3.91102552, 3.03734159, -2.14918518,
         -0.6016975 , 5.7381525 , -0.50977498, -3.56422281,
         -8.73414803, -0.09919716, -0.6016975 , -2.39429784,
         -10.84962463, -3.65614533], dtype=float32)
```

In general, this is the fastest way to calculate PSSM scores. The scores returned by `pssm.calculate` are for the forward strand only. To obtain the scores on the reverse strand, you can take the reverse complement of the PSSM:

```python
>>> rpssm = pssm.reverse_complement()
>>> rpssm.calculate(test_seq)
```
17.6.3 Selecting a score threshold

If you want to use a less arbitrary way of selecting thresholds, you can explore the distribution of PSSM scores. Since the space for a score distribution grows exponentially with motif length, we are using an approximation with a given precision to keep computation cost manageable:

```python
>>> distribution = pssm.distribution(background=background, precision=10**4)
```

The `distribution` object can be used to determine a number of different thresholds. We can specify the requested false-positive rate (probability of “finding” a motif instance in background generated sequence):

```python
>>> threshold = distribution.threshold_fpr(0.01)
>>> print("%5.3f" % threshold)
4.009
```

or the false-negative rate (probability of “not finding” an instance generated from the motif):

```python
>>> threshold = distribution.threshold_fnr(0.1)
>>> print("%5.3f" % threshold)
-0.510
```

or a threshold (approximately) satisfying some relation between the false-positive rate and the false-negative rate ($\frac{\text{fnr}}{\text{fpr}} \approx t$):

```python
>>> threshold = distribution.threshold_balanced(1000)
>>> print("%5.3f" % threshold)
6.241
```

or a threshold satisfying (roughly) the equality between the $-\log$ of the false-positive rate and the information content (as used in patser software by Hertz and Stormo):

```python
>>> threshold = distribution.threshold_patser()
>>> print("%5.3f" % threshold)
0.346
```

For example, in case of our motif, you can get the threshold giving you exactly the same results (for this sequence) as searching for instances with balanced threshold with rate of 1000.

```python
>>> threshold = distribution.threshold_fpr(0.01)
>>> print("%5.3f" % threshold)
4.009
```

```python
>>> for position, score in pssm.search(test_seq, threshold=threshold):
...     print("Position %d: score = %5.3f" % (position, score))
...
Position 0: score = 5.622
Position -20: score = 4.601
Position 13: score = 5.738
Position -6: score = 4.601
```
17.7 Each motif object has an associated Position-Specific Scoring Matrix

To facilitate searching for potential TFBSs using PSSMs, both the position-weight matrix and the position-specific scoring matrix are associated with each motif. Using the Arnt motif as an example:

```python
>>> from Bio import motifs
>>> with open("Arnt.sites") as handle:
...     motif = motifs.read(handle, "sites")
...
>>> print(motif.counts)
 0 1 2 3 4 5
A:  4.00  19.00  0.00  0.00  0.00  0.00
C:  16.00  0.00  20.00  0.00  0.00  0.00
G:  0.00  1.00  0.00  20.00  0.00  20.00
T:  0.00  0.00  0.00  0.00  20.00  0.00

>>> print(motif.pwm)
 0 1 2 3 4 5
A:  0.20  0.95  0.00  0.00  0.00  0.00
C:  0.80  0.00  1.00  0.00  0.00  0.00
G:  0.00  0.05  0.00  1.00  0.00  1.00
T:  0.00  0.00  0.00  0.00  1.00  0.00

>>> print(motif.pssm)
 0 1 2 3 4 5
A:  -0.32  1.93 -Inf -Inf -Inf -Inf
C:  1.68  -Inf  2.00 -Inf -Inf -Inf
G:  -Inf  -2.32 -Inf  2.00 -Inf  2.00
T:  -Inf  -Inf -Inf -Inf  2.00 -Inf
```

The negative infinities appear here because the corresponding entry in the frequency matrix is 0, and we are using zero pseudocounts by default:

```python
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.pseudocounts[letter]))
...
A: 0.00
C: 0.00
G: 0.00
T: 0.00
```

If you change the `.pseudocounts` attribute, the position-frequency matrix and the position-specific scoring matrix are recalculated automatically:

```python
>>> motif.pseudocounts = 3.0
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.pseudocounts[letter]))
...
A: 3.00
C: 3.00
```
>>> print(motif.pwm)

    0  1  2  3  4  5
A:  0.22 0.69 0.09 0.09 0.09 0.09
C:  0.59 0.09 0.72 0.09 0.09 0.09
G:  0.09 0.12 0.09 0.72 0.09 0.72
T:  0.09 0.09 0.09 0.09 0.72 0.09

>>> print(motif.pssm)

    0  1  2  3  4  5
A: -0.19 1.46 -1.42 -1.42 -1.42 -1.42
C:  1.25 -1.42 1.52 -1.42 -1.42 -1.42
G: -1.42 -1.00 -1.42 1.52 -1.42 1.52
T: -1.42 -1.42 -1.42 -1.42 1.52 -1.42

You can also set the \texttt{.pseudocounts} to a dictionary over the four nucleotides if you want to use different pseudocounts for them. Setting \texttt{motif.pseudocounts} to \texttt{None} resets it to its default value of zero.

The position-specific scoring matrix depends on the background distribution, which is uniform by default:

```python
>>> for letter in "ACGT":
...     print("%s: %.2f" % (letter, motif.background[letter]))
...
A: 0.25
C: 0.25
G: 0.25
T: 0.25
```

Again, if you modify the background distribution, the position-specific scoring matrix is recalculated:

```python
>>> motif.background = {"A": 0.2, "C": 0.3, "G": 0.3, "T": 0.2}
```

```
>>> print(motif.pssm)

    0  1  2  3  4  5
A:  0.13 1.78 -1.09 -1.09 -1.09 -1.09
C:  0.98 -1.68 1.26 -1.68 -1.68 -1.68
G: -1.68 -1.26 -1.68 1.26 -1.68 1.26
T: -1.09 -1.09 -1.09 1.85 -1.09
```

Setting \texttt{motif.background} to \texttt{None} resets it to a uniform distribution:

```python
>>> motif.background = None
```

```
>>> for letter in "ACGT":
...     print("%s: %.2f" % (letter, motif.background[letter]))
...
A: 0.25
C: 0.25
G: 0.25
T: 0.25
```

If you set \texttt{motif.background} equal to a single value, it will be interpreted as the GC content:
Note that you can now calculate the mean of the PSSM scores over the background against which it was computed:

```python
>>> print("%f %f" % motif.pssm.mean(motif.background))
4.703928
```
as well as its standard deviation:

```python
>>> print("%f %f" % motif.pssm.std(motif.background))
3.290900
```
and its distribution:

```python
>>> distribution = motif.pssm.distribution(background=motif.background)
>>> threshold = distribution.threshold_fpr(0.01)
>>> print("%f %f" % threshold)
3.854375
```

Note that the position-weight matrix and the position-specific scoring matrix are recalculated each time you call `motif.pwm` or `motif.pssm`, respectively. If speed is an issue and you want to use the PWM or PSSM repeatedly, you can save them as a variable, as in

```python
>>> pssm = motif.pssm
```

### 17.8 Comparing motifs

Once we have more than one motif, we might want to compare them.

Before we start comparing motifs, I should point out that motif boundaries are usually quite arbitrary. This means we often need to compare motifs of different lengths, so comparison needs to involve some kind of alignment. This means we have to take into account two things:

- alignment of motifs
- some function to compare aligned motifs

To align the motifs, we use ungapped alignment of PSSMs and substitute zeros for any missing columns at the beginning and end of the matrices. This means that effectively we are using the background distribution for columns missing from the PSSM. The distance function then returns the minimal distance between motifs, as well as the corresponding offset in their alignment.

To give an example, let us first load another motif, which is similar to our test motif `m`:

```python
>>> with open("REB1.pfm") as handle:
...   m_reb1 = motifs.read(handle, "pfm")
...```

```python
>>> m_reb1.consensus
346
```
To make the motifs comparable, we choose the same values for the pseudocounts and the background distribution as our motif \( m \):

\[
\begin{align*}
\text{m_reb1.pseudocounts} & = \{"A": 0.6, "C": 0.4, "G": 0.4, "T": 0.6\} \\
\text{m_reb1.background} & = \{"A": 0.3, "C": 0.2, "G": 0.2, "T": 0.3\}
\end{align*}
\]

We'll compare these motifs using the Pearson correlation. Since we want it to resemble a distance measure, we actually take \( 1 - r \), where \( r \) is the Pearson correlation coefficient (PCC):

\[
\text{distance, offset} = \text{pssm.dist_pearson(pssm_reb1)}
\]

\[
\text{print("distance = \%5.3g" % distance)}
\]

\[
\text{distance} = 0.239
\]

\[
\text{print(offset)}
\]

\[
-2
\]

This means that the best PCC between motif \( m \) and \( m_{reb1} \) is obtained with the following alignment:

\[
\begin{align*}
m: & \quad bbTACGCbb \\
m_{reb1}: & \quad GTTACCCGG
\end{align*}
\]

where \( b \) stands for background distribution. The PCC itself is roughly \( 1 - 0.239 = 0.761 \).

## 17.9 De novo motif finding

Currently, Biopython has only limited support for *de novo* motif finding. Namely, we support running xxmotif and also parsing of MEME. Since the number of motif finding tools is growing rapidly, contributions of new parsers are welcome.

### 17.9.1 MEME

Let's assume, you have run MEME on sequences of your choice with your favorite parameters and saved the output in the file `meme.out`. You can retrieve the motifs reported by MEME by running the following piece of code:

\[
\text{from Bio import motifs}
\]

\[
\text{with open("meme.psp_test.classic.zoops.xml") as handle:}
\]

\[
\quad \text{motifsM = motifs.parse(handle, "meme")}
\]

\[
\ldots
\]
Besides the most wanted list of motifs, the result object contains more useful information, accessible through properties with self-explanatory names:

- .alphabet
- .datafile
- .sequences
- .version
- .command

The motifs returned by the MEME Parser can be treated exactly like regular Motif objects (with instances), they also provide some extra functionality, by adding additional information about the instances.

```python
>>> motifsM[0].consensus
Seq('GCTTATGTAA')
>>> motifsM[0].alignment.sequences[0].sequence_name
'iYFL005W'
>>> motifsM[0].alignment.sequences[0].sequence_id
'sequence_15'
>>> motifsM[0].alignment.sequences[0].start
480
>>> motifsM[0].alignment.sequences[0].strand
'+

>>> motifsM[0].alignment.sequences[0].pvalue
1.97e-06
```

### 17.10 Useful links

- [Sequence motif in wikipedia](#)
- [PWM in wikipedia](#)
- [Consensus sequence in wikipedia](#)
- [Comparison of different motif finding programs](#)
Chapter 18

Cluster analysis

Cluster analysis is the grouping of items into clusters based on the similarity of the items to each other. In bioinformatics, clustering is widely used in gene expression data analysis to find groups of genes with similar gene expression profiles. This may identify functionally related genes, as well as suggest the function of presently unknown genes.

The Biopython module Bio.Cluster provides commonly used clustering algorithms and was designed with the application to gene expression data in mind. However, this module can also be used for cluster analysis of other types of data. Bio.Cluster and the underlying C Clustering Library is described by De Hoon et al. [10].

The following four clustering approaches are implemented in Bio.Cluster:

- Hierarchical clustering (pairwise centroid-, single-, complete-, and average-linkage);
- \( k \)-means, \( k \)-medians, and \( k \)-medoids clustering;
- Self-Organizing Maps;
- Principal Component Analysis.

Data representation

The data to be clustered are represented by a \( n \times m \) Numerical Python array \( data \). Within the context of gene expression data clustering, typically the rows correspond to different genes whereas the columns correspond to different experimental conditions. The clustering algorithms in Bio.Cluster can be applied both to rows (genes) and to columns (experiments).

Missing values

The \( n \times m \) Numerical Python integer array \( mask \) indicates if any of the values in \( data \) are missing. If \( mask[i, j] = 0 \), then \( data[i, j] \) is missing and is ignored in the analysis.

Random number generator

The \( k \)-means/medians/medoids clustering algorithms and Self-Organizing Maps (SOMs) include the use of a random number generator. The uniform random number generator in Bio.Cluster is based on the algorithm by L’Ecuyer [27], while random numbers following the binomial distribution are generated using the BTPE algorithm by Kachitvichyanukul and Schmeiser [23]. The random number generator is initialized automatically during its first call. As this random number generator uses a combination of two multiplicative linear congruential generators, two (integer) seeds are needed for initialization, for which we use the system-supplied random number generator \( rand \) (in the C standard library). We initialize this generator by calling
\texttt{srand} with the epoch time in seconds, and use the first two random numbers generated by \texttt{rand} as seeds for the uniform random number generator in \texttt{Bio.Cluster}.

### 18.1 Distance functions

In order to cluster items into groups based on their similarity, we should first define what exactly we mean by similar. \texttt{Bio.Cluster} provides eight distance functions, indicated by a single character, to measure similarity, or conversely, distance:

- 'e': Euclidean distance;
- 'b': City-block distance.
- 'c': Pearson correlation coefficient;
- 'a': Absolute value of the Pearson correlation coefficient;
- 'u': Uncentered Pearson correlation (equivalent to the cosine of the angle between two data vectors);
- 'x': Absolute uncentered Pearson correlation;
- 's': Spearman’s rank correlation;
- 'k': Kendall’s $\tau$.

The first two are true distance functions that satisfy the triangle inequality:

$$d(u, v) \leq d(u, w) + d(w, v) \text{ for all } u, v, w,$$

and are therefore referred to as \textit{metrics}. In everyday language, this means that the shortest distance between two points is a straight line.

The remaining six distance measures are related to the correlation coefficient, where the distance $d$ is defined in terms of the correlation $r$ by $d = 1 - r$. Note that these distance functions are \textit{semi-metrics} that do not satisfy the triangle inequality. For example, for

- $u = (1, 0, -1)$;
- $v = (1, 1, 0)$;
- $w = (0, 1, 1)$;

we find a Pearson distance $d(u, w) = 1.8660$, while $d(u, v) + d(v, w) = 1.6340$.

#### Euclidean distance

In \texttt{Bio.Cluster}, we define the Euclidean distance as

$$d = \frac{1}{n} \sum_{i=1}^{n} (x_i - y_i)^2.$$

Only those terms are included in the summation for which both $x_i$ and $y_i$ are present, and the denominator $n$ is chosen accordingly. As the expression data $x_i$ and $y_i$ are subtracted directly from each other, we should make sure that the expression data are properly normalized when using the Euclidean distance.
City-block distance

The city-block distance, alternatively known as the Manhattan distance, is related to the Euclidean distance. Whereas the Euclidean distance corresponds to the length of the shortest path between two points, the city-block distance is the sum of distances along each dimension. As gene expression data tend to have missing values, in Bio.Cluster we define the city-block distance as the sum of distances divided by the number of dimensions:

\[ d = \frac{1}{n} \sum_{i=1}^{n} |x_i - y_i|. \]

This is equal to the distance you would have to walk between two points in a city, where you have to walk along city blocks. As for the Euclidean distance, the expression data are subtracted directly from each other, and we should therefore make sure that they are properly normalized.

The Pearson correlation coefficient

The Pearson correlation coefficient is defined as

\[ r = \frac{1}{n} \sum_{i=1}^{n} \left( \frac{x_i - \bar{x}}{\sigma_x} \right) \left( \frac{y_i - \bar{y}}{\sigma_y} \right), \]

in which \( \bar{x}, \bar{y} \) are the sample mean of \( x \) and \( y \) respectively, and \( \sigma_x, \sigma_y \) are the sample standard deviation of \( x \) and \( y \). The Pearson correlation coefficient is a measure for how well a straight line can be fitted to a scatterplot of \( x \) and \( y \). If all the points in the scatterplot lie on a straight line, the Pearson correlation coefficient is either +1 or -1, depending on whether the slope of line is positive or negative. If the Pearson correlation coefficient is equal to zero, there is no correlation between \( x \) and \( y \).

The Pearson distance is then defined as

\[ d_P = 1 - r. \]

As the Pearson correlation coefficient lies between -1 and 1, the Pearson distance lies between 0 and 2.

Absolute Pearson correlation

By taking the absolute value of the Pearson correlation, we find a number between 0 and 1. If the absolute value is 1, all the points in the scatter plot lie on a straight line with either a positive or a negative slope. If the absolute value is equal to zero, there is no correlation between \( x \) and \( y \).

The corresponding distance is defined as

\[ d_A = 1 - |r|, \]

where \( r \) is the Pearson correlation coefficient. As the absolute value of the Pearson correlation coefficient lies between 0 and 1, the corresponding distance lies between 0 and 1 as well.

In the context of gene expression experiments, the absolute correlation is equal to 1 if the gene expression profiles of two genes are either exactly the same or exactly opposite. The absolute correlation coefficient should therefore be used with care.

Uncentered correlation (cosine of the angle)

In some cases, it may be preferable to use the uncentered correlation instead of the regular Pearson correlation coefficient. The uncentered correlation is defined as

\[ r_U = \frac{1}{n} \sum_{i=1}^{n} \left( \frac{x_i}{\sigma_x} \right) \left( \frac{y_i}{\sigma_y} \right), \]
where
\[ \sigma_x^{(0)} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i^2}, \]
\[ \sigma_y^{(0)} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} y_i^2}. \]

This is the same expression as for the regular Pearson correlation coefficient, except that the sample means \( \bar{x}, \bar{y} \) are set equal to zero. The uncentered correlation may be appropriate if there is a zero reference state. For instance, in the case of gene expression data given in terms of log-ratios, a log-ratio equal to zero corresponds to the green and red signal being equal, which means that the experimental manipulation did not affect the gene expression.

The distance corresponding to the uncentered correlation coefficient is defined as
\[ d_U \equiv 1 - r_U, \]
where \( r_U \) is the uncentered correlation. As the uncentered correlation coefficient lies between -1 and 1, the corresponding distance lies between 0 and 2.

The uncentered correlation is equal to the cosine of the angle of the two data vectors in \( n \)-dimensional space, and is often referred to as such.

**Absolute uncentered correlation**

As for the regular Pearson correlation, we can define a distance measure using the absolute value of the uncentered correlation:
\[ d_{AU} \equiv 1 - |r_U|, \]
where \( r_U \) is the uncentered correlation coefficient. As the absolute value of the uncentered correlation coefficient lies between 0 and 1, the corresponding distance lies between 0 and 1 as well.

Geometrically, the absolute value of the uncentered correlation is equal to the cosine between the supporting lines of the two data vectors (i.e., the angle without taking the direction of the vectors into consideration).

**Spearman rank correlation**

The Spearman rank correlation is an example of a non-parametric similarity measure, and tends to be more robust against outliers than the Pearson correlation.

To calculate the Spearman rank correlation, we replace each data value by their rank if we would order the data in each vector by their value. We then calculate the Pearson correlation between the two rank vectors instead of the data vectors.

As in the case of the Pearson correlation, we can define a distance measure corresponding to the Spearman rank correlation as
\[ d_S \equiv 1 - r_S, \]
where \( r_S \) is the Spearman rank correlation.

**Kendall’s \( \tau \)**

Kendall’s \( \tau \) is another example of a non-parametric similarity measure. It is similar to the Spearman rank correlation, but instead of the ranks themselves only the relative ranks are used to calculate \( \tau \) (see Snedecor & Cochran [43]).

We can define a distance measure corresponding to Kendall’s \( \tau \) as
\[ d_K \equiv 1 - \tau. \]
As Kendall’s \( \tau \) is always between -1 and 1, the corresponding distance will be between 0 and 2.
Weighting

For most of the distance functions available in Bio.Cluster, a weight vector can be applied. The weight vector contains weights for the items in the data vector. If the weight for item $i$ is $w_i$, then that item is treated as if it occurred $w_i$ times in the data. The weight do not have to be integers.

Calculating the distance matrix

The distance matrix is a square matrix with all pairwise distances between the items in data, and can be calculated by the function distancematrix in the Bio.Cluster module:

```python
>>> from Bio.Cluster import distancematrix
>>> matrix = distancematrix(data)
```

where the following arguments are defined:

- **data** (required)
  Array containing the data for the items.

- **mask** (default: None)
  Array of integers showing which data are missing. If mask[i, j] == 0, then data[i, j] is missing. If mask is None, then all data are present.

- **weight** (default: None)
  The weights to be used when calculating distances. If weight is None, then equal weights are assumed.

- **transpose** (default: 0)
  Determines if the distances between the rows of data are to be calculated (transpose is False), or between the columns of data (transpose is True).

- **dist** (default: 'e', Euclidean distance)
  Defines the distance function to be used (see 18.1).

To save memory, the distance matrix is returned as a list of 1D arrays. The number of columns in each row is equal to the row number. Hence, the first row has zero elements. For example,

```python
>>> from numpy import array
>>> from Bio.Cluster import distancematrix
>>> data = array([[0, 1, 2, 3],
...                 [4, 5, 6, 7],
...                 [8, 9, 10, 11],
...                 [1, 2, 3, 4]])  # fmt: skip
>>> distances = distancematrix(data, dist="e")
```

yields a distance matrix

```python
[arary((), dtype=float64), array([16.]), array([64., 16.]), array([1., 9., 49.])]
```

which can be rewritten as

```python
[arary((), dtype=float64), array([16.0]), array([64.0, 16.0]), array([1.0, 9.0, 49.0])]
```

This corresponds to the distance matrix:

\[
\begin{pmatrix}
0 & 16 & 64 & 1 \\
16 & 0 & 16 & 9 \\
64 & 16 & 0 & 49 \\
1 & 9 & 49 & 0
\end{pmatrix}
\]

353
18.2 Calculating cluster properties

Calculating the cluster centroids

The centroid of a cluster can be defined either as the mean or as the median of each dimension over all cluster items. The function `clustercentroids` in `Bio.Cluster` can be used to calculate either:

```python
>>> from Bio.Cluster import clustercentroids
>>> cdata, cmask = clustercentroids(data)
```

where the following arguments are defined:

- **data** (required)
  Array containing the data for the items.

- **mask** (default: None)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.

- **clusterid** (default: None)
  Vector of integers showing to which cluster each item belongs. If `clusterid` is `None`, then all items are assumed to belong to the same cluster.

- **method** (default: 'a')
  Specifies whether the arithmetic mean (`method='a'`) or the median (`method='m'`) is used to calculate the cluster center.

- **transpose** (default: 0)
  Determines if the centroids of the rows of `data` are to be calculated (`transpose` is `False`), or the centroids of the columns of `data` (`transpose` is `True`).

This function returns the tuple `(cdata, cmask)`. The centroid data are stored in the 2D Numerical Python array `cdata`, with missing data indicated by the 2D Numerical Python integer array `cmask`. The dimensions of these arrays are (number of clusters, number of columns) if `transpose` is `0`, or (number of rows, number of clusters) if `transpose` is `1`. Each row (if `transpose` is `0`) or column (if `transpose` is `1`) contains the averaged data corresponding to the centroid of each cluster.

Calculating the distance between clusters

Given a distance function between `items`, we can define the distance between two `clusters` in several ways. The distance between the arithmetic means of the two clusters is used in pairwise centroid-linkage clustering and in k-means clustering. In k-medoids clustering, the distance between the medians of the two clusters is used instead. The shortest pairwise distance between items of the two clusters is used in pairwise single-linkage clustering, while the longest pairwise distance is used in pairwise maximum-linkage clustering. In pairwise average-linkage clustering, the distance between two clusters is defined as the average over the pairwise distances.

To calculate the distance between two clusters, use

```python
>>> from Bio.Cluster import clusterdistance
>>> distance = clusterdistance(data)
```

where the following arguments are defined:

- **data** (required)
  Array containing the data for the items.
• **mask** (default: None)
  Array of integers showing which data are missing. If \( \text{mask}[i, j] == 0 \), then \( \text{data}[i, j] \) is missing. If \( \text{mask} \) is None, then all data are present.

• **weight** (default: None)
  The weights to be used when calculating distances. If \( \text{weight} \) is None, then equal weights are assumed.

• **index1** (default: 0)
  A list containing the indices of the items belonging to the first cluster. A cluster containing only one item \( i \) can be represented either as a list \([i]\) or as an integer \( i \).

• **index2** (default: 0)
  A list containing the indices of the items belonging to the second cluster. A cluster containing only one items \( i \) can be represented either as a list \([i]\) or as an integer \( i \).

• **method** (default: 'a')
  Specifies how the distance between clusters is defined:
  - 'a': Distance between the two cluster centroids (arithmetic mean);
  - 'm': Distance between the two cluster centroids (median);
  - 's': Shortest pairwise distance between items in the two clusters;
  - 'x': Longest pairwise distance between items in the two clusters;
  - 'v': Average over the pairwise distances between items in the two clusters.

• **dist** (default: 'e', Euclidean distance)
  Defines the distance function to be used (see 18.1).

• **transpose** (default: 0)
  If \( \text{transpose} \) is False, calculate the distance between the rows of \( \text{data} \). If \( \text{transpose} \) is True, calculate the distance between the columns of \( \text{data} \).

### 18.3 Partitioning algorithms

Partitioning algorithms divide items into \( k \) clusters such that the sum of distances over the items to their cluster centers is minimal. The number of clusters \( k \) is specified by the user. Three partitioning algorithms are available in Bio.Cluster:

- \( k \)-means clustering
- \( k \)-medians clustering
- \( k \)-medoids clustering

These algorithms differ in how the cluster center is defined. In \( k \)-means clustering, the cluster center is defined as the mean data vector averaged over all items in the cluster. Instead of the mean, in \( k \)-medians clustering the median is calculated for each dimension in the data vector. Finally, in \( k \)-medoids clustering the cluster center is defined as the item which has the smallest sum of distances to the other items in the cluster. This clustering algorithm is suitable for cases in which the distance matrix is known but the original data matrix is not available, for example when clustering proteins based on their structural similarity.

The expectation-maximization (EM) algorithm is used to find this partitioning into \( k \) groups. In the initialization of the EM algorithm, we randomly assign items to clusters. To ensure that no empty clusters are produced, we use the binomial distribution to randomly choose the number of items in each cluster to be one or more. We then randomly permute the cluster assignments to items such that each item has an equal probability to be in any cluster. Each cluster is thus guaranteed to contain at least one item.

We then iterate:
• Calculate the centroid of each cluster, defined as either the mean, the median, or the medoid of the cluster;
• Calculate the distances of each item to the cluster centers;
• For each item, determine which cluster centroid is closest;
• Reassign each item to its closest cluster, or stop the iteration if no further item reassignments take place.

To avoid clusters becoming empty during the iteration, in k-means and k-medians clustering the algorithm keeps track of the number of items in each cluster, and prohibits the last remaining item in a cluster from being reassigned to a different cluster. For k-medoids clustering, such a check is not needed, as the item that functions as the cluster centroid has a zero distance to itself, and will therefore never be closer to a different cluster.

As the initial assignment of items to clusters is done randomly, usually a different clustering solution is found each time the EM algorithm is executed. To find the optimal clustering solution, the k-means algorithm is repeated many times, each time starting from a different initial random clustering. The sum of distances of the items to their cluster center is saved for each run, and the solution with the smallest value of this sum will be returned as the overall clustering solution.

How often the EM algorithm should be run depends on the number of items being clustered. As a rule of thumb, we can consider how often the optimal solution was found; this number is returned by the partitioning algorithms as implemented in this library. If the optimal solution was found many times, it is unlikely that better solutions exist than the one that was found. However, if the optimal solution was found only once, there may well be other solutions with a smaller within-cluster sum of distances. If the number of items is large (more than several hundreds), it may be difficult to find the globally optimal solution.

The EM algorithm terminates when no further reassignments take place. We noticed that for some sets of initial cluster assignments, the EM algorithm fails to converge due to the same clustering solution reappearing periodically after a small number of iteration steps. We therefore check for the occurrence of such periodic solutions during the iteration. After a given number of iteration steps, the current clustering result is saved as a reference. By comparing the clustering result after each subsequent iteration step to the reference state, we can determine if a previously encountered clustering result is found. In such a case, the iteration is halted. If after a given number of iterations the reference state has not yet been encountered, the current clustering solution is saved to be used as the new reference state. Initially, ten iteration steps are executed before resaving the reference state. This number of iteration steps is doubled each time, to ensure that periodic behavior with longer periods can also be detected.

**k-means and k-medians**

The k-means and k-medians algorithms are implemented as the function `kcluster` in Bio.Cluster:

```python
>>> from Bio.Cluster import kcluster
>>> clusterid, error, nfound = kcluster(data)
```

where the following arguments are defined:

- **data** (required)
  Array containing the data for the items.
- **nclusters** (default: 2)
  The number of clusters $k$.
- **mask** (default: None)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.
• **weight** (default: *None*)
The weights to be used when calculating distances. If **weight** is *None*, then equal weights are assumed.

• **transpose** (default: 0)
Determines if rows (**transpose** is 0) or columns (**transpose** is 1) are to be clustered.

• **npass** (default: 1)
The number of times the *k*-means/-medians clustering algorithm is performed, each time with a different (random) initial condition. If **initialid** is given, the value of **npass** is ignored and the clustering algorithm is run only once, as it behaves deterministically in that case.

• **method** (default: a)
describes how the center of a cluster is found:
  - **method**=='a': arithmetic mean (*k*-means clustering);
  - **method**=='m': median (*k*-medians clustering).

For other values of **method**, the arithmetic mean is used.

• **dist** (default: ‘e’, Euclidean distance)
Defines the distance function to be used (see 18.1). Whereas all eight distance measures are accepted by *kcluster*, from a theoretical viewpoint it is best to use the Euclidean distance for the *k*-means algorithm, and the city-block distance for *k*-medians.

• **initialid** (default: *None*)
Specifies the initial clustering to be used for the EM algorithm. If **initialid** is *None*, then a different random initial clustering is used for each of the **npass** runs of the EM algorithm. If **initialid** is not *None*, then it should be equal to a 1D array containing the cluster number (between 0 and **nclusters**-1) for each item. Each cluster should contain at least one item. With the initial clustering specified, the EM algorithm is deterministic.

This function returns a tuple (**clusterid**, **error**, **nfound**), where **clusterid** is an integer array containing the number of the cluster to which each row or cluster was assigned, **error** is the within-cluster sum of distances for the optimal clustering solution, and **nfound** is the number of times this optimal solution was found.

**k-medoids clustering**
The **kmedoids** routine performs *k*-medoids clustering on a given set of items, using the distance matrix and the number of clusters passed by the user:

```python
>>> from Bio.Cluster import kmedoids
>>> clusterid, error, nfound = kmedoids(distance)
```

where the following arguments are defined: , **nclusters**=2, **npass**=1, **initialid**=*None*)—

• **distance** (required)
The matrix containing the distances between the items; this matrix can be specified in three ways:
  - as a 2D Numerical Python array (in which only the left-lower part of the array will be accessed):
    ```python
distance = array([[0.0, 1.1, 2.3], [1.1, 0.0, 4.5], [2.3, 4.5, 0.0]])
```
  - as a 1D Numerical Python array containing consecutively the distances in the left-lower part of the distance matrix:
    ```python
distance = array([1.1, 2.3, 4.5])
```
– as a list containing the rows of the left-lower part of the distance matrix:

\[
\text{distance} = [\text{array([[]])}, \text{array([1.1])}, \text{array([2.3, 4.5])}]
\]

These three expressions correspond to the same distance matrix.

- **nclusters** (default: 2)
  The number of clusters \( k \).

- **npass** (default: 1)
  The number of times the \( k \)-medoids clustering algorithm is performed, each time with a different (random) initial condition. If \text{initialid} is given, the value of \text{npass} is ignored, as the clustering algorithm behaves deterministically in that case.

- **initialid** (default: None)
  Specifies the initial clustering to be used for the EM algorithm. If \text{initialid} is None, then a different random initial clustering is used for each of the \text{npass} runs of the EM algorithm. If \text{initialid} is not None, then it should be equal to a 1D array containing the cluster number (between 0 and \text{nclusters}-1) for each item. Each cluster should contain at least one item. With the initial clustering specified, the EM algorithm is deterministic.

This function returns a tuple \((\text{clusterid}, \text{error}, \text{nfound})\), where \text{clusterid} is an array containing the number of the cluster to which each item was assigned, \text{error} is the within-cluster sum of distances for the optimal \( k \)-medoids clustering solution, and \text{nfound} is the number of times the optimal solution was found. Note that the cluster number in \text{clusterid} is defined as the item number of the item representing the cluster centroid.

### 18.4 Hierarchical clustering

Hierarchical clustering methods are inherently different from the \( k \)-means clustering method. In hierarchical clustering, the similarity in the expression profile between genes or experimental conditions are represented in the form of a tree structure. This tree structure can be shown graphically by programs such as Treeview and Java Treeview, which has contributed to the popularity of hierarchical clustering in the analysis of gene expression data.

The first step in hierarchical clustering is to calculate the distance matrix, specifying all the distances between the items to be clustered. Next, we create a node by joining the two closest items. Subsequent nodes are created by pairwise joining of items or nodes based on the distance between them, until all items belong to the same node. A tree structure can then be created by retracing which items and nodes were merged. Unlike the EM algorithm, which is used in \( k \)-means clustering, the complete process of hierarchical clustering is deterministic.

Several flavors of hierarchical clustering exist, which differ in how the distance between subnodes is defined in terms of their members. In \text{Bio.Cluster}, pairwise single, maximum, average, and centroid linkage are available.

- In pairwise single-linkage clustering, the distance between two nodes is defined as the shortest distance among the pairwise distances between the members of the two nodes.

- In pairwise maximum-linkage clustering, alternatively known as pairwise complete-linkage clustering, the distance between two nodes is defined as the longest distance among the pairwise distances between the members of the two nodes.

- In pairwise average-linkage clustering, the distance between two nodes is defined as the average over all pairwise distances between the items of the two nodes.
In pairwise centroid-linkage clustering, the distance between two nodes is defined as the distance between their centroids. The centroids are calculated by taking the mean over all the items in a cluster. As the distance from each newly formed node to existing nodes and items need to be calculated at each step, the computing time of pairwise centroid-linkage clustering may be significantly longer than for the other hierarchical clustering methods. Another peculiarity is that (for a distance measure based on the Pearson correlation), the distances do not necessarily increase when going up in the clustering tree, and may even decrease. This is caused by an inconsistency between the centroid calculation and the distance calculation when using the Pearson correlation: Whereas the Pearson correlation effectively normalizes the data for the distance calculation, no such normalization occurs for the centroid calculation.

For pairwise single-, complete-, and average-linkage clustering, the distance between two nodes can be found directly from the distances between the individual items. Therefore, the clustering algorithm does not need access to the original gene expression data, once the distance matrix is known. For pairwise centroid-linkage clustering, however, the centroids of newly formed subnodes can only be calculated from the original data and not from the distance matrix.

The implementation of pairwise single-linkage hierarchical clustering is based on the SLINK algorithm \[41\], which is much faster and more memory-efficient than a straightforward implementation of pairwise single-linkage clustering. The clustering result produced by this algorithm is identical to the clustering solution found by the conventional single-linkage algorithm. The single-linkage hierarchical clustering algorithm implemented in this library can be used to cluster large gene expression data sets, for which conventional hierarchical clustering algorithms fail due to excessive memory requirements and running time.

Representing a hierarchical clustering solution

The result of hierarchical clustering consists of a tree of nodes, in which each node joins two items or subnodes. Usually, we are not only interested in which items or subnodes are joined at each node, but also in their similarity (or distance) as they are joined. To store one node in the hierarchical clustering tree, we make use of the class \texttt{Node}, which defined in \texttt{Bio.Cluster}. An instance of \texttt{Node} has three attributes:

- \texttt{left}
- \texttt{right}
- \texttt{distance}

Here, \texttt{left} and \texttt{right} are integers referring to the two items or subnodes that are joined at this node, and \texttt{distance} is the distance between them. The items being clustered are numbered from 0 to (number of items – 1), while clusters are numbered from -1 to – (number of items – 1). Note that the number of nodes is one less than the number of items.

To create a new \texttt{Node} object, we need to specify \texttt{left} and \texttt{right}; \texttt{distance} is optional.

```python
>>> from Bio.Cluster import Node
>>> Node(2, 3)
(2, 3): 0
>>> Node(2, 3, 0.91)
(2, 3): 0.91
```

The attributes \texttt{left}, \texttt{right}, and \texttt{distance} of an existing \texttt{Node} object can be modified directly:

```python
>>> node = Node(4, 5)
>>> node.left = 6
>>> node.right = 2
>>> node.distance = 0.73
>>> node
(6, 2): 0.73
```
An error is raised if left and right are not integers, or if distance cannot be converted to a floating-point value.

The Python class Tree represents a full hierarchical clustering solution. A Tree object can be created from a list of Node objects:

```python
>>> from Bio.Cluster import Node, Tree
>>> nodes = [Node(1, 2, 0.2), Node(0, 3, 0.5), Node(-2, 4, 0.6), Node(-1, -3, 0.9)]
>>> tree = Tree(nodes)
>>> print(tree)
(1, 2): 0.2
(0, 3): 0.5
(-2, 4): 0.6
(-1, -3): 0.9
```

The Tree initializer checks if the list of nodes is a valid hierarchical clustering result:

```python
>>> nodes = [Node(1, 2, 0.2), Node(0, 2, 0.5)]
>>> Tree(nodes)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Inconsistent tree
```

Individual nodes in a Tree object can be accessed using square brackets:

```python
>>> nodes = [Node(1, 2, 0.2), Node(0, -1, 0.5)]
>>> tree = Tree(nodes)
>>> tree[0]
(1, 2): 0.2
>>> tree[1]
(0, -1): 0.5
>>> tree[-1]
(0, -1): 0.5
```

As a Tree object is immutable, we cannot change individual nodes in a Tree object. However, we can convert the tree to a list of nodes, modify this list, and create a new tree from this list:

```python
>>> tree = Tree([Node(1, 2, 0.1), Node(0, -1, 0.5), Node(-2, 3, 0.9)])
>>> print(tree)
(1, 2): 0.1
(0, -1): 0.5
(-2, 3): 0.9
>>> nodes = tree[:]
>>> nodes[0] = Node(0, 1, 0.2)
>>> nodes[1].left = 2
>>> tree = Tree(nodes)
>>> print(tree)
(0, 1): 0.2
(2, -1): 0.5
(-2, 3): 0.9
```

This guarantees that any Tree object is always well-formed.

To display a hierarchical clustering solution with visualization programs such as Java Treeview, it is better to scale all node distances such that they are between zero and one. This can be accomplished by calling the scale method on an existing Tree object:
>>> tree.scale()

This method takes no arguments, and returns None.

Before drawing the tree, you may also want to reorder the tree nodes. A hierarchical clustering solution of $n$ items can be drawn as $2^{n-1}$ different but equivalent dendrograms by switching the left and right subnode at each node. The tree.sort(order) method visits each node in the hierarchical clustering tree and verifies if the average order value of the left subnode is less than or equal to the average order value of the right subnode. If not, the left and right subnodes are exchanged. Here, the order values of the items are given by the user. In the resulting dendrogram, items in the left-to-right order will tend to have increasing order values. The method will return the indices of the elements in the left-to-right order after sorting:

```python
>>> indices = tree.sort(order)
```

such that item indices[i] will occur at position i in the dendrogram.

After hierarchical clustering, the items can be grouped into $k$ clusters based on the tree structure stored in the Tree object by cutting the tree:

```python
>>> clusterid = tree.cut(nclusters=1)
```

where nclusters (defaulting to 1) is the desired number of clusters $k$. This method ignores the top $k-1$ linking events in the tree structure, resulting in $k$ separated clusters of items. The number of clusters $k$ should be positive, and less than or equal to the number of items. This method returns an array clusterid containing the number of the cluster to which each item is assigned. Clusters are numbered 0 to $k-1$ in their left-to-right order in the dendrogram.

### Performing hierarchical clustering

To perform hierarchical clustering, use the treeccluster function in Bio.Cluster.

```python
>>> from Bio.Cluster import treeccluster
>>> tree = treeccluster(data)
```

where the following arguments are defined:

- **data**
  Array containing the data for the items.

- **mask** (default: None)
  Array of integers showing which data are missing. If mask[i, j] == 0, then data[i, j] is missing. If mask is None, then all data are present.

- **weight** (default: None)
  The weights to be used when calculating distances. If weight is None, then equal weights are assumed.

- **transpose** (default: 0)
  Determines if rows (transpose is False) or columns (transpose is True) are to be clustered.

- **method** (default: 'm')
  defines the linkage method to be used:
  - method='s': pairwise single-linkage clustering
  - method='m': pairwise maximum- (or complete-) linkage clustering
  - method='c': pairwise centroid-linkage clustering
  - method='a': pairwise average-linkage clustering
• **dist** (default: 'e', Euclidean distance)
  Defines the distance function to be used (see 18.1).

To apply hierarchical clustering on a precalculated distance matrix, specify the `distancematrix` argument when calling `treecluster` function instead of the `data` argument:

```python
>>> from Bio.Cluster import treecluster
>>> tree = treecluster(distancematrix=distance)
```

In this case, the following arguments are defined:

- **distancematrix**
  The distance matrix, which can be specified in three ways:
  - as a 2D Numerical Python array (in which only the left-lower part of the array will be accessed):
    ```python
distance = array([[0.0, 1.1, 2.3], [1.1, 0.0, 4.5], [2.3, 4.5, 0.0]])
```
  - as a 1D Numerical Python array containing consecutively the distances in the left-lower part of the distance matrix:
    ```python
distance = array([1.1, 2.3, 4.5])
```
  - as a list containing the rows of the left-lower part of the distance matrix:
    ```python
distance = [array([]), array([1.1]), array([2.3, 4.5])]
```

These three expressions correspond to the same distance matrix. As `treecluster` may shuffle the values in the distance matrix as part of the clustering algorithm, be sure to save this array in a different variable before calling `treecluster` if you need it later.

- **method**
  The linkage method to be used:
  - `method=='s'`: pairwise single-linkage clustering
  - `method=='m'`: pairwise maximum- (or complete-) linkage clustering
  - `method=='a'`: pairwise average-linkage clustering

While pairwise single-, maximum-, and average-linkage clustering can be calculated from the distance matrix alone, pairwise centroid-linkage cannot.

When calling `treecluster`, either `data` or `distancematrix` should be `None`.

This function returns a `Tree` object. This object contains (number of items − 1) nodes, where the number of items is the number of rows if rows were clustered, or the number of columns if columns were clustered. Each node describes a pairwise linking event, where the node attributes `left` and `right` each contain the number of one item or subnode, and `distance` the distance between them. Items are numbered from 0 to (number of items − 1), while clusters are numbered -1 to −(number of items − 1).

### 18.5 Self-Organizing Maps

Self-Organizing Maps (SOMs) were invented by Kohonen to describe neural networks (see for instance Kohonen, 1997 [25]). Tamayo (1999) first applied Self-Organizing Maps to gene expression data [46].

SOMs organize items into clusters that are situated in some topology. Usually a rectangular topology is chosen. The clusters generated by SOMs are such that neighboring clusters in the topology are more similar to each other than clusters far from each other in the topology.

The first step to calculate a SOM is to randomly assign a data vector to each cluster in the topology. If rows are being clustered, then the number of elements in each data vector is equal to the number of columns.
An SOM is then generated by taking rows one at a time, and finding which cluster in the topology has the closest data vector. The data vector of that cluster, as well as those of the neighboring clusters, are adjusted using the data vector of the row under consideration. The adjustment is given by

\[ \Delta \mathbf{x}_{\text{cell}} = \tau \cdot (\mathbf{x}_{\text{row}} - \mathbf{x}_{\text{cell}}). \]

The parameter \( \tau \) is a parameter that decreases at each iteration step. We have used a simple linear function of the iteration step:

\[ \tau = \tau_{\text{init}} \cdot \left( 1 - \frac{i}{n} \right), \]

\( \tau_{\text{init}} \) is the initial value of \( \tau \) as specified by the user, \( i \) is the number of the current iteration step, and \( n \) is the total number of iteration steps to be performed. While changes are made rapidly in the beginning of the iteration, at the end of iteration only small changes are made.

All clusters within a radius \( R \) are adjusted to the gene under consideration. This radius decreases as the calculation progresses as

\[ R = R_{\text{max}} \cdot \left( 1 - \frac{i}{n} \right), \]

in which the maximum radius is defined as

\[ R_{\text{max}} = \sqrt{N_x^2 + N_y^2}, \]

where \((N_x, N_y)\) are the dimensions of the rectangle defining the topology.

The function `somcluster` implements the complete algorithm to calculate a Self-Organizing Map on a rectangular grid. First it initializes the random number generator. The node data are then initialized using the random number generator. The order in which genes or samples are used to modify the SOM is also randomized. The total number of iterations in the SOM algorithm is specified by the user.

To run `somcluster`, use

```python
>>> from Bio.Cluster import somcluster
>>> clusterid, celldata = somcluster(data)
```

where the following arguments are defined:

- **data** (required)
  Array containing the data for the items.

- **mask** (default: None)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is None, then all data are present.

- **weight** (default: None)
  contains the weights to be used when calculating distances. If `weight` is None, then equal weights are assumed.

- **transpose** (default: 0)
  Determines if rows (transpose is 0) or columns (transpose is 1) are to be clustered.

- **nxgrid, nygrid** (default: 2, 1)
  The number of cells horizontally and vertically in the rectangular grid on which the Self-Organizing Map is calculated.

- **inittau** (default: 0.02)
  The initial value for the parameter \( \tau \) that is used in the SOM algorithm. The default value for `inittau` is 0.02, which was used in Michael Eisen’s Cluster/TreeView program.
• **niter** (default: 1)
  The number of iterations to be performed.

• **dist** (default: 'e', Euclidean distance)
  Defines the distance function to be used (see 18.1).

This function returns the tuple \( \text{clusterid}, \text{celldata} \):

• **clusterid**:
  An array with two columns, where the number of rows is equal to the number of items that were
  clustered. Each row contains the \( x \) and \( y \) coordinates of the cell in the rectangular SOM grid to which
  the item was assigned.

• **celldata**:
  An array with dimensions \((\text{nxgrid}, \text{nygrid}, \text{number of columns})\) if rows are being clustered, or \((\text{nxgrid}, \text{nygrid}, \text{number of rows})\)
  if columns are being clustered. Each element \([ix][iy]\) of this array is a 1D vector containing the gene
  expression data for the centroid of the cluster in the grid cell with coordinates \([ix][iy]\).

---

**18.6 Principal Component Analysis**

Principal Component Analysis (PCA) is a widely used technique for analyzing multivariate data. A practical
example of applying Principal Component Analysis to gene expression data is presented by Yeung and Ruzzo
(2001) [51].

In essence, PCA is a coordinate transformation in which each row in the data matrix is written as a linear
sum over basis vectors called principal components, which are ordered and chosen such that each maximally
explains the remaining variance in the data vectors. For example, an \( n \times 3 \) data matrix can be represented
as an ellipsoidal cloud of \( n \) points in three dimensional space. The first principal component is the longest
axis of the ellipsoid, the second principal component the second longest axis of the ellipsoid, and the third
principal component is the shortest axis. Each row in the data matrix can be reconstructed as a suitable
linear combination of the principal components. However, in order to reduce the dimensionality of the data,
usually only the most important principal components are retained. The remaining variance present in the
data is then regarded as unexplained variance.

The principal components can be found by calculating the eigenvectors of the covariance matrix of the
data. The corresponding eigenvalues determine how much of the variance present in the data is explained
by each principal component.

Before applying principal component analysis, typically the mean is subtracted from each column in the
data matrix. In the example above, this effectively centers the ellipsoidal cloud around its centroid in 3D
space, with the principal components describing the variation of points in the ellipsoidal cloud with respect
to their centroid.

The function `pca` below first uses the singular value decomposition to calculate the eigenvalues and
eigenvectors of the data matrix. The singular value decomposition is implemented as a translation in C of
the Algol procedure `svd` [14], which uses Householder bidiagonalization and a variant of the QR algorithm.
The principal components, the coordinates of each data vector along the principal components, and the
eigenvalues corresponding to the principal components are then evaluated and returned in decreasing order
of the magnitude of the eigenvalue. If data centering is desired, the mean should be subtracted from each
column in the data matrix before calling the `pca` routine.

To apply Principal Component Analysis to a rectangular matrix `data`, use

```python
>>> from Bio.Cluster import pca
>>> columnmean, coordinates, components, eigenvalues = pca(data)
```

This function returns a tuple `columnmean, coordinates, components, eigenvalues`:
- columnmean
  Array containing the mean over each column in data.

- coordinates
  The coordinates of each row in data with respect to the principal components.

- components
  The principal components.

- eigenvalues
  The eigenvalues corresponding to each of the principal components.

The original matrix data can be recreated by calculating columnmean + dot(coordinates, components).

18.7 Handling Cluster/TreeView-type files
Cluster/TreeView are GUI-based codes for clustering gene expression data. They were originally written by Michael Eisen while at Stanford University [12]. Bio.Cluster contains functions for reading and writing data files that correspond to the format specified for Cluster/TreeView. In particular, by saving a clustering result in that format, TreeView can be used to visualize the clustering results. We recommend using Alok Saldanha's http://jtreeview.sourceforge.net/ Java TreeView program [38], which can display hierarchical as well as k-means clustering results.

An object of the class Record contains all information stored in a Cluster/TreeView-type data file. To store the information contained in the data file in a Record object, we first open the file and then read it:

```python
>>> from Bio import Cluster
>>> with open("mydatafile.txt") as handle:
...     record = Cluster.read(handle)
...
```

This two-step process gives you some flexibility in the source of the data. For example, you can use

```python
>>> import gzip  # Python standard library
>>> handle = gzip.open("mydatafile.txt.gz", "rt")
```
to open a gzipped file, or

```python
>>> from urllib.request import urlopen
>>> from io import TextIOWrapper
>>> url = "https://raw.githubusercontent.com/biopython/biopython/master/Tests/Cluster/cyano.txt"
>>> handle = TextIOWrapper(urlopen(url))
```
to open a file stored on the Internet before calling read.

The read command reads the tab-delimited text file mydatafile.txt containing gene expression data in the format specified for Michael Eisen's Cluster/TreeView program. In this file format, rows represent genes and columns represent samples or observations. For a simple time course, a minimal input file would look like this:

Each row (gene) has an identifier that always goes in the first column. In this example, we are using yeast open reading frame codes. Each column (sample) has a label in the first row. In this example, the labels describe the time at which a sample was taken. The first column of the first row contains a special field that tells the program what kind of objects are in each row. In this case, YORF stands for yeast open reading frame. This field can be any alphanumeric value. The remaining cells in the table contain data for the appropriate gene and sample. The 5.8 in row 2 column 4 means that the observed value for gene YAL001C
<table>
<thead>
<tr>
<th>YORF</th>
<th>0 minutes</th>
<th>30 minutes</th>
<th>1 hour</th>
<th>2 hours</th>
<th>4 hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAL001C</td>
<td>1</td>
<td>1.3</td>
<td>2.4</td>
<td>5.8</td>
<td>2.4</td>
</tr>
<tr>
<td>YAL002W</td>
<td>0.9</td>
<td>0.8</td>
<td>0.7</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>YAL003W</td>
<td>0.8</td>
<td>2.1</td>
<td>4.2</td>
<td>10.1</td>
<td>10.1</td>
</tr>
<tr>
<td>YAL005C</td>
<td>1.1</td>
<td>1.3</td>
<td>0.8</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>YAL010C</td>
<td>1.2</td>
<td>1</td>
<td>1.1</td>
<td>4.5</td>
<td>8.3</td>
</tr>
</tbody>
</table>

at 2 hours was 5.8. Missing values are acceptable and are designated by empty cells (e.g. YAL004C at 2 hours).

The input file may contain additional information. A maximal input file would look like this:
The added columns NAME, GWEIGHT, and GORDER and rows EWEIGHT and EORDER are optional. The NAME column allows you to specify a label for each gene that is distinct from the ID in column 1.

A Record object has the following attributes:

- **data**
  The data array containing the gene expression data. Genes are stored row-wise, while samples are stored column-wise.

- **mask**
  This array shows which elements in the data array, if any, are missing. If \( \text{mask}[i, j] = 0 \), then \( \text{data}[i, j] \) is missing. If no data were found to be missing, \( \text{mask} \) is set to `None`.

- **geneid**
  This is a list containing a unique description for each gene (i.e., ORF numbers).

- **genename**
  This is a list containing a description for each gene (i.e., gene name). If not present in the data file, \( \text{genename} \) is set to `None`.

- **gweight**
  The weights that are to be used to calculate the distance in expression profile between genes. If not present in the data file, \( \text{gweight} \) is set to `None`.

- **gorder**
  The preferred order in which genes should be stored in an output file. If not present in the data file, \( \text{gorder} \) is set to `None`.

- **expid**
  This is a list containing a description of each sample, e.g. experimental condition.

- **eweight**
  The weights that are to be used to calculate the distance in expression profile between samples. If not present in the data file, \( \text{eweight} \) is set to `None`.
• **eorder**
The preferred order in which samples should be stored in an output file. If not present in the data file, `eorder` is set to `None`.

• **uniqid**
The string that was used instead of UNIQID in the data file.

After loading a `Record` object, each of these attributes can be accessed and modified directly. For example, the data can be log-transformed by taking the logarithm of `record.data`.

**Calculating the distance matrix**
To calculate the distance matrix between the items stored in the record, use

```python
>>> matrix = record.distancematrix()
```

where the following arguments are defined:

- **transpose** (default: 0)
  Determines if the distances between the rows of `data` are to be calculated (`transpose` is `False`), or between the columns of `data` (`transpose` is `True`).

- **dist** (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see 18.1).

This function returns the distance matrix as a list of rows, where the number of columns of each row is equal to the row number (see section 18.1).

**Calculating the cluster centroids**
To calculate the centroids of clusters of items stored in the record, use

```python
>>> cdata, cmask = record.clustercentroids()
```

- **clusterid** (default: `None`)
  Vector of integers showing to which cluster each item belongs. If `clusterid` is not given, then all items are assumed to belong to the same cluster.

- **method** (default: `'a'`)
  Specifies whether the arithmetic mean (`method=='a'`) or the median (`method=='m'`) is used to calculate the cluster center.

- **transpose** (default: 0)
  Determines if the centroids of the rows of `data` are to be calculated (`transpose` is `False`), or the centroids of the columns of `data` (`transpose` is `True`).

This function returns the tuple `cdata, cmask`; see section 18.2 for a description.

**Calculating the distance between clusters**
To calculate the distance between clusters of items stored in the record, use

```python
>>> distance = record.clusterdistance()
```

where the following arguments are defined:
- **index1** (default: 0)
  A list containing the indices of the items belonging to the first cluster. A cluster containing only one item $i$ can be represented either as a list [i], or as an integer i.

- **index2** (default: 0)
  A list containing the indices of the items belonging to the second cluster. A cluster containing only one item $i$ can be represented either as a list [i], or as an integer i.

- **method** (default: 'a')
  Specifies how the distance between clusters is defined:
  - 'a': Distance between the two cluster centroids (arithmetic mean);
  - 'm': Distance between the two cluster centroids (median);
  - 's': Shortest pairwise distance between items in the two clusters;
  - 'x': Longest pairwise distance between items in the two clusters;
  - 'v': Average over the pairwise distances between items in the two clusters.

- **dist** (default: 'e', Euclidean distance)
  Defines the distance function to be used (see 18.1).

- **transpose** (default: 0)
  If transpose is False, calculate the distance between the rows of data. If transpose is True, calculate the distance between the columns of data.

**Performing hierarchical clustering**

To perform hierarchical clustering on the items stored in the record, use

```python
>>> tree = record.treecluster()
```

where the following arguments are defined:

- **transpose** (default: 0)
  Determines if rows (transpose is False) or columns (transpose is True) are to be clustered.

- **method** (default: 'm')
  defines the linkage method to be used:
  - `method='s'`: pairwise single-linkage clustering
  - `method='m'`: pairwise maximum- (or complete-) linkage clustering
  - `method='c'`: pairwise centroid-linkage clustering
  - `method='a'`: pairwise average-linkage clustering

- **dist** (default: 'e', Euclidean distance)
  Defines the distance function to be used (see 18.1).

- **transpose**
  Determines if genes or samples are being clustered. If transpose is False, genes (rows) are being clustered. If transpose is True, samples (columns) are clustered.

This function returns a `Tree` object. This object contains (number of items – 1) nodes, where the number of items is the number of rows if rows were clustered, or the number of columns if columns were clustered. Each node describes a pairwise linking event, where the node attributes left and right each contain the number of one item or subnode, and distance the distance between them. Items are numbered from 0 to (number of items – 1), while clusters are numbered -1 to – (number of items – 1).
Performing $k$-means or $k$-medians clustering

To perform $k$-means or $k$-medians clustering on the items stored in the record, use

```python
>>> clusterid, error, nfound = record.kcluster()
```

where the following arguments are defined:

- **nclusters** (default: 2)
  The number of clusters $k$.

- **transpose** (default: 0)
  Determines if rows (transpose is 0) or columns (transpose is 1) are to be clustered.

- **npass** (default: 1)
  The number of times the $k$-means/-medians clustering algorithm is performed, each time with a different (random) initial condition. If `initialid` is given, the value of npass is ignored and the clustering algorithm is run only once, as it behaves deterministically in that case.

- **method** (default: 'a')
  describes how the center of a cluster is found:
    - `method='a'`: arithmetic mean ($k$-means clustering);
    - `method='m'`: median ($k$-medians clustering).
  For other values of method, the arithmetic mean is used.

- **dist** (default: 'e', Euclidean distance)
  Defines the distance function to be used (see 18.1).

This function returns a tuple `(clusterid, error, nfound)`, where `clusterid` is an integer array containing the number of the cluster to which each row or cluster was assigned, `error` is the within-cluster sum of distances for the optimal clustering solution, and `nfound` is the number of times this optimal solution was found.

Calculating a Self-Organizing Map

To calculate a Self-Organizing Map of the items stored in the record, use

```python
>>> clusterid, celldata = record.somcluster()
```

where the following arguments are defined:

- **transpose** (default: 0)
  Determines if rows (transpose is 0) or columns (transpose is 1) are to be clustered.

- **nxgrid, nygrid** (default: 2, 1)
  The number of cells horizontally and vertically in the rectangular grid on which the Self-Organizing Map is calculated.

- **inittau** (default: 0.02)
  The initial value for the parameter $\tau$ that is used in the SOM algorithm. The default value for `inittau` is 0.02, which was used in Michael Eisen’s Cluster/TreeView program.

- **niter** (default: 1)
  The number of iterations to be performed.

- **dist** (default: 'e', Euclidean distance)
  Defines the distance function to be used (see 18.1).
This function returns the tuple (`clusterid, celldata`):

- **clusterid:**
  An array with two columns, where the number of rows is equal to the number of items that were clustered. Each row contains the \( x \) and \( y \) coordinates of the cell in the rectangular SOM grid to which the item was assigned.

- **celldata:**
  An array with dimensions \((nxgrid.nygrid, \text{number of columns})\) if rows are being clustered, or \((nxgrid.nygrid, \text{number of rows})\) if columns are being clustered. Each element \([ix][iy]\) of this array is a 1D vector containing the gene expression data for the centroid of the cluster in the grid cell with coordinates \([ix][iy]\).

### Saving the clustering result

To save the clustering result, use

```python
>>> record.save(jobname, geneclusters, expclusters)
```

where the following arguments are defined:

- **jobname**
  The string `jobname` is used as the base name for names of the files that are to be saved.

- **geneclusters**
  This argument describes the gene (row-wise) clustering result. In case of \( k \)-means clustering, this is a 1D array containing the number of the cluster each gene belongs to. It can be calculated using `kcluster`. In case of hierarchical clustering, `geneclusters` is a `Tree` object.

- **expclusters**
  This argument describes the (column-wise) clustering result for the experimental conditions. In case of \( k \)-means clustering, this is a 1D array containing the number of the cluster each experimental condition belongs to. It can be calculated using `kcluster`. In case of hierarchical clustering, `expclusters` is a `Tree` object.

This method writes the text file `jobname.cdt`, `jobname.gtr`, `jobname.atr`, `jobname*.kgg`, and/or `jobname*.kag` for subsequent reading by the Java TreeView program. If `geneclusters` and `expclusters` are both `None`, this method only writes the text file `jobname.cdt`; this file can subsequently be read into a new `Record` object.

### 18.8 Example calculation

This is an example of a hierarchical clustering calculation, using single linkage clustering for genes and maximum linkage clustering for experimental conditions. As the Euclidean distance is being used for gene clustering, it is necessary to scale the node distances `genetree` such that they are all between zero and one. This is needed for the Java TreeView code to display the tree diagram correctly. To cluster the experimental conditions, the uncentered correlation is being used. No scaling is needed in this case, as the distances in `exptree` are already between zero and two.

The example data `cyano.txt` can be found in Biopython’s `Tests/Cluster` subdirectory and is from the paper [20, Hihara et al., 2001].

```python
>>> from Bio import Cluster
>>> with open("cyano.txt") as handle:
...     record = Cluster.read(handle)
... ```
>>> genetree = record.treecluster(method="s")
>>> genetree.scale()
>>> exptree = record.treecluster(dist="u", transpose=1)
>>> record.save("cyano_result", genetree, exptree)

This will create the files cyano_result.cdt, cyano_result.gtr, and cyano_result.atr.

Similarly, we can save a k-means clustering solution:

>>> from Bio import Cluster
>>> with open("cyano.txt") as handle:
...     record = Cluster.read(handle)
...

>>> (geneclusters, error, ifound) = record.kcluster(nclusters=5, npass=1000)
>>> (expclusters, error, ifound) = record.kcluster(nclusters=2, npass=100, transpose=1)
>>> record.save("cyano_result", geneclusters, expclusters)

This will create the files cyano_result_K_G2_A2.cdt, cyano_result_K_G2.kgg, and cyano_result_K_A2.kag.
Chapter 19

Graphics including GenomeDiagram

The Bio.Graphics module depends on the third party Python library ReportLab. Although focused on producing PDF files, ReportLab can also create encapsulated postscript (EPS) and (SVG) files. In addition to these vector based images, provided certain further dependencies such as the Python Imaging Library (PIL) are installed, ReportLab can also output bitmap images (including JPEG, PNG, GIF, BMP and PICT formats).

19.1 GenomeDiagram

19.1.1 Introduction

The Bio.Graphics.GenomeDiagram module was added to Biopython 1.50, having previously been available as a separate Python module dependent on Biopython. GenomeDiagram is described in the Bioinformatics journal publication by Pritchard et al. (2006) [35], which includes some examples images. There is a PDF copy of the old manual here, http://biopython.org/DIST/docs/GenomeDiagram/userguide.pdf which has some more examples.

As the name might suggest, GenomeDiagram was designed for drawing whole genomes, in particular prokaryotic genomes, either as linear diagrams (optionally broken up into fragments to fit better) or as circular wheel diagrams. Have a look at Figure 2 in Toth et al. (2006) [47] for a good example. It proved also well suited to drawing quite detailed figures for smaller genomes such as phage, plasmids or mitochondria, for example see Figures 1 and 2 in Van der Auwera et al. (2009) [48] (shown with additional manual editing).

This module is easiest to use if you have your genome loaded as a SeqRecord object containing lots of SeqFeature objects - for example as loaded from a GenBank file (see Chapters 4 and 5).

19.1.2 Diagrams, tracks, feature-sets and features

GenomeDiagram uses a nested set of objects. At the top level, you have a diagram object representing a sequence (or sequence region) along the horizontal axis (or circle). A diagram can contain one or more tracks, shown stacked vertically (or radially on circular diagrams). These will typically all have the same length and represent the same sequence region. You might use one track to show the gene locations, another to show regulatory regions, and a third track to show the GC percentage.

The most commonly used type of track will contain features, bundled together in feature-sets. You might choose to use one feature-set for all your CDS features, and another for rRNA features. This isn’t required - they can all go in the same feature-set, but it makes it easier to update the properties of just selected features (e.g. make all the tRNA features red).

There are two main ways to build up a complete diagram. Firstly, the top down approach where you create a diagram object, and then using its methods add track(s), and use the track methods to add feature-
set(s), and use their methods to add the features. Secondly, you can create the individual objects separately (in whatever order suits your code), and then combine them.

19.1.3 A top down example

We’re going to draw a whole genome from a SeqRecord object read in from a GenBank file (see Chapter 5). This example uses the pPCP1 plasmid from Yersinia pestis biovar Microtus, the file is included with the Biopython unit tests under the GenBank folder, or online NC_005816.gb from our website.

```python
from reportlab.lib import colors
from reportlab.lib.units import cm
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO

record = SeqIO.read("NC_005816.gb", "genbank")

We're using a top down approach, so after loading in our sequence we next create an empty diagram, then add an (empty) track, and to that add an (empty) feature set:

```python
gd_diagram = GenomeDiagram.Diagram("Yersinia pestis biovar Microtus plasmid pPCP1")
gd_track_for_features = gd_diagram.new_track(1, name="Annotated Features")
gd_feature_set = gd_track_for_features.new_set()
```

Now the fun part - we take each gene SeqFeature object in our SeqRecord, and use it to generate a feature on the diagram. We’re going to color them blue, alternating between a dark blue and a light blue.

```python
for feature in record.features:
    if feature.type != "gene":  # Exclude this feature
        continue
    if len(gd_feature_set) % 2 == 0:
        color = colors.blue
    else:
        color = colors.lightblue
    gd_feature_set.add_feature(feature, color=color, label=True)
```

Now we come to actually making the output file. This happens in two steps, first we call the draw method, which creates all the shapes using ReportLab objects. Then we call the write method which renders these to the requested file format. Note you can output in multiple file formats:

```python
gd_diagram.draw(
    format="linear",
    orientation="landscape",
    pagesize="A4",
    fragments=4,
    start=0,
    end=len(record),
)
gd_diagram.write("plasmid_linear.pdf", "PDF")
gd_diagram.write("plasmid_linear.eps", "EPS")
gd_diagram.write("plasmid_linear.svg", "SVG")
```

Also, provided you have the dependencies installed, you can also do bitmaps, for example:
The expected output is shown in Figure 19.1. Notice that the `fragments` argument which we set to four controls how many pieces the genome gets broken up into.

If you want to do a circular figure, then try this:

```python
gd_diagram.draw(
    format="circular",
    circular=True,
    pagesize=(20 * cm, 20 * cm),
    start=0,
    end=len(record),
    circle_core=0.7,
)
```

The expected output is shown in Figure 19.2. These figures are not very exciting, but we’ve only just got started.

### 19.1.4 A bottom up example

Now let’s produce exactly the same figures, but using the bottom up approach. This means we create the different objects directly (and this can be done in almost any order) and then combine them.

```python
from reportlab.lib import colors
from reportlab.lib.units import cm
```
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO

record = SeqIO.read("NC_005816.gb", "genbank")

# Create the feature set and its feature objects,
gd_feature_set = GenomeDiagram.FeatureSet()
for feature in record.features:
    if feature.type != "gene":  # Exclude this feature
        continue
    if len(gd_feature_set) % 2 == 0:  # Exclude this feature
        color = colors.blue
    else:
        color = colors.lightblue
    gd_feature_set.add_feature(feature, color=color, label=True)
# (this for loop is the same as in the previous example)

# Create a track, and a diagram
gd_track_for_features = GenomeDiagram.Track(name="Annotated Features")
gd_diagram = GenomeDiagram.Diagram("Yersinia pestis biovar Microtus plasmid pPCP1")

# Now have to glue the bits together...
gd_track_for_features.add_set(gd_feature_set)
gd_diagram.add_track(gd_track_for_features, 1)

You can now call the draw and write methods as before to produce a linear or circular diagram, using the code at the end of the top-down example above. The figures should be identical.
19.1.5 Features without a SeqFeature

In the above example we used a `SeqRecord`'s `SeqFeature` objects to build our diagram (see also Section 4.3). Sometimes you won’t have `SeqFeature` objects, but just the coordinates for a feature you want to draw. You have to create minimal `SeqFeature` object, but this is easy:

```python
from Bio.SeqFeature import SeqFeature, SimpleLocation

my_seq_feature = SeqFeature(SimpleLocation(50, 100, strand=+1))
```

For strand, use `+1` for the forward strand, `-1` for the reverse strand, and `None` for both. Here is a short self contained example:

```python
from Bio.SeqFeature import SeqFeature, SimpleLocation
from Bio.Graphics import GenomeDiagram
from reportlab.lib.units import cm

gdd = GenomeDiagram.Diagram("Test Diagram")
gdt_features = gdd.new_track(1, greytrack=False)
gds_features = gdt_features.new_set()

# Add three features to show the strand options,
feature = SeqFeature(SimpleLocation(25, 125, strand=+1))
gds_features.add_feature(feature, name="Forward", label=True)
feature = SeqFeature(SimpleLocation(150, 250, strand=None))
gds_features.add_feature(feature, name="Strandless", label=True)
feature = SeqFeature(SimpleLocation(275, 375, strand=-1))
gds_features.add_feature(feature, name="Reverse", label=True)

gdd.draw(format="linear", pagesize=(15 * cm, 4 * cm), fragments=1, start=0, end=400)
gdd.write("GD_labels_default.pdf", "pdf")
```

The output is shown at the top of Figure 19.3 (in the default feature color, pale green).

Notice that we have used the `name` argument here to specify the caption text for these features. This is discussed in more detail next.

19.1.6 Feature captions

Recall we used the following (where `feature` was a `SeqFeature` object) to add a feature to the diagram:

```python
gd_feature_set.add_feature(feature, color=color, label=True)
```

In the example above the `SeqFeature` annotation was used to pick a sensible caption for the features. By default the following possible entries under the `SeqFeature` object’s qualifiers dictionary are used: `gene`, `label`, `name`, `locus_tag`, and `product`. More simply, you can specify a name directly:

```python
gd_feature_set.add_feature(feature, color=color, label=True, name="My Gene")
```

In addition to the caption text for each feature’s label, you can also choose the font, position (this defaults to the start of the sigil, you can also choose the middle or at the end) and orientation (for linear diagrams only, where this defaults to rotated by 45 degrees):
# Large font, parallel with the track
```
gd_feature_set.add_feature(
    feature, label=True, color="green", label_size=25, label_angle=0
)
```

# Very small font, perpendicular to the track (towards it)
```
gd_feature_set.add_feature(
    feature, 
    label=True, 
    color="purple", 
    label_position="end", 
    label_size=4, 
    label_angle=90,
)
```

# Small font, perpendicular to the track (away from it)
```
gd_feature_set.add_feature( 
    feature, 
    label=True, 
    color="blue", 
    label_position="middle", 
    label_size=6, 
    label_angle=-90, 
)
```

Combining each of these three fragments with the complete example in the previous section should give something like the tracks in Figure 19.3.

We’ve not shown it here, but you can also set `label_color` to control the label’s color (used in Section 19.1.9).

You’ll notice the default font is quite small - this makes sense because you will usually be drawing many (small) features on a page, not just a few large ones as shown here.

## 19.1.7 Feature sigils

The examples above have all just used the default sigil for the feature, a plain box, which was all that was available in the last publicly released standalone version of GenomeDiagram. Arrow sigils were included when GenomeDiagram was added to Biopython 1.50:

```
# Default uses a BOX sigil
gd_feature_set.add_feature(feature)
```

```
# You can make this explicit:
gd_feature_set.add_feature(feature, sigil="BOX")
```

```
# Or opt for an arrow:
gd_feature_set.add_feature(feature, sigil="ARROW")
```

Biopython 1.61 added three more sigils,

```
# Box with corners cut off (making it an octagon)
gd_feature_set.add_feature(feature, sigil="OCTO")
```

```
# Box with jagged edges (useful for showing breaks in contains)
```
Figure 19.3: Simple GenomeDiagram showing label options. The top plot in pale green shows the default label settings (see Section 19.1.5) while the rest show variations in the label size, position and orientation (see Section 19.1.6).
gd_feature_set.add_feature(feature, sigil="JAGGY")

# Arrow which spans the axis with strand used only for direction
gd_feature_set.add_feature(feature, sigil="BIGARROW")

These are shown in Figure 19.4. Most sigils fit into a bounding box (as given by the default BOX sigil), either above or below the axis for the forward or reverse strand, or straddling it (double the height) for strand-less features. The BIGARROW sigil is different, always straddling the axis with the direction taken from the feature’s stand.

19.1.8 Arrow sigils

We introduced the arrow sigils in the previous section. There are two additional options to adjust the shapes of the arrows, firstly the thickness of the arrow shaft, given as a proportion of the height of the bounding box:

# Full height shafts, giving pointed boxes:
gd_feature_set.add_feature(feature, sigil="ARROW", color="brown", arrowshaft_height=1.0)
# Or, thin shafts:
gd_feature_set.add_feature(feature, sigil="ARROW", color="teal", arrowshaft_height=0.2)
# Or, very thin shafts:
gd_feature_set.add_feature(feature, sigil="ARROW", color="darkgreen", arrowshaft_height=0.1)

The results are shown in Figure 19.5.

Secondly, the length of the arrow head - given as a proportion of the height of the bounding box (defaulting to 0.5, or 50%):

# Short arrow heads:
gd_feature_set.add_feature(feature, sigil="ARROW", color="blue", arrowhead_length=0.25)
# Or, longer arrow heads:
gd_feature_set.add_feature(feature, sigil="ARROW", color="orange", arrowhead_length=1)
# Or, very very long arrow heads (i.e. all head, no shaft, so triangles):
gd_feature_set.add_feature(feature, sigil="ARROW", color="red", arrowhead_length=10000)

The results are shown in Figure 19.6.

Biopython 1.61 adds a new BIGARROW sigil which always straddles the axis, pointing left for the reverse strand or right otherwise:

# A large arrow straddling the axis:
gd_feature_set.add_feature(feature, sigil="BIGARROW")

All the shaft and arrow head options shown above for the ARROW sigil can be used for the BIGARROW sigil too.

19.1.9 A nice example

Now let’s return to the pPCP1 plasmid from Yersinia pestis biovar Microtus, and the top down approach used in Section 19.1.3, but take advantage of the sigil options we’ve now discussed. This time we’ll use arrows for the genes, and overlay them with strand-less features (as plain boxes) showing the position of some restriction digest sites.
Figure 19.4: Simple GenomeDiagram showing different sigils (see Section 19.1.7)
Figure 19.5: Simple GenomeDiagram showing arrow shaft options (see Section 19.1.8)
Figure 19.6: Simple GenomeDiagram showing arrow head options (see Section 19.1.8)
from reportlab.lib import colors
from reportlab.lib.units import cm
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO
from Bio.SeqFeature import SeqFeature, SimpleLocation

record = SeqIO.read("NC_005816.gb", "genbank")

gd_diagram = GenomeDiagram.Diagram(record.id)
gd_track_for_features = gd_diagram.new_track(1, name="Annotated Features")
gd_feature_set = gd_track_for_features.new_set()

for feature in record.features:
    if feature.type != "gene":  
        # Exclude this feature
        continue
    if len(gd_feature_set) % 2 == 0:
        color = colors.blue
    else:
        color = colors.lightblue
    gd_feature_set.add_feature(
        feature, sigil="ARROW", color=color, label=True, label_size=14, label_angle=0
    )

# I want to include some strandless features, so for an example
# will use EcoRI recognition sites etc.
for site, name, color in [
    ("GAATTC", "EcoRI", colors.green),
    ("CCCGGG", "SmaI", colors.orange),
    ("AAGCTT", "HindIII", colors.red),
    ("GGATCC", "BamHI", colors.purple),
]:
    index = 0
    while True:
        index = record.seq.find(site, start=index)
        if index == -1:
            break
        feature = SeqFeature(SimpleLocation(index, index + len(site)))
        gd_feature_set.add_feature(
            feature,
            color=color,
            name=name,
            label=True,
            label_size=10,
            label_color=color,
        )
        index += len(site)

gd_diagram.draw(format="linear", pagesize="A4", fragments=4, start=0, end=len(record))
gd_diagram.write("plasmid_linear_nice.pdf", "PDF")
gd_diagram.write("plasmid_linear_nice.eps", "EPS")
The expected output is shown in Figures 19.7 and 19.8.

![Figure 19.7: Linear diagram for Yersinia pestis biovar Microtus plasmid pPCP1 showing selected restriction digest sites (see Section 19.1.9).]

**19.1.10** Multiple tracks

All the examples so far have used a single track, but you can have more than one track – for example show the genes on one, and repeat regions on another. In this example we’re going to show three phage genomes side by side to scale, inspired by Figure 6 in Proux et al. (2002) [36]. We’ll need the GenBank files for the following three phage:

- **NC_002703** – Lactococcus phage Tuc2009, complete genome (38347 bp)
Figure 19.8: Circular diagram for *Yersinia pestis biovar Microtus* plasmid pPCP1 showing selected restriction digest sites (see Section 19.1.9).
• **AF323668** – Bacteriophage bIL285, complete genome (35538 bp)

• **NC_003212** – *Listeria innocua* Clip11262, complete genome, of which we are focussing only on integrated prophage 5 (similar length).

You can download these using Entrez if you like, see Section 12.6 for more details. For the third record we’ve worked out where the phage is integrated into the genome, and slice the record to extract it (with the features preserved, see Section 4.7), and must also reverse complement to match the orientation of the first two phage (again preserving the features, see Section 4.9):

```python
from Bio import SeqIO

A_rec = SeqIO.read("NC_002703.gbk", "gb")
B_rec = SeqIO.read("AF323668.gbk", "gb")
C_rec = SeqIO.read("NC_003212.gbk", "gb")[2587879:2625807].reverse_complement(name=True)
```

The figure we are imitating used different colors for different gene functions. One way to do this is to edit the GenBank file to record color preferences for each feature - something Sanger’s Artemis editor does, and which GenomeDiagram should understand. Here however, we’ll just hard code three lists of colors.

Note that the annotation in the GenBank files doesn’t exactly match that shown in Proux *et al.*, they have drawn some unannotated genes.

```python
from reportlab.lib.colors import (red, grey, orange, green, brown, blue, lightblue, purple,
)

A_colors = (red * 5 + grey * 7 + orange * 2 + grey * 2 + orange + grey * 11 + green * 4 + grey + green * 2 + grey, green + brown * 5 + blue * 4 + lightblue * 5 + grey, lightblue + purple * 2 + grey
)
B_colors = (red * 6
```
Now to draw them – this time we add three tracks to the diagram, and also notice they are given different start/end values to reflect their different lengths (this requires Biopython 1.59 or later).

```python
from Bio.Graphics import GenomeDiagram

name = "Proux Fig 6"
gd_diagram = GenomeDiagram.Diagram(name)
max_len = 0
for record, gene_colors in zip([A_rec, B_rec, C_rec], [A_colors, B_colors, C_colors]):
    max_len = max(max_len, len(record))
    gd_track_for_features = gd_diagram.new_track(1, name=record.name, greytrack=True, start=0, end=len(record))
    gd_feature_set = gd_track_for_features.new_set()
    i = 0
    for feature in record.features:
        if feature.type != "gene":  # Exclude this feature
            continue
        gd_feature_set.add_feature(feature,
                                    sigil="ARROW",
                                    color=gene_colors[i],
                                    label=True,
                                    name=str(i + 1),
                                    label_position="start",
                                    label_size=6,
                                    label_angle=0,
                                    textcolor=feature.type)
GD_diagram.draw(format="linear", pagesize="A4", fragments=1, start=0, end=max_len)
GD_diagram.write(name + ".pdf", "PDF")
GD_diagram.write(name + ".eps", "EPS")
GD_diagram.write(name + ".svg", "SVG")

The expected output is shown in Figure 19.9. I did wonder why in the original manuscript there were no

![Linear diagram with three tracks for Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from Listeria innocua Clip11262 (NC_003212) (see Section 19.1.10).](image)

red or orange genes marked in the bottom phage. Another important point is here the phage are shown with different lengths - this is because they are all drawn to the same scale (they are different lengths).

The key difference from the published figure is they have color-coded links between similar proteins – which is what we will do in the next section.

### 19.1.11 Cross-Links between tracks

Biopython 1.59 added the ability to draw cross links between tracks - both simple linear diagrams as we will show here, but also linear diagrams split into fragments and circular diagrams.
Continuing the example from the previous section inspired by Figure 6 from Proux et al. 2002 [36], we would need a list of cross links between pairs of genes, along with a score or color to use. Realistically you might extract this from a BLAST file computationally, but here I have manually typed them in.

My naming convention continues to refer to the three phage as A, B and C. Here are the links we want to show between A and B, given as a list of tuples (percentage similarity score, gene in A, gene in B).

# Tuc2009 (NC_002703) vs bIL285 (AF323668)

\[
A_{\text{vs}}B = [(99, "Tuc2009_01", "int"),
(33, "Tuc2009_03", "orf4"),
(94, "Tuc2009_05", "orf6"),
(100, "Tuc2009_06", "orf7"),
(97, "Tuc2009_07", "orf8"),
(98, "Tuc2009_08", "orf9"),
(98, "Tuc2009_09", "orf10"),
(100, "Tuc2009_10", "orf12"),
(100, "Tuc2009_11", "orf13"),
(94, "Tuc2009_12", "orf14"),
(87, "Tuc2009_13", "orf15"),
(94, "Tuc2009_14", "orf16"),
(94, "Tuc2009_15", "orf17"),
(88, "Tuc2009_17", "rusA"),
(91, "Tuc2009_18", "orf20"),
(93, "Tuc2009_19", "orf22"),
(71, "Tuc2009_20", "orf23"),
(51, "Tuc2009_22", "orf27"),
(97, "Tuc2009_23", "orf28"),
(88, "Tuc2009_24", "orf29"),
(26, "Tuc2009_26", "orf38"),
(19, "Tuc2009_46", "orf52"),
(77, "Tuc2009_48", "orf54"),
(91, "Tuc2009_49", "orf55"),
(95, "Tuc2009_52", "orf60"),
]

Likewise for B and C:

# bIL285 (AF323668) vs Listeria innocua prophage 5 (in NC_003212)

\[
B_{\text{vs}}C = [(42, "orf39", "lin2581"),
(31, "orf40", "lin2580"),
(49, "orf41", "lin2579"),
(54, "orf42", "lin2578"),
(55, "orf43", "lin2577"),
(33, "orf44", "lin2576"),
(51, "orf46", "lin2575"),
(33, "orf47", "lin2574"),
(40, "orf48", "lin2573"),
(25, "orf49", "lin2572"),
(50, "orf50", "lin2571"),
(48, "orf51", "lin2570"),
(24, "orf52", "lin2568"),
]
For the first and last phage these identifiers are locus tags, for the middle phage there are no locus tags so I’ve used gene names instead. The following little helper function lets us lookup a feature using either a locus tag or gene name:

```python
def get_feature(features, id, tags=["locus_tag", "gene"]):
    """Search list of SeqFeature objects for an identifier under the given tags."""
    for f in features:
        for key in tags:
            # tag may not be present in this feature
            for x in f.qualifiers.get(key, []):
                if x == id:
                    return f
    raise KeyError(id)
```

We can now turn those list of identifier pairs into SeqFeature pairs, and thus find their location coordinates. We can now add all that code and the following snippet to the previous example (just before the `gd_diagram.draw(...)` line – see the finished example script `Proux_et_al_2002_Figure_6.py` included in the `Doc/examples` folder of the Biopython source code) to add cross links to the figure:

```python
from Bio.Graphics.GenomeDiagram import CrossLink
from reportlab.lib import colors

# Note it might have been clearer to assign the track numbers explicitly...
for rec_X, tn_X, rec_Y, tn_Y, X_vs_Y in [(A_rec, 3, B_rec, 2, A_vs_B), (B_rec, 2, C_rec, 1, B_vs_C), ]:
    track_X = gd_diagram.tracks[tn_X]
    track_Y = gd_diagram.tracks[tn_Y]
    for score, id_X, id_Y in X_vs_Y:
        feature_X = get_feature(rec_X.features, id_X)
        feature_Y = get_feature(rec_Y.features, id_Y)
        color = colors.linearlyInterpolatedColor(
            colors.white, colors.firebrick, 0, 100, score
        )
        link_xy = CrossLink(
            (track_X, feature_X.location.start, feature_X.location.end),
            (track_Y, feature_Y.location.start, feature_Y.location.end),
            color,
            colors.lightgrey,
        )
        gd_diagram.cross_track_links.append(link_xy)
```

There are several important pieces to this code. First the `GenomeDiagram` object has a `cross_track_links` attribute which is just a list of `CrossLink` objects. Each `CrossLink` object takes two sets of track-specific coordinates (here given as tuples, you can alternatively use a `GenomeDiagram.Feature` object instead). You can optionally supply a color, border color, and say if this link should be drawn flipped (useful for showing inversions).
You can also see how we turn the BLAST percentage identity score into a color, interpolating between white (0%) and a dark red (100%). In this example we don’t have any problems with overlapping cross-links. One way to tackle that is to use transparency in ReportLab, by using colors with their alpha channel set. However, this kind of shaded color scheme combined with overlap transparency would be difficult to interpret. The expected output is shown in Figure 19.10.

![Figure 19.10: Linear diagram with three tracks for Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from Listeria innocua Clip11262 (NC_003212) plus basic cross-links shaded by percentage identity (see Section 19.1.11).](image)

There is still a lot more that can be done within Biopython to help improve this figure. First of all, the cross links in this case are between proteins which are drawn in a strand specific manor. It can help to add a background region (a feature using the ‘BOX’ sigil) on the feature track to extend the cross link. Also, we could reduce the vertical height of the feature tracks to allocate more to the links instead – one way to do that is to allocate space for empty tracks. Furthermore, in cases like this where there are no large gene overlaps, we can use the axis-straddling BIGARROW sigil, which allows us to further reduce the vertical space needed for the track. These improvements are demonstrated in the example script Proux_et_al_2002_Figure_6.py included in the Doc/examples folder of the Biopython source code. The expected output is shown in Figure 19.11.

Beyond that, finishing touches you might want to do manually in a vector image editor include fine tuning the placement of gene labels, and adding other custom annotation such as highlighting particular regions.

Although not really necessary in this example since none of the cross-links overlap, using a transparent color in ReportLab is a very useful technique for superimposing multiple links. However, in this case a
Figure 19.11: Linear diagram with three tracks for Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from Listeria innocua Clip11262 (NC_003212) plus cross-links shaded by percentage identity (see Section 19.1.11).
shaded color scheme should be avoided.

19.1.12 Further options

You can control the tick marks to show the scale – after all every graph should show its units, and the number of the grey-track labels.

Also, we have only used the FeatureSet so far. GenomeDiagram also has a GraphSet which can be used for show line graphs, bar charts and heat plots (e.g. to show plots of GC% on a track parallel to the features).

These options are not covered here yet, so for now we refer you to the User Guide (PDF) included with the standalone version of GenomeDiagram (but please read the next section first), and the docstrings.

19.1.13 Converting old code

If you have old code written using the standalone version of GenomeDiagram, and you want to switch it over to using the new version included with Biopython then you will have to make a few changes - most importantly to your import statements.

Also, the older version of GenomeDiagram used only the UK spellings of color and center (colour and centre). You will need to change to the American spellings, although for several years the Biopython version of GenomeDiagram supported both.

For example, if you used to have:

```python
from GenomeDiagram import GDFeatureSet, GDDiagram

gdd = GDDiagram("An example")
...
```

you could just switch the import statements like this:

```python
from Bio.Graphics.GenomeDiagram import FeatureSet as GDFeatureSet, Diagram as GDDiagram

gdd = GDDiagram("An example")
...
```

and hopefully that should be enough. In the long term you might want to switch to the new names, but you would have to change more of your code:

```python
from Bio.Graphics.GenomeDiagram import FeatureSet, Diagram

gdd = Diagram("An example")
...
```

or:

```python
from Bio.Graphics import GenomeDiagram

gdd = GenomeDiagram.Diagram("An example")
...
```

If you run into difficulties, please ask on the Biopython mailing list for advice. One catch is that we have not included the old module GenomeDiagram.GDUtilities yet. This included a number of GC% related functions, which will probably be merged under Bio.SeqUtils later on.
19.2 Chromosomes

The Bio.Graphics.BasicChromosome module allows drawing of chromosomes. There is an example in Jupe et al. (2012) [22] (open access) using colors to highlight different gene families.

19.2.1 Simple Chromosomes

Here is a very simple example - for which we’ll use *Arabidopsis thaliana*.

You can skip this bit, but first I downloaded the five sequenced chromosomes as five individual FASTA files from the NCBI’s FTP site ftp://ftp.ncbi.nlm.nih.gov/genomes/archive/old_refseq/Arabidopsis_thaliana/ and then parsed them with Bio.SeqIO to find out their lengths. You could use the GenBank files for this (and the next example uses those for plotting features), but if all you want is the length it is faster to use the FASTA files for the whole chromosomes:

```python
from Bio import SeqIO

entries = [
    ("Chr I", "CHR_I/NC_003070.fna"),
    ("Chr II", "CHR_II/NC_003071.fna"),
    ("Chr III", "CHR_III/NC_003074.fna"),
    ("Chr IV", "CHR_IV/NC_003075.fna"),
    ("Chr V", "CHR_V/NC_003076.fna"),
]

for name, filename in entries:
    record = SeqIO.read(filename, "fasta")
    print(name, len(record))
```

This gave the lengths of the five chromosomes, which we’ll now use in the following short demonstration of the BasicChromosome module:

```python
from reportlab.lib.units import cm
from Bio.Graphics import BasicChromosome

entries = [
    ("Chr I", 30432563),
    ("Chr II", 19705359),
    ("Chr III", 23470805),
    ("Chr IV", 18585042),
    ("Chr V", 26992728),
]

max_len = 30432563  # Could compute this from the entries dict
telomere_length = 1000000  # For illustration

chr_diagram = BasicChromosome.Organism()
chr_diagram.page_size = (29.7 * cm, 21 * cm)  # A4 landscape

for name, length in entries:
    cur_chromosome = BasicChromosome.Chromosome(name)
    # Set the scale to the MAXIMUM length plus the two telomeres in bp,
    # want the same scale used on all five chromosomes so they can be
    # compared to each other
    cur_chromosome.scale_num = max_len + 2 * telomere_length
```

394
Figure 19.12: Simple chromosome diagram for \textit{Arabidopsis thaliana}.

Figure 19.13: Chromosome diagram for \textit{Arabidopsis thaliana} showing tRNA genes.
# Add an opening telomere
start = BasicChromosome.TelomereSegment()
start.scale = telomere_length
cur_chromosome.add(start)

# Add a body - using bp as the scale length here.
body = BasicChromosome.ChromosomeSegment()
body.scale = length
cur_chromosome.add(body)

# Add a closing telomere
end = BasicChromosome.TelomereSegment(inverted=True)
end.scale = telomere_length
cur_chromosome.add(end)

# This chromosome is done
chr_diagram.add(cur_chromosome)

chr_diagram.draw("simple_chrom.pdf", "Arabidopsis thaliana")

This should create a very simple PDF file, shown in Figure 19.12. This example is deliberately short and sweet. The next example shows the location of features of interest.

19.2.2 Annotated Chromosomes

Continuing from the previous example, let’s also show the tRNA genes. We’ll get their locations by parsing the GenBank files for the five Arabidopsis thaliana chromosomes. You’ll need to download these files from the NCBI FTP site ftp://ftp.ncbi.nlm.nih.gov/genomes/archive/old_refseq/Arabidopsis_thaliana/, and preserve the subdirectory names or edit the paths below:

```python
from reportlab.lib.units import cm
from Bio import SeqIO
from Bio.Graphics import BasicChromosome

entries = [
    ("Chr I", "CHR_I/NC_003070.gbk"),
    ("Chr II", "CHR_II/NC_003071.gbk"),
    ("Chr III", "CHR_III/NC_003074.gbk"),
    ("Chr IV", "CHR_IV/NC_003075.gbk"),
    ("Chr V", "CHR_V/NC_003076.gbk"),
]

max_len = 30432563 # Could compute this from the entries dict
telomere_length = 1000000 # For illustration

chr_diagram = BasicChromosome.Organism()
chr_diagram.page_size = (29.7 * cm, 21 * cm) # A4 landscape

for index, (name, filename) in enumerate(entries):
    record = SeqIO.read(filename, "genbank")
    length = len(record)
```
features = [f for f in record.features if f.type == "tRNA"]
# Record an Artemis style integer color in the feature's qualifiers,
# 1 = Black, 2 = Red, 3 = Green, 4 = blue, 5 = cyan, 6 = purple
for f in features:
    f.qualifiers["color"] = [index + 2]

cur_chromosome = BasicChromosome.Chromosome(name)
# Set the scale to the MAXIMUM length plus the two telomeres in bp,
# want the same scale used on all five chromosomes so they can be
# compared to each other
cur_chromosome.scale_num = max_len + 2 * telomere_length

# Add an opening telomere
start = BasicChromosome.TelomereSegment()
start.scale = telomere_length
cur_chromosome.add(start)

# Add a body - again using bp as the scale length here.
body = BasicChromosome.AnnotatedChromosomeSegment(length, features)
body.scale = length
cur_chromosome.add(body)

# Add a closing telomere
end = BasicChromosome.TelomereSegment(inverted=True)
end.scale = telomere_length
cur_chromosome.add(end)

# This chromosome is done
chr_diagram.add(cur_chromosome)

chr_diagram.draw("tRNA_chrom.pdf", "Arabidopsis thaliana")

It might warn you about the labels being too close together - have a look at the forward strand (right
hand side) of Chr I, but it should create a colorful PDF file, shown in Figure 19.12.
Chapter 20

KEGG

KEGG (https://www.kegg.jp/) is a database resource for understanding high-level functions and utilities of the biological system, such as the cell, the organism and the ecosystem, from molecular-level information, especially large-scale molecular datasets generated by genome sequencing and other high-throughput experimental technologies.

Please note that the KEGG parser implementation in Biopython is incomplete. While the KEGG website indicates many flat file formats, only parsers and writers for compound, enzyme, and map are currently implemented. However, a generic parser is implemented to handle the other formats.

20.1 Parsing KEGG records

Parsing a KEGG record is as simple as using any other file format parser in Biopython. (Before running the following codes, please open http://rest.kegg.jp/get/ec:5.4.2.2 with your web browser and save it as ec_5.4.2.2.txt.)

```python
>>> from Bio.KEGG import Enzyme
>>> records = Enzyme.parse(open("ec_5.4.2.2.txt"))
>>> record = list(records)[0]
>>> record.classname
['Isomerases;', 'Intramolecular transferases;', 'Phosphotransferases (phosphomutases)']
>>> record.entry
'5.4.2.2'
```

Alternatively, if the input KEGG file has exactly one entry, you can use read:

```python
>>> from Bio.KEGG import Enzyme
>>> record = Enzyme.read(open("ec_5.4.2.2.txt"))
>>> record.classname
['Isomerases;', 'Intramolecular transferases;', 'Phosphotransferases (phosphomutases)']
>>> record.entry
'5.4.2.2'
```

The following section will shows how to download the above enzyme using the KEGG api as well as how to use the generic parser with data that does not have a custom parser implemented.

20.2 Querying the KEGG API

Biopython has full support for the querying of the KEGG api. Querying all KEGG endpoints are supported; all methods documented by KEGG (https://www.kegg.jp/kegg/rest/keggapi.html) are supported.
interface has some validation of queries which follow rules defined on the KEGG site. However, invalid queries which return a 400 or 404 must be handled by the user.

First, here is how to extend the above example by downloading the relevant enzyme and passing it through the Enzyme parser.

```python
>>> from Bio.KEGG import REST
>>> from Bio.KEGG import Enzyme
>>> request = REST.kegg_get("ec:5.4.2.2")
>>> open("ec_5.4.2.2.txt", "w").write(request.read())
>>> records = Enzyme.parse(open("ec_5.4.2.2.txt"))
>>> record = list(records)[0]
>>> record.classname
['Isomerases;', 'Intramolecular transferases;', 'Phosphotransferases (phosphomutases)']
>>> record.entry
'5.4.2.2'
```

Now, here’s a more realistic example which shows a combination of querying the KEGG API. This will demonstrate how to extract a unique set of all human pathway gene symbols which relate to DNA repair. The steps that need to be taken to do so are as follows. First, we need to get a list of all human pathways. Secondly, we need to filter those for ones which relate to “repair”. Lastly, we need to get a list of all the gene symbols in all repair pathways.

```python
from Bio.KEGG import REST

human_pathways = REST.kegg_list("pathway", "hsa").read()

# Filter all human pathways for repair pathways
repair_pathways = []
for line in human_pathways.rstrip().split("\n"):
    entry, description = line.split("\t")
    if "repair" in description:
        repair_pathways.append(entry)

# Get the genes for pathways and add them to a list
repair_genes = []
for pathway in repair_pathways:
    pathway_file = REST.kegg_get(pathway).read()  # query and read each pathway

    # iterate through each KEGG pathway file, keeping track of which section
    # of the file we're in, only read the gene in each pathway
    current_section = None
    for line in pathway_file.rstrip().split("\n"):
        section = line[:12].strip()  # section names are within 12 columns
        if not section == "":
            current_section = section

        if current_section == "GENE":
            gene_identifiers, gene_description = line[12:].split("; ")
            gene_id, gene_symbol = gene_identifiers.split()

            if not gene_symbol in repair_genes:
                repair_genes.append(gene_symbol)
```

399
print(
    "There are %d repair pathways and %d repair genes. The genes are:" % (len(repair_pathways), len(repair_genes))
)
print("", ",\n join(repair_genes))

The KEGG API wrapper is compatible with all endpoints. Usage is essentially replacing all slashes in
the url with commas and using that list as arguments to the corresponding method in the KEGG module.
Here are a few examples from the api documentation (https://www.kegg.jp/kegg/docs/keggapi.html).

/list/hsa:10458+ece:Z5100    -> REST.kegg_list(["hsa:10458", "ece:Z5100"])  
/find/compound/300-310/mol_weight  -> REST.kegg_find("compound", "300-310", "mol_weight")  
/get/hsa:10458+ece:Z5100/aaseq    -> REST.kegg_get(["hsa:10458", "ece:Z5100"], "aaseq")
Chapter 21

Bio.phenotype: analyze phenotypic data

This chapter gives an overview of the functionalities of the Bio.phenotype package included in Biopython. The scope of this package is the analysis of phenotypic data, which means parsing and analyzing growth measurements of cell cultures. In its current state the package is focused on the analysis of high-throughput phenotypic experiments produced by the Phenotype Microarray technology, but future developments may include other platforms and formats.

21.1 Phenotype Microarrays

The Phenotype Microarray is a technology that measures the metabolism of bacterial and eukaryotic cells on roughly 2000 chemicals, divided in twenty 96-well plates. The technology measures the reduction of a tetrazolium dye by NADH, whose production by the cell is used as a proxy for cell metabolism; color development due to the reduction of this dye is typically measured once every 15 minutes. When cells are grown in a media that sustains cell metabolism, the recorded phenotypic data resembles a sigmoid growth curve, from which a series of growth parameters can be retrieved.

21.1.1 Parsing Phenotype Microarray data

The Bio.phenotype package can parse two different formats of Phenotype Microarray data: the CSV (comma separated values) files produced by the machine’s proprietary software and JSON files produced by analysis software, like opm or DuctApe. The parser will return one or a generator of PlateRecord objects, depending on whether the read or parse method is being used. You can test the parse function by using the Plates.csv file provided with the Biopython source code.

```python
>>> from Bio import phenotype
>>> for record in phenotype.parse("Plates.csv", "pm-csv"):
...     print("%s %i" % (record.id, len(record)))
...PM01 96
PM01 96
PM09 96
PM09 96
```

The parser returns a series of PlateRecord objects, each one containing a series of WellRecord objects (holding each well’s experimental data) arranged in 8 rows and 12 columns; each row is indicated by a
uppercase character from A to H, while columns are indicated by a two digit number, from 01 to 12. There are several ways to access WellRecord objects from a PlateRecord objects:

**Well identifier** If you know the well identifier (row + column identifiers) you can access the desired well directly.

```python
>>> record["A02"]
 WellRecord(' (0.0, 12.0), (0.25, 18.0), (0.5, 27.0), (0.75, 35.0), (1.0, 37.0), ..., (71.75, 143.0)')
```

**Well plate coordinates** The same well can be retrieved by using the row and columns numbers (0-based index).

```python
>>> from Bio import phenotype
>>> record = list(phenotype.parse("Plates.csv", "pm-csv"))[-1]
>>> print(record[0, 1].id)
A02
```

**Row or column coordinates** A series of WellRecord objects contiguous to each other in the plate can be retrieved in bulk by using the python list slicing syntax on PlateRecord objects; rows and columns are numbered with a 0-based index.

```python
>>> print(record[0])
Plate ID: PM09
Well: 12
Rows: 1
Columns: 12
PlateRecord('WellRecord['A01'], WellRecord['A02'], WellRecord['A03'], ..., WellRecord['A12']')

>>> print(record[:, 0])
Plate ID: PM09
Well: 8
Rows: 8
Columns: 1
PlateRecord('WellRecord['A01'], WellRecord['B01'], WellRecord['C01'], ..., WellRecord['H01']')

>>> print(record[:3, :3])
Plate ID: PM09
Well: 9
Rows: 3
Columns: 3
PlateRecord('WellRecord['A01'], WellRecord['A02'], WellRecord['A03'], ..., WellRecord['C03']')
```

### 21.1.2 Manipulating Phenotype Microarray data

#### 21.1.2.1 Accessing raw data

The raw data extracted from the PM files is comprised of a series of tuples for each well, containing the time (in hours) and the colorimetric measure (in arbitrary units). Usually the instrument collects data every fifteen minutes, but that can vary between experiments. The raw data can be accessed by iterating on a WellRecord object; in the example below only the first ten time points are shown.

```python
>>> from Bio import phenotype
>>> record = list(phenotype.parse("Plates.csv", "pm-csv"))[-1]
>>> well = record["A02"]
```
>>> for time, signal in well:
...    print(time, signal)
...
(0.0, 12.0)
(0.25, 18.0)
(0.5, 27.0)
(0.75, 35.0)
(1.0, 37.0)
(1.25, 41.0)
(1.5, 44.0)
(1.75, 44.0)
(2.0, 44.0)
(2.25, 44.0)
[...]

This method, while providing a way to access the raw data, doesn’t allow a direct comparison between different WellRecord objects, which may have measurements at different time points.

21.1.2.2 Accessing interpolated data

To make it easier to compare different experiments and in general to allow a more intuitive handling of the phenotypic data, the module allows to define a custom slicing of the time points that are present in the WellRecord object. Colorimetric data for time points that have not been directly measured are derived through a linear interpolation of the available data, otherwise a NaN is returned. This method only works in the time interval where actual data is available. Time intervals can be defined with the same syntax as list indexing; the default time interval is therefore one hour.

>>> well[:10]
[12.0, 37.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0]

Different time intervals can be used, for instance five minutes:

>>> well[63:64:0.083]
[12.0, 37.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0]
>>> well[9.55]
44.0
>>> well[63.33:73.33]
[113.31999999999999, 117.0, 120.31999999999999, 128.0, 129.63999999999999, 132.95999999999998, 136.95999999999998, 140.0, 142.0, nan]

21.1.2.3 Control well subtraction

Many Phenotype Microarray plates contain a control well (usually A01), that is a well where the media shouldn’t support any growth; the low signal produced by this well can be subtracted from the other wells. The PlateRecord objects have a dedicated function for that, which returns another PlateRecord object with the corrected data.
Those wells where metabolic activity is observed show a sigmoid behavior for the colorimetric data. To allow an easier way to compare different experiments a sigmoid curve can be fitted onto the data, so that a series of summary parameters can be extracted and used for comparisons. The parameters that can be extracted from the curve are:

- Minimum (\( \text{min} \)) and maximum (\( \text{max} \)) signal;
- Average height (\( \text{average height} \));
- Area under the curve (\( \text{area} \));
- Curve plateau point (\( \text{plateau} \));
- Curve slope during exponential metabolic activity (\( \text{slope} \));
- Curve lag time (\( \text{lag} \)).

All the parameters (except \( \text{min} \), \( \text{max} \) and \( \text{average height} \)) require the scipy library to be installed.

The fit function uses three sigmoid functions:

- **Gompertz** \( Ae^{-e^{(\frac{\mu_m}{\lambda})(\lambda-t)+1}} + y_0 \)
- **Logistic** \( \frac{A}{1+e^{\frac{\mu_m}{\lambda}(\lambda-t)+2}} + y_0 \)
- **Richards** \( A(1 + ve^{1+v} + e^{\frac{\mu_m}{\lambda}(1+v)(1+\frac{1}{v})(\lambda-t)} - 1) + y_0 \)

Where:

- \( A \) corresponds to the **plateau**
- \( \mu_m \) corresponds to the **slope**
- \( \lambda \) corresponds to the **lag**

These functions have been derived from this publication. The fit method by default tries first to fit the gompertz function: if it fails it will then try to fit the logistic and then the richards function. The user can also specify one of the three functions to be applied.
average_height  61.58
lag         48.60
max         143.00
min         12.00
plateau     120.02
slope       4.99

21.1.3 Writing Phenotype Microarray data

PlateRecord objects can be written to file in the form of JSON files, a format compatible with other software packages such as opm or DuctApe.

```python
>>> phenotype.write(record, "out.json", "pm-json")
1
```
Chapter 22

Cookbook – Cool things to do with it

Biopython now has two collections of “cookbook” examples – this chapter (which has been included in this tutorial for many years and has gradually grown), and http://biopython.org/wiki/Category:Cookbook which is a user contributed collection on our wiki.

We’re trying to encourage Biopython users to contribute their own examples to the wiki. In addition to helping the community, one direct benefit of sharing an example like this is that you could also get some feedback on the code from other Biopython users and developers - which could help you improve all your Python code.

In the long term, we may end up moving all of the examples in this chapter to the wiki, or elsewhere within the tutorial.

22.1 Working with sequence files

This section shows some more examples of sequence input/output, using the Bio.SeqIO module described in Chapter 5.

22.1.1 Filtering a sequence file

Often you’ll have a large file with many sequences in it (e.g. FASTA file or genes, or a FASTQ or SFF file of reads), a separate shorter list of the IDs for a subset of sequences of interest, and want to make a new sequence file for this subset.

Let’s say the list of IDs is in a simple text file, as the first word on each line. This could be a tabular file where the first column is the ID. Try something like this:

    from Bio import SeqIO
    input_file = "big_file.sff"
    id_file = "short_list.txt"
    output_file = "short_list.sff"

    with open(id_file) as id_handle:
        wanted = set(line.rstrip("\n").split(None, 1)[0] for line in id_handle)
        print("Found %i unique identifiers in %s" % (len(wanted), id_file))

        records = (r for r in SeqIO.parse(input_file, "sff") if r.id in wanted)
        count = SeqIO.write(records, output_file, "sff")
        print("Saved %i records from %s to %s" % (count, input_file, output_file))
if count < len(wanted):
    print("Warning %i IDs not found in %s" % (len(wanted) - count, input_file))

Note that we use a Python set rather than a list, this makes testing membership faster.

As discussed in Section 5.6, for a large FASTA or FASTQ file for speed you would be better off not using
the high-level SeqIO interface, but working directly with strings. This next example shows how to do this
with FASTQ files – it is more complicated:

from Bio.SeqIO.QualityIO import FastqGeneralIterator

input_file = "big_file.fastq"
id_file = "short_list.txt"
output_file = "short_list.fastq"

with open(id_file) as id_handle:
    # Taking first word on each line as an identifier
    wanted = set(line.rstrip("\n").split(None, 1)[0] for line in id_handle)
    print("Found %i unique identifiers in %s" % (len(wanted), id_file))

with open(input_file) as in_handle:
    with open(output_file, "w") as out_handle:
        for title, seq, qual in FastqGeneralIterator(in_handle):
            # The ID is the first word in the title line (after the @ sign):
            if title.split(None, 1)[0] in wanted:
                # This produces a standard 4-line FASTQ entry:
                out_handle.write("@%s\n%s\n+\n%s\n" % (title, seq, qual))
                count += 1
    print("Saved %i records from %s to %s" % (count, input_file, output_file))

if count < len(wanted):
    print("Warning %i IDs not found in %s" % (len(wanted) - count, input_file))

22.1.2 Producing randomized genomes

Let’s suppose you are looking at genome sequence, hunting for some sequence feature – maybe extreme local
GC% bias, or possible restriction digest sites. Once you’ve got your Python code working on the real genome
it may be sensible to try running the same search on randomized versions of the same genome for statistical
analysis (after all, any “features” you’ve found could just be there just by chance).

For this discussion, we’ll use the GenBank file for the pPCP1 plasmid from Yersinia pestis biovar Microtus.
The file is included with the Biopython unit tests under the GenBank folder, or you can get it from our
website, NC_005816.gb. This file contains one and only one record, so we can read it in as a SeqRecord
using the Bio.SeqIO.read() function:

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

So, how can we generate a shuffled versions of the original sequence? I would use the built in Python
random module for this, in particular the function random.shuffle – but this works on a Python list. Our
sequence is a Seq object, so in order to shuffle it we need to turn it into a list:

```python
>>> import random
>>> nuc_list = list(original_rec.seq)
>>> random.shuffle(nuc_list) # acts in situ
```
Now, in order to use Bio.SeqIO to output the shuffled sequence, we need to construct a new SeqRecord with a new Seq object using this shuffled list. In order to do this, we need to turn the list of nucleotides (single letter strings) into a long string – the standard Python way to do this is with the string object’s join method.

```python
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> shuffled_rec = SeqRecord(
...   Seq("".join(nuc_list)), id="Shuffled", description="Based on %s" % original_rec.id ...
... )
```

Let’s put all these pieces together to make a complete Python script which generates a single FASTA file containing 30 randomly shuffled versions of the original sequence.

This first version just uses a big for loop and writes out the records one by one (using the SeqRecord’s format method described in Section 5.5.4):

```python
import random
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO

original_rec = SeqIO.read("NC_005816.gb", "genbank")

with open("shuffled.fasta", "w") as output_handle:
    for i in range(30):
        nuc_list = list(original_rec.seq)
        random.shuffle(nuc_list)
        shuffled_rec = SeqRecord(
            Seq("".join(nuc_list)),
            id="Shuffled%i" % (i + 1),
            description="Based on %s" % original_rec.id,
        )
        output_handle.write(shuffled_rec.format("fasta"))
```

Personally I prefer the following version using a function to shuffle the record and a generator expression instead of the for loop:

```python
import random
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO

def make_shuffle_record(record, new_id):
    nuc_list = list(record.seq)
    random.shuffle(nuc_list)
    return SeqRecord(
        Seq("".join(nuc_list)),
        id=new_id,
        description="Based on %s" % original_rec.id,
    )
```
original_rec = SeqIO.read("NC_005816.gb", "genbank")
shuffled_recs = (
    make_shuffle_record(original_rec, "Shuffled%i" % (i + 1)) for i in range(30)
)
SeqIO.write(shuffled_recs, "shuffled.fasta", "fasta")

22.1.3 Translating a FASTA file of CDS entries

Suppose you’ve got an input file of CDS entries for some organism, and you want to generate a new FASTA file containing their protein sequences. i.e. Take each nucleotide sequence from the original file, and translate it. Back in Section 3.8 we saw how to use the Seq object’s translate method, and the optional cds argument which enables correct translation of alternative start codons.

We can combine this with Bio.SeqIO as shown in the reverse complement example in Section 5.5.3. The key point is that for each nucleotide SeqRecord, we need to create a protein SeqRecord - and take care of naming it.

You can write you own function to do this, choosing suitable protein identifiers for your sequences, and the appropriate genetic code. In this example we just use the default table and add a prefix to the identifier:

```python
from Bio.SeqRecord import SeqRecord
def make_protein_record(nuc_record):
    """Returns a new SeqRecord with the translated sequence (default table).""
    return SeqRecord(
        seq=nuc_record.seq.translate(cds=True),
        id="trans_" + nuc_record.id,
        description="translation of CDS, using default table",
    )
```

We can then use this function to turn the input nucleotide records into protein records ready for output. An elegant way and memory efficient way to do this is with a generator expression:

```python
from Bio import SeqIO
proteins = (
    make_protein_record(nuc_rec)
    for nuc_rec in SeqIO.parse("coding_sequences.fasta", "fasta")
)
SeqIO.write(proteins, "translations.fasta", "fasta")
```

This should work on any FASTA file of complete coding sequences. If you are working on partial coding sequences, you may prefer to use nuc_record.seq.translate(to_stop=True) in the example above, as this wouldn’t check for a valid start codon etc.

22.1.4 Making the sequences in a FASTA file upper case

Often you’ll get data from collaborators as FASTA files, and sometimes the sequences can be in a mixture of upper and lower case. In some cases this is deliberate (e.g. lower case for poor quality regions), but usually it is not important. You may want to edit the file to make everything consistent (e.g. all upper case), and you can do this easily using the upper() method of the SeqRecord object (added in Biopython 1.55):

```python
from Bio import SeqIO
proteins = (
    make_protein_record(nuc_rec)
    for nuc_rec in SeqIO.parse("coding_sequences.fasta", "fasta")
)
SeqIO.write(proteins, "translations.fasta", "fasta")
```
from Bio import SeqIO

records = (rec.upper() for rec in SeqIO.parse("mixed.fas", "fasta"))

count = SeqIO.write(records, "upper.fas", "fasta")

print("Converted %i records to upper case" % count)

How does this work? The first line is just importing the Bio.SeqIO module. The second line is the interesting bit – this is a Python generator expression which gives an upper case version of each record parsed from the input file (mixed.fas). In the third line we give this generator expression to the Bio.SeqIO.write() function and it saves the new upper cases records to our output file (upper.fas).

The reason we use a generator expression (rather than a list or list comprehension) is this means only one record is kept in memory at a time. This can be really important if you are dealing with large files with millions of entries.

22.1.5 Sorting a sequence file

Suppose you wanted to sort a sequence file by length (e.g. a set of contigs from an assembly), and you are working with a file format like FASTA or FASTQ which Bio.SeqIO can read, write (and index).

If the file is small enough, you can load it all into memory at once as a list of SeqRecord objects, sort the list, and save it:

from Bio import SeqIO

records = list(SeqIO.parse("ls_orchid.fasta", "fasta"))
records.sort(key=lambda r: len(r))
SeqIO.write(records, "sorted_orchids.fasta", "fasta")

The only clever bit is specifying a comparison method for how to sort the records (here we sort them by length). If you wanted the longest records first, you could flip the comparison or use the reverse argument:

from Bio import SeqIO

records = list(SeqIO.parse("ls_orchid.fasta", "fasta"))
records.sort(key=lambda r: -len(r))
SeqIO.write(records, "sorted_orchids.fasta", "fasta")

Now that’s pretty straight forward - but what happens if you have a very large file and you can’t load it all into memory like this? For example, you might have some next-generation sequencing reads to sort by length. This can be solved using the Bio.SeqIO.index() function.

from Bio import SeqIO

# Get the lengths and ids, and sort on length
len_and_ids = sorted(
    (len(rec), rec.id) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")
)
ids = reversed([id for (length, id) in len_and_ids])
del len_and_ids # free this memory
record_index = SeqIO.index("ls_orchid.fasta", "fasta")
records = (record_index[id] for id in ids)
SeqIO.write(records, "sorted.fasta", "fasta")
First we scan through the file once using \texttt{Bio.SeqIO.parse()}, recording the record identifiers and their lengths in a list of tuples. We then sort this list to get them in length order, and discard the lengths. Using this sorted list of identifiers \texttt{Bio.SeqIO.index()} allows us to retrieve the records one by one, and we pass them to \texttt{Bio.SeqIO.write()} for output.

These examples all use \texttt{Bio.SeqIO} to parse the records into \texttt{SeqRecord} objects which are output using \texttt{Bio.SeqIO.write()}. What if you want to sort a file format which \texttt{Bio.SeqIO.write()} doesn't support, like the plain text SwissProt format? Here is an alternative solution using the \texttt{get_raw()} method added to \texttt{Bio.SeqIO.index()} in Biopython 1.54 (see Section 5.4.2.2).

```python
from Bio import SeqIO

# Get the lengths and ids, and sort on length
len_and_ids = sorted((len(rec), rec.id) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")
)
ids = reversed([id for (length, id) in len_and_ids])
del len_and_ids  # free this memory

record_index = SeqIO.index("ls_orchid.fasta", "fasta")
with open("sorted.fasta", "wb") as out_handle:
    for id in ids:
        out_handle.write(record_index.get_raw(id))
```

Note with Python 3 onwards, we have to open the file for writing in binary mode because the \texttt{get_raw()} method returns \texttt{bytes} objects.

As a bonus, because it doesn’t parse the data into \texttt{SeqRecord} objects a second time it should be faster. If you only want to use this with FASTA format, we can speed this up one step further by using the low-level FASTA parser to get the record identifiers and lengths:

```python
from Bio.SeqIO.FastaIO import SimpleFastaParser
from Bio import SeqIO

# Get the lengths and ids, and sort on length
with open("ls_orchid.fasta") as in_handle:
    len_and_ids = sorted((len(seq), title.split(None, 1)[0])
        for title, seq in SimpleFastaParser(in_handle)
    )

ids = reversed([id for (length, id) in len_and_ids])
del len_and_ids  # free this memory

record_index = SeqIO.index("ls_orchid.fasta", "fasta")
with open("sorted.fasta", "wb") as out_handle:
    for id in ids:
        out_handle.write(record_index.get_raw(id))
```

### 22.1.6 Simple quality filtering for FASTQ files

The FASTQ file format was introduced at Sanger and is now widely used for holding nucleotide sequencing reads together with their quality scores. FASTQ files (and the related QUAL files) are an excellent example of per-letter-annotation, because for each nucleotide in the sequence there is an associated quality score. Any per-letter-annotation is held in a \texttt{SeqRecord} in the \texttt{letter_annotations} dictionary as a list, tuple or string (with the same number of elements as the sequence length).
One common task is taking a large set of sequencing reads and filtering them (or cropping them) based on their quality scores. The following example is very simplistic, but should illustrate the basics of working with quality data in a `SeqRecord` object. All we are going to do here is read in a file of FASTQ data, and filter it to pick out only those records whose PHRED quality scores are all above some threshold (here 20).

For this example we’ll use some real data downloaded from the ENA sequence read archive, `ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz` (2MB) which unzips to a 19MB file `SRR020192.fastq`. This is some Roche 454 GS FLX single end data from virus infected California sea lions (see [https://www.ebi.ac.uk/ena/data/view/SRS004476](https://www.ebi.ac.uk/ena/data/view/SRS004476) for details).

First, let’s count the reads:

```python
from Bio import SeqIO

count = 0
for rec in SeqIO.parse("SRR020192.fastq", "fastq"):
    count += 1
print("%i reads" % count)
```

Now let’s do a simple filtering for a minimum PHRED quality of 20:

```python
from Bio import SeqIO
good_reads = (
    rec
    for rec in SeqIO.parse("SRR020192.fastq", "fastq")
    if min(rec.letter_annotations['phred_quality']) >= 20
)
count = SeqIO.write(good_reads, "good_quality.fastq", "fastq")
print("Saved %i reads" % count)
```

This pulled out only 14580 reads out of the 41892 present. A more sensible thing to do would be to quality trim the reads, but this is intended as an example only.

FASTQ files can contain millions of entries, so it is best to avoid loading them all into memory at once. This example uses a generator expression, which means only one `SeqRecord` is created at a time - avoiding any memory limitations.

Note that it would be faster to use the low-level `FastqGeneralIterator` parser here (see Section 5.6), but that does not turn the quality string into integer scores.

### 22.1.7 Trimming off primer sequences

For this example we’re going to pretend that `GATGACGGTGT` is a 5’ primer sequence we want to look for in some FASTQ formatted read data. As in the example above, we’ll use the `SRR020192.fastq` file downloaded from the ENA (`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz`).

By using the main `Bio.SeqIO` interface, the same approach would work with any other supported file format (e.g. FASTA files). However, for large FASTQ files it would be faster the low-level `FastqGeneralIterator` parser here (see the earlier example, and Section 5.6).

This code uses `Bio.SeqIO` with a generator expression (to avoid loading all the sequences into memory at once), and the `Seq` object’s `startswith` method to see if the read starts with the primer sequence:

```python
from Bio import SeqIO

primer_reads = (
    rec
    for rec in SeqIO.parse("SRR020192.fastq", "fastq")
    if rec.startswith("GATGACGGTGT")
)
count = SeqIO.write(primer_reads, "primer_quality.fastq", "fastq")
print("Saved %i reads" % count)
```
if rec.seq.startswith("GATGACGGTGT")
)
count = SeqIO.write(primer_reads, "with_primer.fastq", "fastq")
print("Saved %i reads" % count)

That should find 13819 reads from SRR014849.fastq and save them to a new FASTQ file, with_primer.fastq.

Now suppose that instead you wanted to make a FASTQ file containing these reads but with the primer sequence removed? That’s just a small change as we can slice the SeqRecord (see Section 4.7) to remove the first eleven letters (the length of our primer):

```
from Bio import SeqIO

trimmed_primer_reads = (
    rec[11:]
    for rec in SeqIO.parse("SRR020192.fastq", "fastq")
    if rec.seq.startswith("GATGACGGTGT")
)
count = SeqIO.write(trimmed_primer_reads, "with_primer_trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

Again, that should pull out the 13819 reads from SRR020192.fastq, but this time strip off the first ten characters, and save them to another new FASTQ file, with_primer_trimmed.fastq.

Now, suppose you want to create a new FASTQ file where these reads have their primer removed, but all the other reads are kept as they were? If we want to still use a generator expression, it is probably clearest to define our own trim function:

```
from Bio import SeqIO

def trim_primer(record, primer):
    if record.seq.startswith(primer):
        return record[len(primer):]
    else:
        return record

trimmed_reads = (
    trim_primer(record, "GATGACGGTGT")
    for record in SeqIO.parse("SRR020192.fastq", "fastq")
)
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

This takes longer, as this time the output file contains all 41892 reads. Again, we’re used a generator expression to avoid any memory problems. You could alternatively use a generator function rather than a generator expression.

```
from Bio import SeqIO

def trim_primers(records, primer):
    """Removes perfect primer sequences at start of reads."""
```
This is a generator function, the records argument should be a list or iterator returning SeqRecord objects.

```python
len_primer = len(primer)  # cache this for later
for record in records:
    if record.seq.startswith(primer):
        yield record[len_primer:]
    else:
        yield record
```

```python
original_reads = SeqIO.parse("SRR020192.fastq", "fastq")
trimmed_reads = trim_primers(original_reads, "GATGACGGTGT")
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

This form is more flexible if you want to do something more complicated where only some of the records are retained – as shown in the next example.

### 22.1.8 Trimming off adaptor sequences

This is essentially a simple extension to the previous example. We are going to pretend `GATGACGGTGT` is an adaptor sequence in some FASTQ formatted read data, again the SRR020192.fastq file from the NCBI (ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz).

This time however, we will look for the sequence *anywhere* in the reads, not just at the very beginning:

```python
from Bio import SeqIO
def trim_adaptors(records, adaptor):
    """Trims perfect adaptor sequences.
    This is a generator function, the records argument should be a list or iterator returning SeqRecord objects.
    """
    len_adaptor = len(adaptor)  # cache this for later
    for record in records:
        index = record.seq.find(adaptor)
        if index == -1:
            # adaptor not found, so won't trim
            yield record
        else:
            # trim off the adaptor
            yield record[index + len_adaptor :]
```

```python
original_reads = SeqIO.parse("SRR020192.fastq", "fastq")
trimmed_reads = trim_adaptors(original_reads, "GATGACGGTGT")
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

Because we are using a FASTQ input file in this example, the `SeqRecord` objects have per-letter-annotation for the quality scores. By slicing the `SeqRecord` object the appropriate scores are used on
the trimmed records, so we can output them as a FASTQ file too.

Compared to the output of the previous example where we only looked for a primer/adaptor at the start
of each read, you may find some of the trimmed reads are quite short after trimming (e.g. if the adaptor
was found in the middle rather than near the start). So, let’s add a minimum length requirement as well:

```python
from Bio import SeqIO

def trim_adaptors(records, adaptor, min_len):
    # Trims perfect adaptor sequences, checks read length.
    # This is a generator function, the records argument should
    # be a list or iterator returning SeqRecord objects.
    len_adaptor = len(adaptor)  # cache this for later
    for record in records:
        len_record = len(record)  # cache this for later
        if len(record) < min_len:
            # Too short to keep
            continue
        index = record.seq.find(adaptor)
        if index == -1:
            # adaptor not found, so won't trim
            yield record
        elif len_record - index - len_adaptor >= min_len:
            # after trimming this will still be long enough
            yield record[index + len_adaptor :]
```

```python
original_reads = SeqIO.parse("SRR020192.fastq", "fastq")
trimmed_reads = trim_adaptors(original_reads, "GATGACGGTGT", 100)
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

By changing the format names, you could apply this to FASTA files instead. This code also could be
extended to do a fuzzy match instead of an exact match (maybe using a pairwise alignment, or taking into
account the read quality scores), but that will be much slower.

### 22.1.9 Converting FASTQ files

Back in Section 5.5.2 we showed how to use Bio.SeqIO to convert between two file formats. Here we’ll go
into a little more detail regarding FASTQ files which are used in second generation DNA sequencing. Please
refer to Cock et al. (2009) [6] for a longer description. FASTQ files store both the DNA sequence (as a
string) and the associated read qualities.

PHRED scores (used in most FASTQ files, and also in QUAL files, ACE files and SFF files) have become
a *de facto* standard for representing the probability of a sequencing error (here denoted by $P_e$) at a given
base using a simple base ten log transformation:

$$Q_{PHRED} = -10 \times \log_{10}(P_e)$$

(22.1)

This means a wrong read ($P_e = 1$) gets a PHRED quality of 0, while a very good read like $P_e = 0.00001$
gets a PHRED quality of 50. While for raw sequencing data qualities higher than this are rare, with post
processing such as read mapping or assembly, qualities of up to about 90 are possible (indeed, the MAQ tool allows for PHRED scores in the range 0 to 93 inclusive).

The FASTQ format has the potential to become a *de facto* standard for storing the letters and quality scores for a sequencing read in a single plain text file. The only fly in the ointment is that there are at least three versions of the FASTQ format which are incompatible and difficult to distinguish...

1. The original Sanger FASTQ format uses PHRED qualities encoded with an ASCII offset of 33. The NCBI are using this format in their Short Read Archive. We call this the fastq (or fastq-sanger) format in Bio.SeqIO.

2. Solexa (later bought by Illumina) introduced their own version using Solexa qualities encoded with an ASCII offset of 64. We call this the fastq-solexa format.

3. Illumina pipeline 1.3 onwards produces FASTQ files with PHRED qualities (which is more consistent), but encoded with an ASCII offset of 64. We call this the fastq-illumina format.

The Solexa quality scores are defined using a different log transformation:

\[
Q_{\text{Solexa}} = -10 \times \log_{10} \left( \frac{P_e}{1 - P_e} \right)
\]  

(22.2)

Given Solexa/Illumina have now moved to using PHRED scores in version 1.3 of their pipeline, the Solexa quality scores will gradually fall out of use. If you equate the error estimates \((P_e)\) these two equations allow conversion between the two scoring systems - and Biopython includes functions to do this in the Bio.SeqIO.QualityIO module, which are called if you use Bio.SeqIO to convert an old Solexa/Illumina file into a standard Sanger FASTQ file:

```python
from Bio import SeqIO

SeqIO.convert("solexa.fastq", "fastq-solexa", "standard.fastq", "fastq")
```

If you want to convert a new Illumina 1.3+ FASTQ file, all that gets changed is the ASCII offset because although encoded differently the scores are all PHRED qualities:

```python
from Bio import SeqIO

SeqIO.convert("illumina.fastq", "fastq-illumina", "standard.fastq", "fastq")
```

Note that using Bio.SeqIO.convert() like this is much faster than combining Bio.SeqIO.parse() and Bio.SeqIO.write() because optimized code is used for converting between FASTQ variants (and also for FASTQ to FASTA conversion).

For good quality reads, PHRED and Solexa scores are approximately equal, which means since both the fasta-solexa and fastq-illumina formats use an ASCII offset of 64 the files are almost the same. This was a deliberate design choice by Illumina, meaning applications expecting the old fasta-solexa style files will probably be OK using the newer fastq-illumina files (on good data). Of course, both variants are very different from the original FASTQ standard as used by Sanger, the NCBI, and elsewhere (format name fastq or fastq-sanger).

For more details, see the built in help (also online):

```python
>>> from Bio.SeqIO import QualityIO
>>> help(QualityIO)
```
22.1.10 Converting FASTA and QUAL files into FASTQ files

FASTQ files hold both sequences and their quality strings. FASTA files hold just sequences, while QUAL files hold just the qualities. Therefore a single FASTQ file can be converted to or from paired FASTA and QUAL files.

Going from FASTQ to FASTA is easy:

```python
from Bio import SeqIO
SeqIO.convert("example.fastq", "fastq", "example.fasta", "fasta")
```

Going from FASTQ to QUAL is also easy:

```python
from Bio import SeqIO
SeqIO.convert("example.fastq", "fastq", "example.qual", "qual")
```

However, the reverse is a little more tricky. You can use `Bio.SeqIO.parse()` to iterate over the records in a single file, but in this case we have two input files. There are several strategies possible, but assuming that the two files are really paired the most memory efficient way is to loop over both together. The code is a little fiddly, so we provide a function called `PairedFastaQualIterator` in the `Bio.SeqIO.QualityIO` module to do this. This takes two handles (the FASTA file and the QUAL file) and returns a `SeqRecord` iterator:

```python
from Bio.SeqIO.QualityIO import PairedFastaQualIterator
for record in PairedFastaQualIterator(open("example.fasta"), open("example.qual")):  
    print(record)
```

This function will check that the FASTA and QUAL files are consistent (e.g. the records are in the same order, and have the same sequence length). You can combine this with the `Bio.SeqIO.write()` function to convert a pair of FASTA and QUAL files into a single FASTQ files:

```python
from Bio import SeqIO
from Bio.SeqIO.QualityIO import PairedFastaQualIterator
with open("example.fasta") as f_handle, open("example.qual") as q_handle:
    records = PairedFastaQualIterator(f_handle, q_handle)
    count = SeqIO.write(records, "temp.fastq", "fastq")
print("Converted %i records" % count)
```

22.1.11 Indexing a FASTQ file

FASTQ files are usually very large, with millions of reads in them. Due to the sheer amount of data, you can’t load all the records into memory at once. This is why the examples above (filtering and trimming) iterate over the file looking at just one `SeqRecord` at a time.

However, sometimes you can’t use a big loop or an iterator - you may need random access to the reads. Here the `Bio.SeqIO.index()` function may prove very helpful, as it allows you to access any read in the FASTQ file by its name (see Section 5.4.2).

Again we’ll use the SRR020192.fastq file from the ENA (ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz), although this is actually quite a small FASTQ file with less than 50,000 reads:
>>> from Bio import SeqIO
>>> fq_dict = SeqIO.index("SRR020192.fastq", "fastq")
>>> len(fq_dict)
41892
>>> list(fq_dict.keys())[:4]
['SRR020192.38240', 'SRR020192.23181', 'SRR020192.40568', 'SRR020192.23186']
>>> fq_dict["SRR020192.23186"].seq
Seq('GTCCCAGTATTCGGATTTGTCTGCCAAAACAATGAAATTGACACAGTTTACAAC...CCG')

When testing this on a FASTQ file with seven million reads, indexing took about a minute, but record access was almost instant.

The sister function Bio.SeqIO.index_db() lets you save the index to an SQLite3 database file for near instantaneous reuse - see Section 5.4.2 for more details.

The example in Section 22.1.5 show how you can use the Bio.SeqIO.index() function to sort a large FASTA file – this could also be used on FASTQ files.

### 22.1.12 Converting SFF files

If you work with 454 (Roche) sequence data, you will probably have access to the raw data as a Standard Flowgram Format (SFF) file. This contains the sequence reads (called bases) with quality scores and the original flow information.

A common task is to convert from SFF to a pair of FASTA and QUAL files, or to a single FASTQ file. These operations are trivial using the Bio.SeqIO.convert() function (see Section 5.5.2):

```python
>>> from Bio import SeqIO
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff", "reads.fasta", "fasta")
10
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff", "reads.qual", "qual")
10
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff", "reads.fastq", "fastq")
10
```

Remember the convert function returns the number of records, in this example just ten. This will give you the *untrimmed* reads, where the leading and trailing poor quality sequence or adaptor will be in lower case. If you want the *trimmed* reads (using the clipping information recorded within the SFF file) use this:

```python
>>> from Bio import SeqIO
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff-trim", "trimmed.fasta", "fasta")
10
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff-trim", "trimmed.qual", "qual")
10
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff-trim", "trimmed.fastq", "fastq")
10
```

If you run Linux, you could ask Roche for a copy of their “off instrument” tools (often referred to as the Newbler tools). This offers an alternative way to do SFF to FASTA or QUAL conversion at the command line (but currently FASTQ output is not supported), e.g.

```
$ sffinfo -seq -notrim E3MFGYR02_random_10_reads.sff > reads.fasta
$ sffinfo -qual -notrim E3MFGYR02_random_10_reads.sff > reads.qual
$ sffinfo -seq -trim E3MFGYR02_random_10_reads.sff > trimmed.fasta
$ sffinfo -qual -trim E3MFGYR02_random_10_reads.sff > trimmed.qual
```
The way Biopython uses mixed case sequence strings to represent the trimming points deliberately mimics what the Roche tools do.

For more information on the Biopython SFF support, consult the built in help:

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

### 22.1.13 Identifying open reading frames

A very simplistic first step at identifying possible genes is to look for open reading frames (ORFs). By this we mean look in all six frames for long regions without stop codons – an ORF is just a region of nucleotides with no in frame stop codons.

Of course, to find a gene you would also need to worry about locating a start codon, possible promoters – and in Eukaryotes there are introns to worry about too. However, this approach is still useful in viruses and Prokaryotes.

To show how you might approach this with Biopython, we’ll need a sequence to search, and as an example we’ll again use the bacterial plasmid – although this time we’ll start with a plain FASTA file with no pre-marked genes: `NC_005816.fna`. This is a bacterial sequence, so we’ll want to use NCBI codon table 11 (see Section 3.8 about translation).

```python
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.fna", "fasta")
>>> table = 11
>>> min_pro_len = 100
```

Here is a neat trick using the `Seq` object’s `split` method to get a list of all the possible ORF translations in the six reading frames:

```python
>>> for strand, nuc in [(+1, record.seq), (-1, record.seq.reverse_complement())]:
...     for frame in range(3):
...         length = 3 * ((len(record) - frame) // 3)  # Multiple of three
...         for pro in nuc[frame : frame + length].translate(table).split("*")):
...             if len(pro) >= min_pro_len:
...                 print("%s...%s - length %i, strand %i, frame %i" % (pro[:30], pro[-3:], len(pro), strand, frame))

GCLMKKKSSIVATIITILSGSANAASSQLIP...YRF - length 315, strand 1, frame 0
KSQELRQTPASSTLHRLLQGSGVMMEL....NEP - length 285, strand 1, frame 1
GLNCSFFSICNWKFIDINLRFQIIYLCRN...YIH - length 176, strand 1, frame 1
VKKILYIKALFLCTVIKLRRGQVMEK...GVA - length 355, strand 1, frame 2
KKEHVSKRRPQRRRRPFFFHRPPPDE...PTR - length 128, strand 1, frame 2
TQKQNCMSQMIWQLRQNTARKTRQNRARI...AIK - length 100, strand 1, frame 2
QGSGYAFHASILSGASVSYFLHLHAVK....CSD - length 114, strand -1, frame 0
IYSTSEHTGQVRMLDTPRMLRPRSPESQTR...FHV - length 111, strand -1, frame 0
WGKLQVIGLSMVHFLSQRFDWLNEQEDA...ESK - length 125, strand -1, frame 1
RGIFMSTDTHVRSNGSGVPAFLSFSTLSSY...LLK - length 130, strand -1, frame 1
WHDKVTGVLHMPFHLSFLPQCAQGSGR...VXR - length 111, strand -1, frame 1
LSHTVTDFTDQMAQVGLCQCVNVFLDEVG...KAA - length 107, strand -1, frame 2
RALTGLSAPGIRQRSQCDRLRELRYVPVSL...PLQ - length 119, strand -1, frame 2
```
Note that here we are counting the frames from the 5’ end (start) of each strand. It is sometimes easier to always count from the 5’ end (start) of the forward strand.

You could easily edit the above loop based code to build up a list of the candidate proteins, or convert this to a list comprehension. Now, one thing this code doesn’t do is keep track of where the proteins are.

You could tackle this in several ways. For example, the following code tracks the locations in terms of the protein counting, and converts back to the parent sequence by multiplying by three, then adjusting for the frame and strand:

```python
from Bio import SeqIO

record = SeqIO.read("NC_005816.gb", "genbank")
table = 11
min_pro_len = 100

def find_orfs_with_trans(seq, trans_table, min_protein_length):
    answer = []
    seq_len = len(seq)
    for strand, nuc in [(+1, seq), (-1, seq.reverse_complement())]:
        for frame in range(3):
            trans = nuc[frame:].translate(trans_table)
            trans_len = len(trans)
            aa_start = 0
            aa_end = 0
            while aa_start < trans_len:
                aa_end = trans.find("*", aa_start)
                if aa_end == -1:
                    aa_end = trans_len
                if aa_end - aa_start >= min_protein_length:
                    if strand == 1:
                        start = frame + aa_start * 3
                        end = min(seq_len, frame + aa_end * 3 + 3)
                    else:
                        start = seq_len - frame - aa_end * 3 - 3
                        end = seq_len - frame - aa_start * 3
                    answer.append((start, end, strand, trans[aa_start:aa_end]))
                aa_start = aa_end + 1
    answer.sort()
    return answer

orf_list = find_orfs_with_trans(record.seq, table, min_pro_len)
for start, end, strand, pro in orf_list:
    print("%s...%s - length %i, strand %i, %i:%i"
          % (pro[:30], pro[-3:], len(pro), strand, start, end))

And the output:

NQIQGVICSPDSGEFMVTFETVMEIKILHK...GVA - length 355, strand 1, 41:1109
WDVKVTGFVLHHPFHLTFSLCPEGATQSGR...VKR - length 111, strand -1, 491:827

420
If you comment out the sort statement, then the protein sequences will be shown in the same order as before, so you can check this is doing the same thing. Here we have sorted them by location to make it easier to compare to the actual annotation in the GenBank file (as visualized in Section 19.1.9).

If however all you want to find are the locations of the open reading frames, then it is a waste of time to translate every possible codon, including doing the reverse complement to search the reverse strand too. All you need to do is search for the possible stop codons (and their reverse complements). Using regular expressions is an obvious approach here (see the Python module re). These are an extremely powerful (but rather complex) way of describing search strings, which are supported in lots of programming languages and also command line tools like grep as well). You can find whole books about this topic!

### 22.2 Sequence parsing plus simple plots

This section shows some more examples of sequence parsing, using the Bio.SeqIO module described in Chapter 5, plus the Python library matplotlib’s pylab plotting interface (see the matplotlib website for a tutorial). Note that to follow these examples you will need matplotlib installed - but without it you can still try the data parsing bits.

#### 22.2.1 Histogram of sequence lengths

There are lots of times when you might want to visualize the distribution of sequence lengths in a dataset – for example the range of contig sizes in a genome assembly project. In this example we’ll reuse our orchid FASTA file ls_orchid.fasta which has only 94 sequences.

```python
from Bio import SeqIO
sizes = [len(rec) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")]
```

Now that we have the lengths of all the genes (as a list of integers), we can use the matplotlib histogram function to display it.

```python
from Bio import SeqIO
sizes = [len(rec) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")]
```
import pylab

pylab.hist(sizes, bins=20)
pylab.title("%i orchid sequences
Lengths %i to %i" % (len(sizes), min(sizes), max(sizes)))
pylab.xlabel("Sequence length (bp)"
pylab.ylabel("Count")
pylab.show()

Figure 22.1: Histogram of orchid sequence lengths.

That should pop up a new window containing the graph shown in Figure 22.1. Notice that most of these orchid sequences are about 740 bp long, and there could be two distinct classes of sequence here with a subset of shorter sequences.

Tip: Rather than using `pylab.show()` to show the plot in a window, you can also use `pylab.savefig(...)` to save the figure to a file (e.g. as a PNG or PDF).

### 22.2.2 Plot of sequence GC%

Another easily calculated quantity of a nucleotide sequence is the GC%. You might want to look at the GC% of all the genes in a bacterial genome for example, and investigate any outliers which could have been recently acquired by horizontal gene transfer. Again, for this example we’ll reuse our orchid FASTA file ls_orchid.fasta.

First of all, we will use `Bio.SeqIO` to parse the FASTA file and compile a list of all the GC percentages. Again, you could do this with a for loop, but I prefer this:
from Bio import SeqIO
from Bio.SeqUtils import gc_fraction

gc_values = sorted(100 * gc_fraction(rec.seq) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")
)

Having read in each sequence and calculated the GC%, we then sorted them into ascending order. Now we'll take this list of floating point values and plot them with matplotlib:

```python
import matplotlib.pyplot as plt

plt.plot(gc_values)
plt.title("%i orchid sequences
GC% %0.1f to %0.1f" % (len(gc_values), min(gc_values), max(gc_values)))
plt.xlabel("Genes")
plt.ylabel("GC%")
plt.show()
```

Figure 22.2: Histogram of orchid sequence lengths.

As in the previous example, that should pop up a new window with the graph shown in Figure 22.2. If you tried this on the full set of genes from one organism, you'd probably get a much smoother plot than this.
22.2.3 Nucleotide dot plots

A dot plot is a way of visually comparing two nucleotide sequences for similarity to each other. A sliding window is used to compare short sub-sequences to each other, often with a mismatch threshold. Here for simplicity we'll only look for perfect matches (shown in black in Figure 22.3).

![Dot plot using window size 7 (allowing no mis-matches)](image)

Figure 22.3: Nucleotide dot plot of two orchid sequence lengths (using pylab's imshow function).

To start off, we'll need two sequences. For the sake of argument, we'll just take the first two from our orchid FASTA file ls_orchid.fasta:

```python
from Bio import SeqIO

with open("ls_orchid.fasta") as in_handle:
    record_iterator = SeqIO.parse(in_handle, "fasta")
    rec_one = next(record_iterator)
    rec_two = next(record_iterator)
```

We're going to show two approaches. Firstly, a simple naive implementation which compares all the window sized sub-sequences to each other to compiles a similarity matrix. You could construct a matrix or array object, but here we just use a list of lists of booleans created with a nested list comprehension:

```python
window = 7
seq_one = rec_one.seq.upper()
seq_two = rec_two.seq.upper()
data = [
    [True if seq_one[i : i + window] != seq_two[j : j + window] else False
     for j in range(len(seq_two) - window + 1)]
     for i in range(len(seq_one) - window + 1)]
```
for j in range(len(seq_one) - window):
]
for i in range(len(seq_two) - window)
]

Note that we have not checked for reverse complement matches here. Now we'll use the matplotlib's 
pylab.imshow() function to display this data, first requesting the gray color scheme so this is done in black 
and white:

import pylab
pylab.gray()
pylab.imshow(data)
pylab.xlabel("%s (length %i bp)" % (rec_one.id, len(rec_one)))
pylab.ylabel("%s (length %i bp)" % (rec_two.id, len(rec_two)))
pylab.title("Dot plot using window size %i
(allowing no mis-matches)" % window)
pylab.show()

That should pop up a new window showing the graph in Figure 22.3. As you might have expected, these 
two sequences are very similar with a partial line of window sized matches along the diagonal. There are no 
off diagonal matches which would be indicative of inversions or other interesting events.

The above code works fine on small examples, but there are two problems applying this to larger sequences, 
which we will address below. First of all, this brute force approach to the all against all comparisons is very 
slow. Instead, we'll compile dictionaries mapping the window sized sub-sequences to their locations, and 
then take the set intersection to find those sub-sequences found in both sequences. This uses more memory, 
but is much faster. Secondly, the pylab.imshow() function is limited in the size of matrix it can display. 
As an alternative, we'll use the pylab.scatter() function.

We start by creating dictionaries mapping the window-sized sub-sequences to locations:

window = 7
dict_one = {}
dict_two = {}
for seq, section_dict in [
    (rec_one.seq.upper(), dict_one),
    (rec_two.seq.upper(), dict_two),
    ]:
    for i in range(len(seq) - window):
        section = seq[i : i + window]
        try:
            section_dict[section].append(i)
        except KeyError:
            section_dict[section] = [i]

# Now find any sub-sequences found in both sequences
matches = set(dict_one).intersection(dict_two)
print("%i unique matches" % len(matches))

In order to use the pylab.scatter() we need separate lists for the x and y coordinates:

# Create lists of x and y coordinates for scatter plot
x = []
y = []
for section in matches:
    for i in dict_one[section]:
        x.append(section[i])
        y.append(i)
for j in dict_two[section]:
    x.append(i)
    y.append(j)

We are now ready to draw the revised dot plot as a scatter plot:

```python
import pylab

pylab.cla()  # clear any prior graph
pylab.gray()
pylab.scatter(x, y)
pylab.xlim(0, len(rec_one) - window)
pylab.ylim(0, len(rec_two) - window)
pylab.xlabel("%s (length %i bp)" % (rec_one.id, len(rec_one)))
pylab.ylabel("%s (length %i bp)" % (rec_two.id, len(rec_two)))
pylab.title("Dot plot using window size %i
(allowing no mis-matches)" % window)
pylab.show()
```

That should pop up a new window showing the graph in Figure 22.4. Personally I find this second plot

![Dot plot using window size 7 (allowing no mis-matches)](image)

Figure 22.4: Nucleotide dot plot of two orchid sequence lengths (using `pylab`’s `scatter` function).

much easier to read! Again note that we have not checked for reverse complement matches here – you could extend this example to do this, and perhaps plot the forward matches in one color and the reverse matches in another.
22.2.4 Plotting the quality scores of sequencing read data

If you are working with second generation sequencing data, you may want to try plotting the quality data. Here is an example using two FASTQ files containing paired end reads, `SRR001666_1.fastq` for the forward reads, and `SRR001666_2.fastq` for the reverse reads. These were downloaded from the ENA sequence read archive FTP site (`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR001/SRR001666/SRR001666_1.fastq.gz` and `ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR001/SRR001666/SRR001666_2.fastq.gz`), and are from *E. coli* – see [https://www.ebi.ac.uk/ena/data/view/SRR001666](https://www.ebi.ac.uk/ena/data/view/SRR001666) for details.

In the following code the `pylab.subplot(...)` function is used in order to show the forward and reverse qualities on two subplots, side by side. There is also a little bit of code to only plot the first fifty reads.

```python
import pylab
from Bio import SeqIO

for subfigure in [1, 2]:
    filename = "SRR001666_%i.fastq" % subfigure
    pylab.subplot(1, 2, subfigure)
    for i, record in enumerate(SeqIO.parse(filename, "fastq")):
        if i >= 50:
            break
        # trick!
        pylab.plot(record.letter_annotations["phred_quality"])
    pylab.xlim(0, 45)
    pylab.ylabel("PHRED quality score")
    pylab.xlabel("Position")
    pylab.savefig("SRR001666.png")
print("Done")
```

You should note that we are using the `Bio.SeqIO` format name `fastq` here because the NCBI has saved these reads using the standard Sanger FASTQ format with PHRED scores. However, as you might guess from the read lengths, this data was from an Illumina Genome Analyzer and was probably originally in one of the two Solexa/Illumina FASTQ variant file formats instead.

This example uses the `pylab.savefig(...)` function instead of `pylab.show(...)`, but as mentioned before both are useful. The result is shown in Figure 22.5.

22.3 BioSQL – storing sequences in a relational database

BioSQL is a joint effort between the OBF projects (BioPerl, BioJava etc) to support a shared database schema for storing sequence data. In theory, you could load a GenBank file into the database with BioPerl, then using Biopython extract this from the database as a record object with features - and get more or less the same thing as if you had loaded the GenBank file directly as a `SeqRecord` using `Bio.SeqIO` (Chapter 5).

Biopython’s BioSQL module is currently documented at [http://biopython.org/wiki/BioSQL](http://biopython.org/wiki/BioSQL) which is part of our wiki pages.
Figure 22.5: Quality plot for some paired end reads.
Chapter 23

The Biopython testing framework

Biopython has a regression testing framework (the file run_tests.py) based on unittest, the standard unit testing framework for Python. Providing comprehensive tests for modules is one of the most important aspects of making sure that the Biopython code is as bug-free as possible before going out. It also tends to be one of the most undervalued aspects of contributing. This chapter is designed to make running the Biopython tests and writing good test code as easy as possible. Ideally, every module that goes into Biopython should have a test (and should also have documentation!). All our developers, and anyone installing Biopython from source, are strongly encouraged to run the unit tests.

23.1 Running the tests

When you download the Biopython source code, or check it out from our source code repository, you should find a subdirectory called Tests. This contains the key script run_tests.py, lots of individual scripts named test_XXX.py, and lots of other subdirectories which contain input files for the test suite.

As part of building and installing Biopython you will typically run the full test suite at the command line from the Biopython source top level directory using the following:

```
$ python setup.py test
```

This is actually equivalent to going to the Tests subdirectory and running:

```
$ python run_tests.py
```

You’ll often want to run just some of the tests, and this is done like this:

```
$ python run_tests.py test_SeqIO.py test_AlignIO.py
```

When giving the list of tests, the .py extension is optional, so you can also just type:

```
$ python run_tests.py test_SeqIO test_AlignIO
```

To run the docstring tests (see section 23.3), you can use

```
$ python run_tests.py doctest
```

You can also skip any tests which have been setup with an explicit online component by adding --offline, e.g.

```
$ python run_tests.py --offline
```
By default, `run_tests.py` runs all tests, including the docstring tests. If an individual test is failing, you can also try running it directly, which may give you more information.

Tests based on Python's standard `unittest` framework will import `unittest` and then define `unittest.TestCase` classes, each with one or more sub-tests as methods starting with `test_` which check some specific aspect of the code.

### 23.1.1 Running the tests using Tox

Like most Python projects, you can also use Tox to run the tests on multiple Python versions, provided they are already installed in your system.

We do not provide the configuration `tox.ini` file in our code base because of difficulties pinning down user-specific settings (e.g. executable names of the Python versions). You may also only be interested in testing Biopython only against a subset of the Python versions that we support.

If you are interested in using Tox, you could start with the example `tox.ini` shown below:

```ini
[tox]
envlist = pypy,py38,py39

[testenv]
changedir = Tests
commands = {envpython} run_tests.py --offline
deps =
    numpy
    reportlab
```

Using the template above, executing `tox` will test your Biopython code against PyPy, Python 3.8 and 3.9. It assumes that those Pythons' executables are named “python3.8” for Python 3.8, and so on.

### 23.2 Writing tests

Let's say you want to write some tests for a module called `Biospam`. This can be a module you wrote, or an existing module that doesn't have any tests yet. In the examples below, we assume that `Biospam` is a module that does simple math.

Each Biopython test consists of a script containing the test itself, and optionally a directory with input files used by the test:

1. `test_Biospam.py` – The actual test code for your module.
2. `Biospam` [optional]– A directory where any necessary input files will be located. If you have any output files that should be manually reviewed, output them here (but this is discouraged) to prevent clogging up the main Tests directory. In general, use a temporary file/folder.

Any script with a `test_` prefix in the `Tests` directory will be found and run by `run_tests.py`. Below, we show an example test script `test_Biospam.py`. If you put this script in the Biopython `Tests` directory, then `run_tests.py` will find it and execute the tests contained in it:

```
$ python run_tests.py
  test_Ace ... ok
  test_AlignIO ... ok
  test_BioSQL ... ok
  test_BioSQL_SeqIO ... ok
  test_Biospam ... ok
```
23.2.1 Writing a test using unittest

The unittest-framework has been included with Python since version 2.1, and is documented in the Python Library Reference (which I know you are keeping under your pillow, as recommended). There is also online documentation for unittest. If you are familiar with the unittest system (or something similar like the nose test framework), you shouldn’t have any trouble. You may find looking at the existing examples within Biopython helpful too.

Here’s a minimal unittest-style test script for Biospam, which you can copy and paste to get started:

```python
import unittest
from Bio import Biospam

class BiospamTestAddition(unittest.TestCase):
    def test_addition1(self):
        result = Biospam.addition(2, 3)
        self.assertEqual(result, 5)

    def test_addition2(self):
        result = Biospam.addition(9, -1)
        self.assertEqual(result, 8)

class BiospamTestDivision(unittest.TestCase):
    def test_division1(self):
        result = Biospam.division(3.0, 2.0)
        self.assertAlmostEqual(result, 1.5)

    def test_division2(self):
        result = Biospam.division(10.0, -2.0)
        self.assertAlmostEqual(result, -5.0)

if __name__ == "__main__":
    runner = unittest.TextTestRunner(verbosity=2)
    unittest.main(testRunner=runner)
```

In the division tests, we use `assertAlmostEqual` instead of `assertEqual` to avoid tests failing due to roundoff errors; see the unittest chapter in the Python documentation for details and for other functionality available in unittest (online reference).

These are the key points of unittest-based tests:

- Test cases are stored in classes that derive from `unittest.TestCase` and cover one basic aspect of your code

- You can use methods `setUp` and `tearDown` for any repeated code which should be run before and after each test method. For example, the `setUp` method might be used to create an instance of the object...
you are testing, or open a file handle. The tearDown should do any “tidying up”, for example closing the file handle.

- The tests are prefixed with test_ and each test should cover one specific part of what you are trying to test. You can have as many tests as you want in a class.

- At the end of the test script, you can use

```python
if __name__ == "__main__":
    runner = unittest.TextTestRunner(verbosity=2)
    unittest.main(testRunner=runner)
```

to execute the tests when the script is run by itself (rather than imported from run_tests.py). If you run this script, then you’ll see something like the following:

```bash
$ python test_BiospamMyModule.py
```

```
test_addition1 (__main__.TestAddition) ... ok
test_addition2 (__main__.TestAddition) ... ok
test_division1 (__main__.TestDivision) ... ok
```

```
$ python test_BiospamMyModule.py
```

```
test_addition1 (__main__.TestAddition) ... ok
```

```
test_division1 (__main__.TestDivision) ... ok
```

```
Ran 4 tests in 0.059s
OK
```

- To indicate more clearly what each test is doing, you can add docstrings to each test. These are shown when running the tests, which can be useful information if a test is failing.

```python
import unittest
from Bio import Biospam

class BiospamTestAddition(unittest.TestCase):
    def test_addition1(self):
        """An addition test""
        result = Biospam.addition(2, 3)
        self.assertEqual(result, 5)

    def test_addition2(self):
        """A second addition test""
        result = Biospam.addition(9, -1)
        self.assertEqual(result, 8)

class BiospamTestDivision(unittest.TestCase):
    def test_division1(self):
        """Now let’s check division""
        result = Biospam.division(3.0, 2.0)
        self.assertAlmostEqual(result, 1.5)

    def test_division2(self):
        """A second division test""
        result = Biospam.division(10.0, -2.0)
```

432
self.assertAlmostEqual(result, -5.0)

if __name__ == "__main__":
    runner = unittest.TextTestRunner(verbosity=2)
    unittest.main(testRunner=runner)

Running the script will now show you:

$ python test_BiospamMyModule.py
An addition test ... ok
A second addition test ... ok
Now let's check division ... ok
A second division test ... ok

Ran 4 tests in 0.001s

OK

If your module contains docstring tests (see section 23.3), you may want to include those in the tests to be run. You can do so as follows by modifying the code under if __name__ == "__main__": to look like this:

if __name__ == "__main__":
    unittest_suite = unittest.TestLoader().loadTestsFromName("test_Biospam")
    doctest_suite = doctest.DocTestSuite(Biospam)
    suite = unittest.TestSuite((unittest_suite, doctest_suite))
    runner = unittest.TextTestRunner(sys.stdout, verbosity=2)
    runner.run(suite)

This is only relevant if you want to run the docstring tests when you execute python test_Biospam.py if it has some complex run-time dependency checking.

In general instead include the docstring tests by adding them to the run_tests.py as explained below.

23.3 Writing doctests

Python modules, classes and functions support built in documentation using docstrings. The doctest framework (included with Python) allows the developer to embed working examples in the docstrings, and have these examples automatically tested.

Currently only part of Biopython includes doctests. The run_tests.py script takes care of running the doctests. For this purpose, at the top of the run_tests.py script is a manually compiled list of modules to skip, important where optional external dependencies which may not be installed (e.g. the Reportlab and NumPy libraries). So, if you've added some doctests to the docstrings in a Biopython module, in order to have them excluded in the Biopython test suite, you must update run_tests.py to include your module. Currently, the relevant part of run_tests.py looks as follows:

# Following modules have historic failures. If you fix one of these
# please remove here!
EXCLUDE_DOCTEST_MODULES = [
    "Bio.PDB",
    "Bio.PDB.AbstractPropertyMap",
]
"Bio.Phylo.Applications._Fasttree",
"Bio.Phylo._io",
"Bio.Phylo.TreeConstruction",
"Bio.Phylo._utils",

# Exclude modules with online activity
# They are not excluded by default, use --offline to exclude them

# Silently ignore any doctests for modules requiring numpy!
if numpy is None:
    EXCLUDE_DOCTEST_MODULES.extend(
        ["Bio.Affy.CelFile",
         "Bio.Cluster",
         # ...
        ]
    )

Note that we regard doctests primarily as documentation, so you should stick to typical usage. Generally complicated examples dealing with error conditions and the like would be best left to a dedicated unit test.

Note that if you want to write doctests involving file parsing, defining the file location complicates matters. Ideally use relative paths assuming the code will be run from the Tests directory, see the Bio.SeqIO doctests for an example of this.

To run the docstring tests only, use

$ python run_tests.py doctest

Note that the doctest system is fragile and care is needed to ensure your output will match on all the different versions of Python that Biopython supports (e.g. differences in floating point numbers).

### 23.4 Writing doctests in the Tutorial

This Tutorial you are reading has a lot of code snippets, which are often formatted like a doctest. We have our own system in file test_Tutorial.py to allow tagging code snippets in the Tutorial source to be run as Python doctests. This works by adding special %doctest comment lines before each Python block, e.g.

```python
%doctest
\begin{minted}{pycon}
>>> from Bio.Seq import Seq
>>> s = Seq("ACGT")
>>> len(s)
4
\end{minted}
```

Often code examples are not self-contained, but continue from the previous Python block. Here we use the magic comment %cont-doctest as shown here:

```python
%cont-doctest
\begin{minted}{pycon}
```
The special %doctest comment line can take a working directory (relative to the Doc/ folder) to use if you have any example data files, e.g. %doctest examples will use the Doc/examples folder, while %doctest ../Tests/GenBank will use the Tests/GenBank folder.

After the directory argument, you can specify any Python dependencies which must be present in order to run the test by adding lib:XXX to indicate import XXX must work, e.g. %doctest examples lib:numpy

You can run the Tutorial doctests via:

```
$ python test_Tutorial.py
```

or:

```
$ python run_tests.py test_Tutorial.py
```
Chapter 24

Where to go from here – contributing to Biopython

24.1 Bug Reports + Feature Requests

Getting feedback on the Biopython modules is very important to us. Open-source projects like this benefit greatly from feedback, bug-reports (and patches!) from a wide variety of contributors.

The main forums for discussing feature requests and potential bugs are the Biopython mailing list and issues or pull requests on GitHub.

Additionally, if you think you’ve found a new bug, you can submit it to our issue tracker at https://github.com/biopython/biopython/issues (this replaced the older Open Bioinformatics Foundation hosted RedMine tracker). This way, it won’t get buried in anyone’s Inbox and forgotten about.

24.2 Mailing lists and helping newcomers

We encourage all our uses to sign up to the main Biopython mailing list. Once you’ve got the hang of an area of Biopython, we’d encourage you to help answer questions from beginners. After all, you were a beginner once.

24.3 Contributing Documentation

We’re happy to take feedback or contributions - either via a bug-report or on the Mailing List. While reading this tutorial, perhaps you noticed some topics you were interested in which were missing, or not clearly explained. There is also Biopython’s built in documentation (the docstrings, these are also online), where again, you may be able to help fill in any blanks.

24.4 Contributing cookbook examples

As explained in Chapter 22, Biopython now has a wiki collection of user contributed “cookbook” examples, http://biopython.org/wiki/Category:Cookbook – maybe you can add to this?

24.5 Maintaining a distribution for a platform

We currently provide source code archives (suitable for any OS, if you have the right build tools installed), and pre-compiled wheels via https://github.com/biopython/biopython-wheels to cover the major operating
systems.

Most major Linux distributions have volunteers who take these source code releases, and compile them into packages for Linux users to easily install (taking care of dependencies etc). This is really great and we are of course very grateful. If you would like to contribute to this work, please find out more about how your Linux distribution handles this. There is a similar process for conda packages via https://github.com/conda-forge/biopython-feedstock thanks to the conda-forge team.

Below are some tips for certain platforms to maybe get people started with helping out:

**Windows** – You must first make sure you have a C compiler on your Windows computer, and that you can compile and install things (this is the hard bit - see the Biopython installation instructions for info on how to do this).

**RPMs** – RPMs are pretty popular package systems on some Linux platforms. There is lots of documentation on RPMs available at http://www.rpm.org to help you get started with them. To create an RPM for your platform is really easy. You just need to be able to build the package from source (having a C compiler that works is thus essential) – see the Biopython installation instructions for more info on this.

To make the RPM, you just need to do:

```
$ python setup.py bdist_rpm
```

This will create an RPM for your specific platform and a source RPM in the directory *dist*. This RPM should be good and ready to go, so this is all you need to do! Nice and easy.

**Macintosh** – Since Apple moved to Mac OS X, things have become much easier on the Mac. We generally treat it as just another Unix variant, and installing Biopython from source is just as easy as on Linux. The easiest way to get all the GCC compilers etc installed is to install Apple’s X-Code. We might be able to provide click and run installers for Mac OS X, but to date there hasn’t been any demand.

Once you’ve got a package, please test it on your system to make sure it installs everything in a good way and seems to work properly. Once you feel good about it, make a pull request on GitHub and write to our Biopython mailing list. You’ve done it. Thanks!

### 24.6 Contributing Unit Tests

Even if you don’t have any new functionality to add to Biopython, but you want to write some code, please consider extending our unit test coverage. We’ve devoted all of Chapter 23 to this topic.

### 24.7 Contributing Code

There are no barriers to joining Biopython code development other than an interest in creating biology-related code in Python. The best place to express an interest is on the Biopython mailing lists – just let us know you are interested in coding and what kind of stuff you want to work on. Normally, we try to have some discussion on modules before coding them, since that helps generate good ideas – then just feel free to jump right in and start coding!

The main Biopython release tries to be fairly uniform and interworkable, to make it easier for users. You can read about some of (fairly informal) coding style guidelines we try to use in Biopython in the contributing documentation at http://biopython.org/wiki/contributing. We also try to add code to the distribution along with tests (see Chapter 23 for more info on the regression testing framework) and documentation, so that everything can stay as workable and well documented as possible (including docstrings). This is, of course, the most ideal situation, under many situations you’ll be able to find other people on the list who
will be willing to help add documentation or more tests for your code once you make it available. So, to end this paragraph like the last, feel free to start working!

Please note that to make a code contribution you must have the legal right to contribute it and license it under the Biopython license. If you wrote it all yourself, and it is not based on any other code, this shouldn’t be a problem. However, there are issues if you want to contribute a derivative work - for example something based on GPL or LPGL licensed code would not be compatible with our license. If you have any queries on this, please discuss the issue on the mailing list or GitHub.

Another point of concern for any additions to Biopython regards any build time or run time dependencies. Generally speaking, writing code to interact with a standalone tool (like BLAST, EMBOSS or ClustalW) doesn’t present a big problem. However, any dependency on another library - even a Python library (especially one needed in order to compile and install Biopython like NumPy) would need further discussion.

Additionally, if you have code that you don’t think fits in the distribution, but that you want to make available, we maintain Script Central (http://biopython.org/wiki/Scriptcentral) which has pointers to freely available code in Python for bioinformatics.

Hopefully this documentation has got you excited enough about Biopython to try it out (and most importantly, contribute!). Thanks for reading all the way through!
Chapter 25

Appendix: Useful stuff about Python

If you haven’t spent a lot of time programming in Python, many questions and problems that come up in using Biopython are often related to Python itself. This section tries to present some ideas and code that come up often (at least for us!) while using the Biopython libraries. If you have any suggestions for useful pointers that could go here, please contribute!

25.1 What the heck is a handle?

Handles are mentioned quite frequently throughout this documentation, and are also fairly confusing (at least to me!). Basically, you can think of a handle as being a “wrapper” around text information.

Handles provide (at least) two benefits over plain text information:

1. They provide a standard way to deal with information stored in different ways. The text information can be in a file, or in a string stored in memory, or the output from a command line program, or at some remote website, but the handle provides a common way of dealing with information in all of these formats.

2. They allow text information to be read incrementally, instead of all at once. This is really important when you are dealing with huge text files which would use up all of your memory if you had to load them all.

Handles can deal with text information that is being read (e. g. reading from a file) or written (e. g. writing information to a file). In the case of a “read” handle, commonly used functions are read(), which reads the entire text information from the handle, and readline(), which reads information one line at a time. For “write” handles, the function write() is regularly used.

The most common usage for handles is reading information from a file, which is done using the built-in Python function open. Here, we handle to the file m_cold.fasta which you can download here (or find included in the Biopython source code as Doc/examples/m_cold.fasta).

```python
>>> handle = open("m_cold.fasta", "r")
>>> handle.readline()
"gi|8332116|gb|BE037100.1|BE037100 MP14H09 MP Mesembryanthemum ...
"
```

Handles are regularly used in Biopython for passing information to parsers. For example, since Biopython 1.54 the main functions in Bio.SeqIO and Bio.AlignIO have allowed you to use a filename instead of a handle:

```python
from Bio import SeqIO
```

439
for record in SeqIO.parse("m_cold.fasta", "fasta"):
    print(record.id, len(record))

On older versions of Biopython you had to use a handle, e.g.

from Bio import SeqIO

handle = open("m_cold.fasta", "r")
for record in SeqIO.parse(handle, "fasta"):
    print(record.id, len(record))
handle.close()

This pattern is still useful - for example suppose you have a gzip compressed FASTA file you want to parse:

import gzip
from Bio import SeqIO

handle = gzip.open("m_cold.fasta.gz", "rt")
for record in SeqIO.parse(handle, "fasta"):
    print(record.id, len(record))
handle.close()

With our parsers for plain text files, it is essential to use gzip in text mode (the default is binary mode).
See Section 5.2 for more examples like this, including reading bzip2 compressed files.

25.1.1 Creating a handle from a string

One useful thing is to be able to turn information contained in a string into a handle. The following example shows how to do this using StringIO from the Python standard library:

>>> my_info = "A string\n with multiple lines."
>>> print(my_info)
A string
 with multiple lines.
>>> from io import StringIO
>>> my_info_handle = StringIO(my_info)
>>> first_line = my_info_handle.readline()
>>> print(first_line)
A string
<BLANKLINE>
>>> second_line = my_info_handle.readline()
>>> print(second_line)
with multiple lines.
Bibliography


444