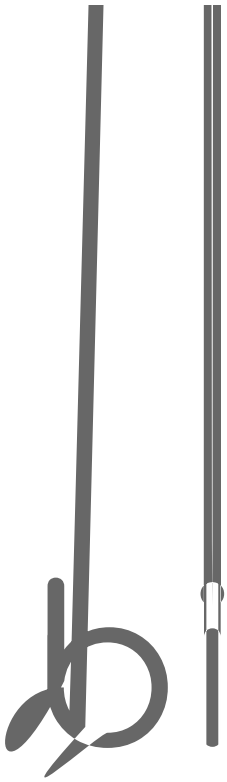


Biopython Tutorial and Cookbook

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

Contents

1 Introduction 9
1.1 What is Biopython? 9
1.2 What can I nd in the Biopython package 9
1.3 Installing Biopython 10
1.4 Frequently Asked Questions (FAQ) 11

2 Quick Start { What can you do with Biopython?

4.3.1 SeqFeature objects 39
4.3.2 Pos456nseca456nse . . . . . .
4.331 Seuencredescribedeyef(eature)-333(re)-333loca456n. . . . . .
441Comparis6n. . . . . .
451Referencrse . . . . . .
461Threformatede . . . . . .
471Slicinge(SeRecorde)]TJ 0 g 0 G [10160(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-5
481Addinge SeReco1rde(ob)-56(jects)]TJ 0 g 0 G [4560(.)-500(.)-500(.)-500(.)-500(.)-500(.)-499(.)-500(.)-500(.)-500(.)-5
491Reve1rs1e-complementinge SeRecorde(ob)-56(jects)]TJ 0 g 0 G [3780(.)-500(.)-500(.)-499(.)-500(.)-500(.)-500(.)-500
. . . SeuencreFilrse . . . . . .
verererecordsesSeuencre lee . . . . . .
recordse.lee . . . . . .
. . . . . .
Parsingefromec1ompressede . . . . . .
Parsingefromethrenets . . . . . .
Precordsefromethrenets . . . . . .
Pfromethrenets . . . . . .
(Seuencre)-34(lese)-333ase . . . . . .
. Seuencre lesearirse . . . . . .
Seuencre lesearirse lese(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-50
WF SeuencreFilrse . . . . . .
. . . tripse . . . . . .
ertingeetweenesSeuencre leeformatse . . . . . .
Q-5004-pa[-1r. . . . . .
ertinge70TJ 1 0 0 rg 1 0 0 RG/FF8 9.9626 Tf (49440 -11.955 Td [53.1)-(.)-5006(arsinMultipl(re)-33ge)-343(re)-4(Align g 032(tre
481Addinge SeReco1rde(ob)-56(jects)]TJ-34(f(eature)-333744(sSeuencre)-33(lee)]TJ 0 g 0 G [8110(.)-560(.(2r)-
. . . tripse
ertingeetweenesSeuencre 8114lese. . . . . .
ertinge82500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-500(.)-5006. . . . . .
. . . tripse . . . sSeuencre lee-arra0 7(y.)-500(.)-500(.)-
4.331 Seuencredescribedeye . . .ol. . . . . .
. tripse Seuencredescribedeye .lustalW . . . . . .
WF SeuencreFilrse
. tripse Seuencredescribedredescribedeye . ...Seue
... .o-333(Seuencrse)]TJ595(.)-500(.)-500(.)-500(.)-500(.)]TJ 0 g 0 G [-111566s . sSeuencre lee
. tripse Seuencredescribedeye . . (.s
. tripse Seuencredescribedeye .

7 BLAST 95
7.1 Running BLAST over the Internet 95
7.2 Running BLAST locally 97
7.2.1 Introduction 97
7.2.2 Standalone NCBI BLAST+ 97
7.2.3 Other versions of BLAST 98
7.3 Parsing BLAST output 98
7.4 The BLAST recoi9nclass500(.)]TJ 0 g 0 406 . . . . . .

10 Swiss-Prot and ExPASy 151
10.1 Parsing Swiss-Prot files . . . 151
10.1.1 Parsing Swiss-Prot records . . . 151
10.1.2 Parsing the Swiss-Prot keyword and category list . . . 153
10.2 Parsing Prosite records . . . 154

11.8.1 Downloading structures from the Protein Data Bank 178
11.8.2 Downloading the entire PDB 178
11.8.3 Keeping a local copy of the PDB up to date 179

23.7 Contributing Code 330

24 Appendix: Useful stu about Python 331
24.1 What the heck is a handle? 331
24.1.1 Creating .1.1 Cre-615(34omg)-334(.1.1)-s(tring)]TJ 0 g 0 G [3758(.)-500(.)-499(.)-500(.)-500(.)-500(.)-500(

Chapter 1

Introduction

1.1 What is Biopython?

The Biopython Project is an international association of developers of freely available Python (

1.4 Frequently Asked Questions (FAQ)

1. *How do I cite Biopython in a scienti c publication?*
   Please cite our application note [1. Cock *et al.*, 2009] as the main Biopython reference. In addition, please cite any publications from the following list i[tions, articular[tions reference fr[tionsspo
   wthain Biopytn morenfrmcationen our

5. *Do you have a change-log listing what's new in each release?*
   See the le NEWS.rst included with the source code (originally called just NEWS

9. *What is wrong with my sequence comparisons?*

21. *Why doesn't my* `SeqRecord` *object have a* `letter_annotations` *attribute?*
    Per-letter-annotation support was added in Biopython 1.50.

22. *Why can't I slice my*

Chapter 2

Quick Start { What can you do with Biopython?

followed by what you would type in:

```
>>> from Bio.Seq import Seq
>>> my_seq = SeqkAGTACACTGGT"qq
>>> my_seq
Seq'AGTACACTGGT',>
>>>my_seqq

>>> my_se.alphabetq


edifd  histisdDNAdproteindtqokaby83(,>)3964atproteind  ithtadottoftAlanines,>,>andq

jecadiersd(from)468(th(t)469(Pythond)469(stringt)468(ind)469(th(t)468(methop)-28dsd)469(itq)]TJ-216.1010
>>> my_seq
Seq'AGTACACTGGT',>
>>> my_se.complementqqq
Seq'TCATGTGACCA',>
>>> my_se.reverse_complementqqq
```

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

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 nish 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 ading(to)28(an)66(w8(ou)-)27 [(ibraries)66(w8)-337(w8)-345(Biop)28(yt(ou)ill

Chapter 3

Sequence objects

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

The Seq object has a .count()

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

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

```
>>> my_seq[0::3]
Seq('GCTGTAGTAAG', IUPACUnambiguousDNA())
>>> my_seq[1::3]
Seq('AGGCATGCATCTAAG', IUPACUnambiguousDNA())
```

```
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Seq import Seq
```

In all of these operations, the alphabet property is maintained. This is very useful in case you accidentally

In the bacterial genetic code GTG is a valid start codon, and while it does *normally* encode Valine, if used as

```
--+---------+---------+---------+---------+--
G | GTT V   | GCT A   | GAT D   | GGT G   | T
G | GTC V   | GCC A   | GAC D   | GGC G   | C
G | GTA V   | GCA A   | GAA E   | GGA G   | A
G | GTG V   | GCG A   | GAG E   | GGG G   | G
--+---------+---------+---------+---------+--
```

and:

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

  |  T      |  C      |  A      |  G      |
--+---------+---------+---------+---------+--
T | TTT F   | TCT S   | TAT Y   | TGT C   | T
T | TTC F   | TCC S   | TAC Y   | TGC C   | C
T | TTA L   | TCA S   | TAA Stop| TGA W   | A
T | TTG L   | TCG S   | TAG Stop| TGG W   | G
--+---------+---------+---------+---------+--
C | CTT L   | CCT P   | CAT H   | CGT R   | T
C | CTC L   | CCC P   | CAC H   | CGC R   | C
C | CTA L   | CCA P   | CAA Q   | CGA R   | A
C | CTG L   | CCG P   | CAG Q   | CGG R   | G
--+---------+---------+---------+---------+--
A | ATT I(s)| ACT T   | AAT N   | AGT S   | T
A | ATC I(s)| ACC T   | AAC N   | AGC S   | C
A | ATA M(s)| ACA T   | AAA K   | AGA Stop| A
A | ATG M(s)| ACC52525(T)-1575(|)-525(AAA)-525(S)-1575(|)-525GAGA Stop| G
--+---------+---------+---------+----6----+--
G | GTT V   | GCT A   | GAT D   | GGT G   | T
G | GTC V   | GCC A   | GAC D   | GGC G   | C
G | GTA V   | GCA A   | GAA E   | GGA G   | A
G | ATG V   | GCG A   | GAG E   | GGG G   5 G
```

For example, you might argue that the two DNA

3.13 UnknownSeq objects

The UnknownSeq object is a subclass of the basic Seq object and its purpose is to represent a sequence where we know the length, but not the actual letters making it up. You could of course use a normal Seq object

3.14 Working with strings directly

Chapter 4

Sequence annotation objects

Chapter 3 introduced the sequence classes. Immediately \above" the Seq class is the Sequence Record or SeqRecord class, de ned in the

.annotations { A dictionary of additional information about the sequence. The keys are the name of

Working with per-letter-annotations is similar, `letter_annotations` is a dictionary like attribute which

```
>>> record.seq


 from theq-362(LOCUSq-361(line,q-369(while)-362(theq)]T/FF299.9626Tf174.41160Td[idq)]T/F899.9626Tf14.062
>>> recorddescriptionq



>>> recordletter_annotationsq



>>> recorddbxrefsq
```

```
>>> my_location.start
AfterPosition(5)
>>> print(my_location.start)
```

```
>>> 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 les de ned with joins are the union of the exons { they do not cover any introns.

4.3.3 Sequen [(.7(Seqt%crib(.71(ed)-3(Seb)71(y)-3(Sea)-3(Sefeature)-3(Seor)-3(Selo(.71(cat

4.4 Comparison

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

```
>>> from Bio.Seq import Seq
```

4.6 The format method

The `format()` method of the `SeqRecord` class gives a string containing your record formatted using one of the output le formats supported by

For this example we're going to focus in on the pim gene, YP_pPCP05. If you have a look at the GenBank
le directly you'll nd isis loo-30str0(gone,)]TJ/F29 9.9

```
>>> from Bio import SeqIO
>>> record = next(SeqIO.parse("example.fastq", "fastq"))
>>> len(record)
25
>>> print(record.seq)
CCCTTCTTGTCTTCAGCGTTTCTCC

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

Let's suppose this was Roche 454 data, and that from other information you think the TTT

For the sequence, thi [(F)8Ieshe theSeqheersthethemthe135()1(ohe)8(d.)-412(An)28(yhe)-23featurehi [(F)artheF

5.1.1 Reading Sequence Files

In general `Bio.SeqIO.parse()` is used to read in sequence files

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

Note that if you try to use `next()`

You can of course still use a for loop with a list ofSeqRecordobjects. Using a list is much more exiblethan an iterator (for example, you ca

or:

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

In general, `organism' is used for the scienti c 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 elds are identical.

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

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

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

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

In either case, the result is:

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

```
...     handle = bz2.open("ls_orchid.gbk.bz2", "rt")  # Python 3
... else:
...     handle = bz2.BZ2File("ls_orchid.gbk.bz2", "r")  # Python 2
...
>>> with handle:
```

The expected output of this example is:

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

Notice this time we have three features.

5.4.1.2 Indexing a dictionary using the SEGUID checksum

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

Once again, working with the orchids GenBank file:

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

This should give:

```
Z78533.1 JUEoWn6DPhgZ9nAyowsgtoD9TTo
Z78532.1 MN/s0q9zDoCVEEc+k/IFwCNF2pY
```

```
>>> seq_record = orchid_dict["Z78475.1"]
>>> print(seq_record.description)
```

to preserve the text exactly (e.g. GenBank or EMBL output from Biopython does not yet preserve every last bit of annotation).

Let's suppose you have download the whole of UniProt in the plain text SwissPort le format from their FTP site (ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/complete/

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

```
>>> import glob
```

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

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

or:

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

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

5.4.5 Discussion

So, which of these methods should you use and why? It depends on what you are trying to do (and how much

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 (626 Tf 50.44 [(ydas)-rformat300.

```
from Bio import SeqIO
```

```
>>> records = (rec.reverse_complement(id="rc_"+rec.id, description = "reverse complement") \
...           for recor=pR_B%3:S4îS3âP¿ÍD!rq7âF'} records = (rec.reverse_complement(id="rc_"+rec.id, des
```

Chapter 6

Multiple Sequence Alignment objects

6.1.1 Single Alignments

```
from Bio import AlignIO
alignments = list(AlignIO.parse("resampled.phy", "phylip"))
last_align = alignments[-1]
first_align = alignments[0]
```

6.1.3 Ambiguous Alignments

Many alignment file formats can explicitly store more than one alignment, and the division between each alignment is clear. However, when a general sequence file format has been used there is no such block structure. The most common such situation is when alignments have been saved in the FASTA file format.

```
>YYY
ACTACGGCAAGCACAGG
>Alpha
--ACTACGAC--TAGCTCAGG
>ZZZ
GGACTACGACAATAGCTCAGG
```

In this third example, because of the di ering lengths, this cannot be treated as a single alignment containing

6.2 Writing Alignments

We've talked about using `Bio.AlignIO.read()` and `Bio.AlignIO.parse()` for alignment input (reading les), and now we'll look at `Bio.AlignIO.write()`

Its more common to want to load an existing alignment, and save that, perhaps after some simple

existing file

Q9T0Q8_BPIKE/1-52	RA
COATB_BPI22/32-83	KA
COATB_BPM13/24-72	KA
COATB_BPZJ2/1-49	KA
Q9T0Q9_BPFD/1-49	KA
COATB_BPIF1/22-73	RA

KA
KA
KA
KA
RA

If you have to work with the original strict PHYLIP format, then you may need to compress the identifers

```
from Bio import AlignIO
alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
print(alignment.format("clustal"))
```

As described in Section 4.6, the `SeqRecord` object has a similar method using output formats supported by `Bio.SeqIO`.

Internally the `format()` method is using the `StringIO` string based handle and calling `Bio.AlignIO.write()`. You can doobbobb eT.955 Tf.s(ob) T.955 example [(b)28rnalou.s(ob)are [(b)2-316(out3(d)-2s(ob)olde5T.955 vrnalersJ 1 s(

```
>>> print(alignment[:, 6:9])
```

6.3.2 Alignments as arrays

Depending on what you are doing, it can be more useful to turn the alignment object into an array of letters { and you can do this with NumPy:

```
>>> import numpy as np
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
```

6.4.1 ClustalW

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

For the most basic usage, all you need is to have a FASTA input file, such as opuntia.fasta (available online or in the Doc/examples subdirectory of the Biopython source code). You can then tell MUSCLE to read in this FASTA file, and write the alignment to an output file:

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

Note that MUSCLE uses \-in" and \-out" but in Biopython we have to use \input" and \out" as the keyword arguments

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

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

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

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

```
>>> handle = StringIO()
>>> SeqIO.write(records, handle, "fasta")
6
>>> data = handle.getvalue()
```

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

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

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

6.4.5 EMBOSS needle and water

The EMBOSS suite includes the `water` and `needle` tools for Smith-Waterman algorithm local alignment,

Why not try running this by hand at the command prompt? You should see it does a pairwise comparison

In this example, we told EMBOSS to write the output to a le, but you *can* tell it to write the output to stdout instead (useful ian

Score=292.5

Chapter 7

BLAST

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

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

After doing this, the results are in the file `my_blast.xml` and the original handle has had all its data extracted (so we closed it). However, the `parse` function of the BLAST parser (described in 7.3) takes a file-handle-like object, so we can just open the saved file for input:

```
>>> result_handle = open("my_blast.xml")
```

Now that we've got the BLAST results back into a handle again, we are ready to do something with them, so this leads us right into the parsing section (see Section 7.3 below). You may want to jump ahead to that now ....

7.2 Running BLAST locally

7.2.1 Introduction

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

Or, you can use a for-loop:

```
>>> for blast_record in blast_records:
...     # Do something with blast_record
```

Note though that you can step through the BLAST records only once. Usually, from each BLAST record

```
length: 783
e value: 0.034
tacttgttgatattggatcgaacaaactggagaaccaacatgctcacgtcacttttagtcccttacatattcctc...
||||||||| | |||||||||||  || ||||  || || ||||||||  ||||||  |  | ||||||||  |||  ||...
tacttgttggtgttggatcgaaccaattggaagacgaatatgctcacatcacttctcattccttacatcttcttc...
```

Basically, you can do anything you want to with the info in the BLAST report once you have parsed it. This will, of course, depend on what you want to use it for, but hopefully this helps you get started on doing what you need to do!

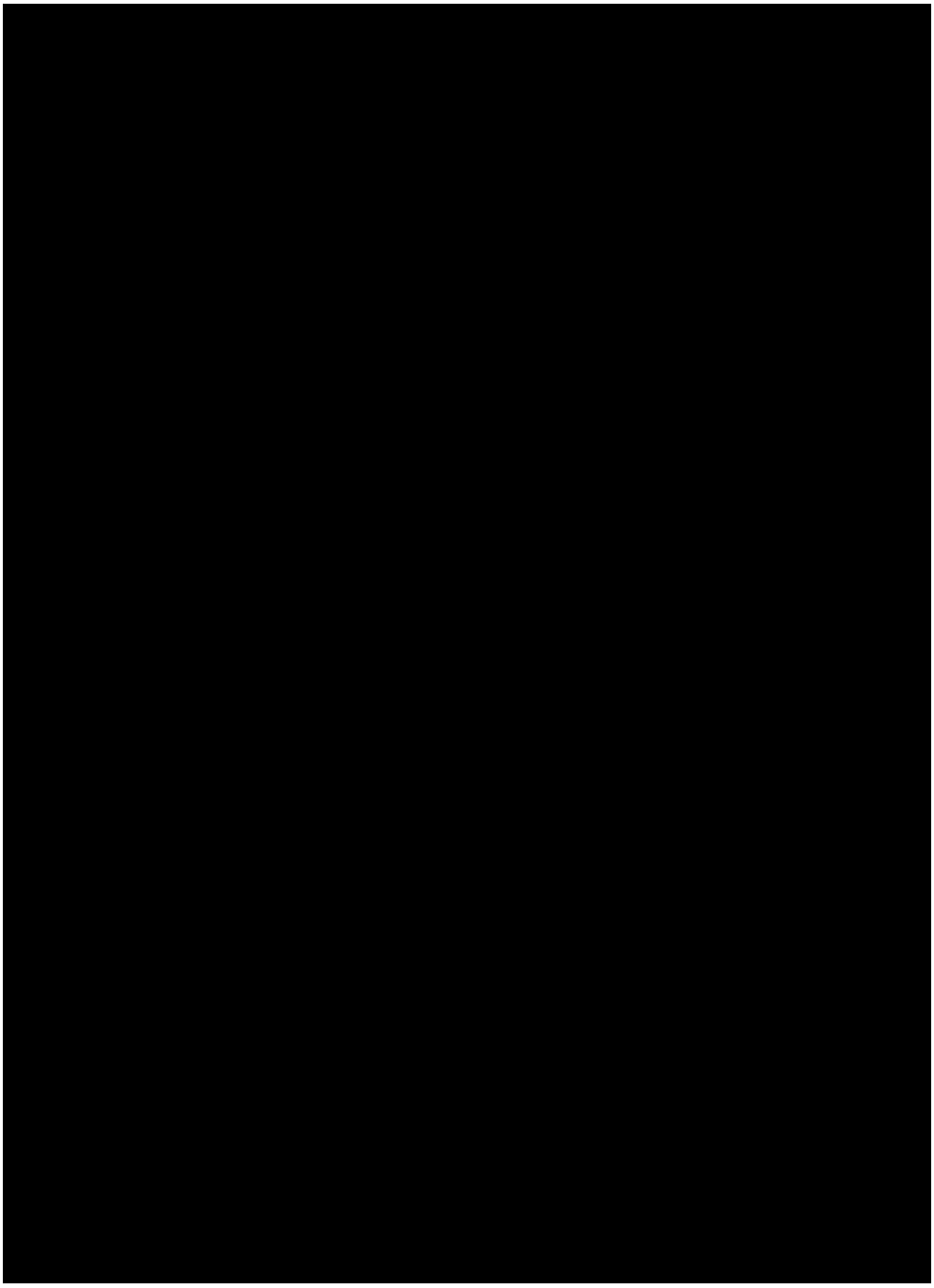


Figure 7.2: Class diagram for the PSIBlast Record class.

7.5.3 Finding a bad record somewhere in a huge plain-text BLAST file

One really ugly problem that happens to me is that I'll be parsing a huge blast file for a while, and the parser will bomb out with a ValueError. This is a serious problem, since you can't tell if the ValueError is due to a parser problem, or a problem with the BLAST. To make it even worse, you have no idea where the parse failed, so you can't just ignore the error, since this could be ignoring an important data point.

{

Chapter 8

8.1 The SearchIO object model

Now let's check our BLAT results using the same procedure as above:

Sometimes, knowing whether a hit is present is not enough; you also want to know the rank of the hit. Here, the `index`

```
      mystery_seq
 Hit: gi|301171322|ref|NR_035857.1| (86)
      Pan troglodytes microRNA mir-520c (MIR520C), microRNA
HSPs: ----  --------  ---------  ------  ---------------  ---------------------
         #   E-value  Bit score    Span      Query range              Hit range
      ----  --------  ---------  ------  ---------------  ---------------------
         0   8.9e-20     100.47      60           [1:61]                [13:73]
```

Here, we've got a similar level of detail as with the BLAST01i88(got)-297(w288(a)-a8(e'w288(a)earlier.)-429(Thd [etail)- [

8.1.3 HSP

HSP (high-scoring pair) represents region(s) in the hit sequence that contains signi cant 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 di erent

Check out the HSP documentation for a full list of these prede ned properties.

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

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

These details are format-speci c; they may not be present in other formats. To see which details are

(tica755(taiabl(e)-633(for)-633ar)-633givk)28(nr)-633(sequence)-633(searc)28(h)-633(to)-28(oe,)-708(y-28(ue)-633(houlde)-63
lly y-28(ue)-334(ma)28(y)-333(also)-334ushe

[illegible]

```
>>> blast_hspequry1

>>> 69 0 Tdequry1
61
```

```
>>> blat_hsp.hit is None
True
>>> blat_hsp.query is None
True
>>> blat_hsp.aln is None
True
```

This does not a ect 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-speci c attributes are still present as well:

```
>>> blat_hsp.score          # PSL score
61
>>> blat_hsp.mismatch_num   # the mismatch column
0
```

```
>>> blat_hsp2.hit_range         # hit start and end coordinates of the entire HSP
(54233104, 54264463)
>>> blat_hsp2.hit_range_all     # hit start and end coordinates of each fragment
[04, 54264463) 54264463]1
>>> blat_hsp2.hitspan>          # hit pan> of the entire HSP

fragment      >>> blat_hsp2.hitspane_all

>>> blat_hsp2.hitintert_ranges # start and end coordinates of the hit equencet
[04, 224, 54264203]1
>>> blat_hsp2.hitintert(panes)15725(#)-525(pan>)-525(of)-525interveningf the hit equencet
```

```
Query range: [0:61] (1)
  Hit range: [0:61] (1)
  Fragments: 1 (61 columns)
     Query - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
       Hit - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
```

At this level, the BLAT fragment looks quite similar to the BLAST fragment, save for the query and hit sequences which are not present:

```
>>> blat_qresult = SearchIO.read('my_blat.psl', 'blat-psl')
>>> blat_frag = blat_qresult[0][0][0]    # first hit, first hsp, first fragment
>>> print(blat_frag)
      Query: mystery_seq <unknown description>
        Hit: chr19 <unknown description>
Query range: [0:61] (1)
  Hit range: [54204480:54204541] (1)
  Fragments: 1 (? columns)
```

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

```
>>> blast_frag.query_start      # query start coordinate
0
>>> blast_frag.hit_strand       # hit sequence strand
1
>>> blast_frag.hit              # hit sequence, as a SeqRecord object
```

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`

need to access only a few of the queries. This is because parse will parse all queries it sees before it fetches your query of interest.

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

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

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

Or also with the format-specific keyword argument:

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

Or with the key_function 80617 | ref | NF8 925('of)4(their)]TJ/F29 9.91.206f 76.768 0 Td [(Bio.SeqIO)]TJ/F8 9.6260 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:

Chapter 9

Accessing NCBI's Entrez databases

Entrez (http://www.ncbi.nlm.nih.gov/Entrez) is a data retrieval system that provides users access to

f(er)a05.pbgtanquetrisOAccesmanus,Biopn

us,Entrezogrammsing also, Oconsistsing

The Entrez Programming Utilities can also generate output in other formats, such as the Fasta or

9.2 EInfo: Obtaining information about the Entrez databases

EInfo provides eld 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 's <amata33(of)ccessi5(ailInfo)hrougha33(of)-375(use)05(coun)31(tnfoutil [(

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]`

9.4 EPost: Uploading a list of identi ers

/product="maturase K"
/protein_id="ACC99456.1"
/translation="IFYEPVEIFGYDNKSSLVLVKRLITRMYQQNFLISSVNDSNQKG

```
>>> from Bio import SeqIO
>>> handle = Entrez.efetch(db="nucleotide", id="EU490707", rettype="gb", retmode="text")
>>> record = SeqIO.read(handle, "genbank")
>>> handle.close()
>>> print(record.id)
EU490707.1
>>> print(record.name)
EU490707
>>> print(record.description)
Selenipedium aequinoctia5(retmodaturas5(retmoK(retmotdatKn)retmo"gele,)-525parctia,)-525cds;,)-525chlor
>>> printelet(recordfeaturesnd)

>>> print(rprt(recordseqnd)
```

```
>>> 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 interested3(w)28(tere3(w)28 nd3(w)28out)-303(iterf)-303(a)-303(pap)
articularllytow(tems.)4045wnin

```
        <MenuName>PubMed</MenuName>
        <Description>PubMed bibliographic record</Description>
        <Count>20161961</Count>
        <LastUpdate>2010/09/10 04:52</LastUpdate>
        <FieldList>
                <Field>
...
                </Field>
        </FieldList>
        <DocsumList>
                <Docsum>
                        <DsName>PubDate</DsName>
                        <DsType>4</DsType>
                        <DsTypeName>string</DsTypeName>
                </Docsum>
                <Docsum>
                        <DsName>EPubDate</DsName>
...
        </DbInfo>
</eInfoResult>
```

‹DocsumLilt>
In this le, for some reasr3 -41.Ti65 ype>4‹/DsType>

9.12.1 Parsing Medline records

You can nd the Medline parser in `Bio.Medline`. Suppose we want to parse the le `pubmed_result1.txt`,
cir6in1(ars27pp)Medl279 Medl27pp-375(rTd [26 Td [(Y)8327pp-341(27pp-342(7ppthisdl279)-342(7ppi41(27ppTd p[(ciyth

PROTSIM ORG=10090; PROTGI=6754794; PROTID=NP_035004.1; PCT=76.55; ALN=288

9.13 Using a proxy

NOTE - We've just done a separate search and fetch here, the NCBI much prefer you to take advantage

```
>>> for row in record["eGQueryResult"]:
...     if row["DbName"]=="nuccore":
...         print(row["Count"])
4457
```

```
 'GBSeq_create-date', 'GBSeq_other-seqids', 'GBSeq_division',
 'GBSeq_taxonomy', 'GBSeq_references', 'GBSeq_update-date',
 'GBSeq_organism', 'GBSeq_locus', 'GBSeq_strandedness']

>>> print(records[0]["GBSeq_primary-accession"])
DQ110336

>>> print(records[0]["GBSeq_other-seqids"])
['gb|DQ110336.1|', 'gi|187237168']

>>> print(records[0]["GBSeq_definition"])
Cypripedium calceolus voucher Davis 03-03 A maturase (matR) gene, partial cds;
mitochondrial

>>> print(records[0]["GBSeq_organism"])
Cypripedium calceolus
```

You could use this to quickly set up searches { but for heavy usage, see Section 9.15.

9.14.3 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 http://www.ncbi.nlm.nih.gov/.

In this example we'll show how to query the NCBI databases,to retrieve the records from the query, and then parse them using `Bio.SeqIO` - something touched on in Section 5.3.1. For simplicity, this example *does not* take advantage of the WebEnv history feature { see Section 9.15 for this.

```
>>> gi_str = ",".join(gi_list)
>>> handle = Entrez.efetch(db="nuccore", id=gi_str, rettype="gb", retmode="text")
```

If you want to look at the raw GenBank files, you can read from this handle and print out the result:

```
>>> text = handle.read()
>>> print(text)
LOCUS       AY851612                 892 bp    DNA     linear   PLN 10-APR-2007
DEFINITION  Opuntia subulata rpl16 gene, intron; chloroplast.
ACCESSION   AY851612
VERSION     AY851612.1  GI:57240072
KEYWORDS    .
SOURCE      chloroplast Austrocylindropuntia subulata
  ORGANISM  Austrocylindropuntia subulata
            Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;
```

9.14.4 Finding the lineage of an organism

Staying with a plant example, let's now nd the lineage of the Cypripedioideae orchid family. First, we search the Taxonomy database for Cypripedioideae, which yields exactly one NCBI taxonomy identi er:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"     # Always tell NCBI who you are
>>> handle = Entrez.esearch(db="Taxonomy", term="Cypripedioideae")
>>> record = Entrez.read(handle)
>>> record["IdList"]
['158330']
>>> record["IdList"][0]
'158330'
```

Now, we use efetch to download this entry il3=9tl Oos9 dparrch NCBI taxonomy ide-92.043ti er:

```
>>> from Bio import Entrez
>>> Entrez.email = "history.user@example.com"
>>> search_handle = Entrez.esearch(db="nucleotide",term="Opuntia[orgn] and rpl16",
...                                usehistory="y", idtype="acc")
>>> search_results = Entrez.read(search_handle)
>>> search_handle.close()
```

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

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

```
            if 500 <= err.code <= 599:
                print("Received error from server %s" % err)
                print("Attempt %i of 3" % attempt)
                time.sleep(15)
            else:
                raise
    data = fetch_handle.read()
    fetch_handle.close()
    out_handle.write(data)
out_handle.close()
```

For illustrative purposes, this example downloaded the FASTA records in batches of three. Unless you are downloading genomes or chromosomes, you would normally pick a larger batch size.

9.15.2 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 le in MedLine format:

```
from Bio import Entrez
import time
try:
    from urllib.error import HTTPError  # for Python 3
exd(urllib.errIort)-ror
```

```
            else:
                raise
    data = fetch_handle.read()
    fetch_handle.close()
    out_handle.write(data)
out_handle.close()
```

Open a Swiss-Prot file over the internet from the ExPASy database (see section

As of June 2009, the full Swiss-Prot database downloaded from ExPASy contained 468851 Swiss-Prot records. One concise way to build up a list of the record descriptions is with a list comprehension:

```
>>> from Bio import SwissProt
```

```
HI   Ligand: Metal-binding; 2Fe-2S.
CA   Ligand.
//
ID   3D-structure.
AC   KW-0002
DE   Protein, or part of a protein, whose three-dimensional structure has
DE   been resolved experimentally (for example by X-ray crystallography or
DE   NMR spectroscopy) and whose coordinates are available in the PDB
DE   database. Can alsus525(theusresolved5((for)-5oreticensionalmodelsucture.)]TJ0-11.955Td[(HI)Technic
//
ID2Fe-2S.
```

'PDOC00005'2073To read exactly one Prosite from the handle, you can use theread>>> record = Prosite.read(handle)found.In the

```
[['P11151', 'LIPL_BOVIN'], ['P11153', 'LIPL_CAVPO'], ['P11602', 'LIPL_CHICK'],
['P55031', 'LIPL_FELCA'], ['P06858', 'LIPL_HUMAN'], ['P11152', 'LIPL_MOUSE'],
['O46647', 'LIPL_MUSVI'], ['P49060', 'LIPL_PAPAN'], ['P49923', 'LIPL_PIG'],
['Q06000', 'LIPL_RAT'], ['Q29524', 'LIPL_SHEEP']]
```

The `read` function raises a ValueError if no Enzyme record is found, and also if more than one Enzyme record is found.

The full set of Enzyme records can be downloaded as a single file (`enzyme.dat`) from the ExPASy FTP site, containing 4877 records (release of 3 March 2009). To parse such a file containing multiple Enzyme records, use the `parse` function in `Bio.ExPASy.Enzyme` to obtain an iterator:

```
>>> from Bio.ExPASy import Enzyme
u[()=(u[()opends)-"0Td[(en"0Td_PIG'],)]TJ0-11.956Td[()77 r[()=(u[()rt)-52.0Tdds
```

t375(th)-3375([-949(ExP)875(serv)314er)HEEP']]fun22(me)-339(re22(cin)-3237an)-339(b)237(e)downl22(51)-33922(th)-33237 [-349(ExP)8237wservenzyme(the)]TJ 0 0 1 rg 00 g 0 G /F29 9.'],

```
>>> 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 exis2(y)282455ins

```
SwissProt.read(handle)
```

```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prosite_raw('PS00001')
>>> text = handle.read()
>>> print(text)
```

To retrieve a Prosite record and parse it into a `Bio.Prosite.Record` object, use

```
>>> from Bio import ExPASy
>>> from Bio import Prosite
>>> handle = ExPASy.get_prosite_raw('PS00001')
>>> record = Prosite.read(handle)
```

```
handle = ScanProsite.scan(seq=sequence)
```

If you want to obtain the search results in raw XML format. Instead, use

Chapter 11

```
The available keys are name, head, deposition_date, release_date, structure_method
```

11.2 Structure representation

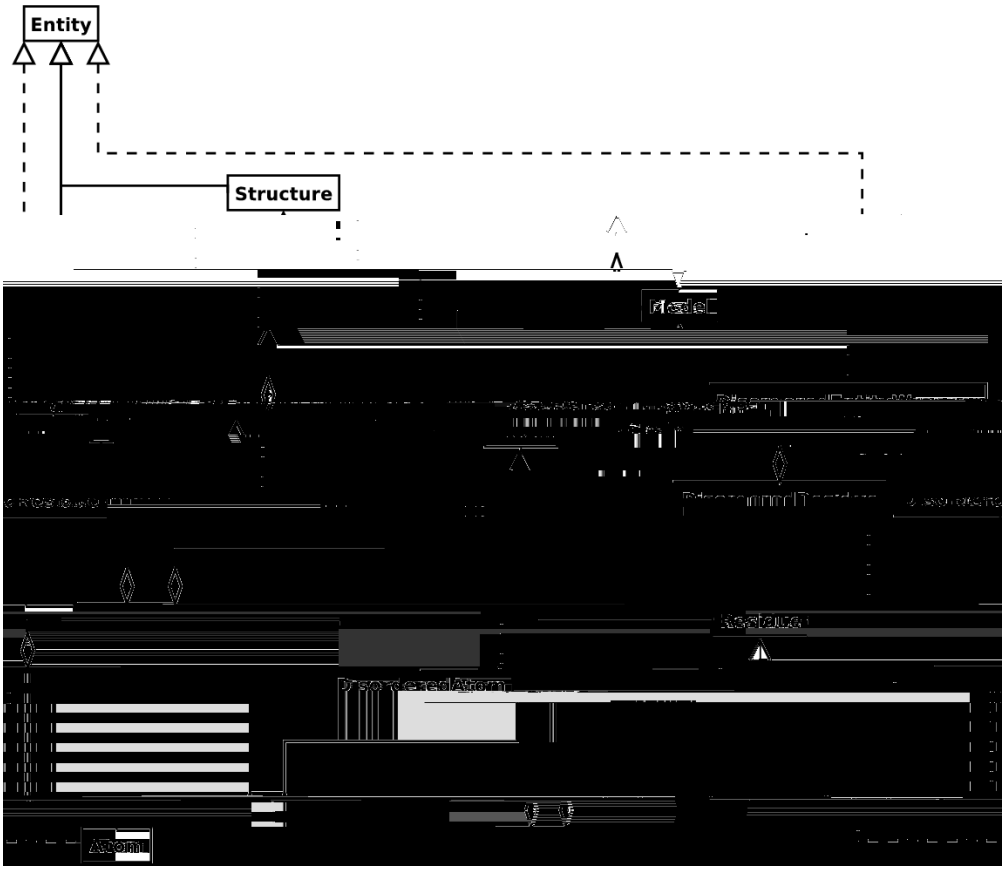
The overall layout of a `Structure` object follows the so-called SMCRA (Structure/Model/Chain/Residue/Atom) architecture:

A structure consists of models

A model consists of chains

A chain consists of residues

A residue consists of atoms


Entity
Structure

The Model with id 0

The Chain with id "A"

The Residue with id (" ", 10, "A").

11.2.4 Residue

A residue id is a tuple with three elements:

The hetero- eld (het eld): this is

{ 'W' in the case of a water molecule;

{

11.2.5 Atom

```
# find rotation matrix that rotates n
# -120 degrees along the ca-c vector
```

11.4.3 Other hetero residues

The het eld string for other hetero residues starts with \H_" followed by the residue name. A glucose molecule e.g. with residue name \GLC" would have het eld \H_GLC". Its residue id could e.g. be (\H_GLC", 1, \

Code	Secondary structure
H	-helix
B	Isolated -bridge residue
E	Strand

11.7.1 Examples

The PDBParser/Structure class was tested on about 800 structures (each belonging to a unique SCOP superfamily). This takes about 20 minutes, or on average 1.5 seconds per structure. Parsing the structure of the large ribosomal subunit (1FKK), which contains about 64000 atoms, takes 10 seconds on a 1000 MHz PC.

Three exceptions were generated in cases where an unambiguous data structure could not be built. In all three cases, the likely cause is an error in the PDB le that should be corrected. Generating an exception in these cases is much better than running the chance of incorrectly describing the structure in a data structure.

11.7.1.1 Duplicate residues

One structure contains two amino acid residues in one chain with the same sequence identi er (resseq 3) and icode. Upon inspection it was found that this chain contains the residues Thr A3, ..., Gly A202, Leu A3, Glu A204. Clearly, Leu A3 should be Leu A203. A couple of similar situations exist for structure 1FFK (which e.g. contains Gly B64, Met B65, Glu B65, Thr B67, i.e. residue Glu B65 should be Glu B66).

11.7.1.2 Duplicate atoms

Structure 1EJG contains a Ser/Pro point mutation in chain A at position 22. In turn, Ser 22 contains some disordered atoms. As expected, all atoms belonging to Ser 22 have a non-blank altloc speci er (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 le: the N atom should be present in both the Ser and the Pro residue, in both cases associated with a suitable altloc identi er.

11.7.2 Automatic correction

Some errors are quite common and can be easilyare-1(oist)-306(for)-30uch intierres-]TJ 0 -11.955 Td [(thtion) cTese aases a

11.7.121 D ,bl-1(ink)-338(altlo)-232c aor a aisordered alomSNormll ach isordered tom of the same atom. This is automaticmll intierresed in the right way

11.7.122 Brok chains

The insertion code (icode).

The het eld string (\W" for waters and \H_" followed by the residue name for other hetero residues)

The residue names of the residues in the case of point mutations (to store the Residue objects in a DisorderedResidue object).

If this does not lead to a unique id something is quite likely wrong, and an exception is generated.

11.7.3.2 Duplicate atoms

All atoms in a residue should have a unique id. This id is generated based on:

The atom name (without spaces, or with spaces if a problem arises).

The altloc speci er.

If this does not lead to a unique id something is quite likely wrong, and an exception is generated.

11.8 Accessing the Protein Data Bank

11.8.1 Downloading structures from the Protein Data Bank

Structures can be downloaded from the PDB (Protein Data Bank) by using the `retrieve_pdb_file` method on a `PDBList`

11.9.4 Who's using Bio.PDB?

```
        ('Other1', [(1, 1),  (4, 3), (200, 200)],
    ]
]
```

So we have two populations, the rst with two individuals, the second with only one. The rst individual of the rst 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 indi2s2l014rst indi2s2l014rst indi2s2l014rst indi2s2l014rst indi2s23657

Chapter 13

Phylogenetics with Bio.Phylo

```
            Clade(name='C')
            Clade(name='D')
        Clade()
            Clade(name='E')
            Clade(name='F')
            Clade(name='G')
```

The

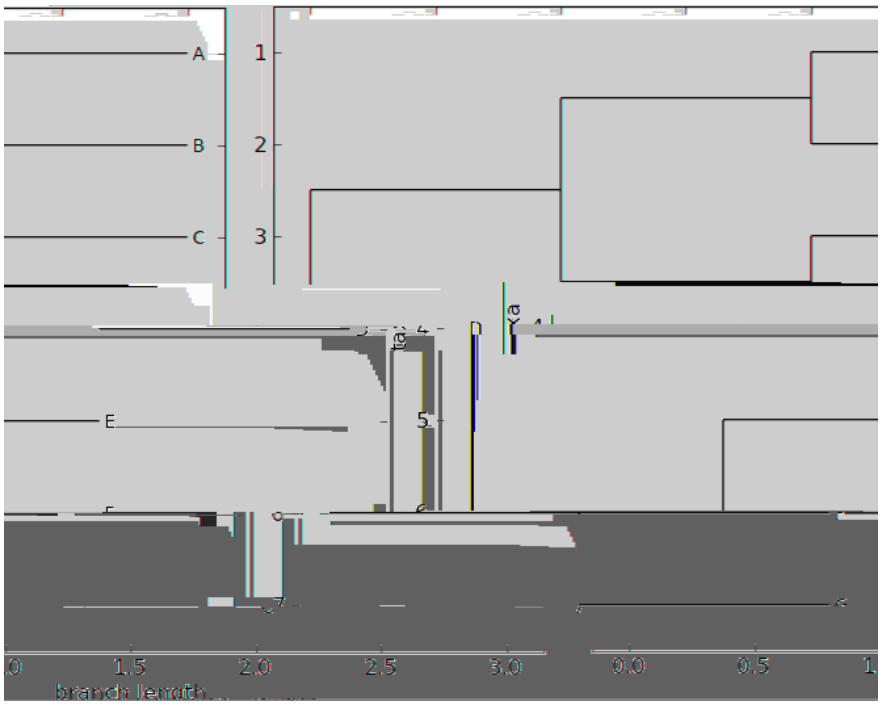


Figure 13.1: A rooted tree drawn with `Phylo.draw`.

Note that the le formats Newick and Nexus don't support branch colors or widths, so if you use these

Figure 13.3: A simple rooted tree plotted with the `draw` function.

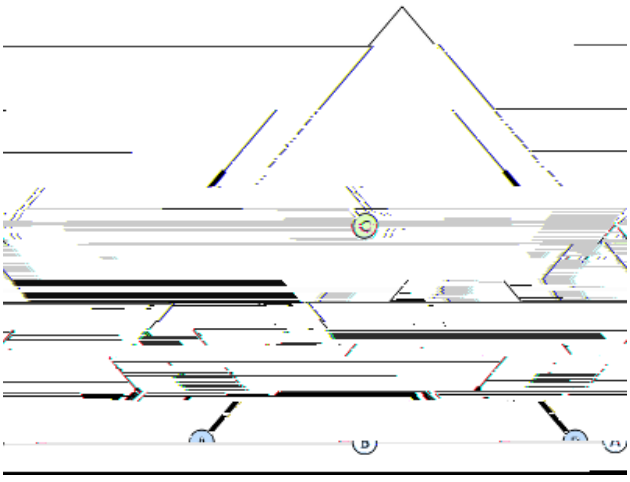


Figure 13.4: A simple rooted tree drawn with `draw_graphviz`, using `dot` for node layout.

Figure 13.5: A rooted tree drawn wle
drawn

Figure 13.7: A larger tree, using `neato`

Note that branch lengths are not displayed accurately, because Graphviz ignores them when creating the node layouts. The branch lengths are retained when exporting a tree as a NetworkX graph object (`to_networkx`), however.

See the Phylo page on the Biopython wiki (

Since oating-point arithmetic can produce some strange behavior, we don't support matching

13.4.2 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

`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

13.6 PAML integration

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 13.4.4).

Chapter 14

Sequence motif analysis using Bio.motifs

then we can create a Motif object as follows:

```
>>> m = motifs.create(instances)
```

The instances are saved in an attribute `m.instances`, which is essentially a Python list with some added functionality, as described b descr]Bed, as descre0rs:

```
>>> m.alphabet
IUPACUnambiguousDNA()
>>> m.alphabet.letters
'GATC'
>>> sorted(m.alphabet.letters)
['A', 'C', 'G', 'T']
>>> m.counts['A',:]
(3, 7, 0, 2, 1)
>>> m.counts[0,:]
(3, 7, 0, 2, 1)
```

The motif has an associated consensus sequence, de ned as the sequence of letters along the positions of the motif for which the largest value in the corresponding columns of the `.counts` matrix is obtained:

```
>>> m.consensus
Seq('TACGC', IUPACUnambiguousDNA())
```

as well as an an84ers1(nsensus)-383(sequence,40333(corresp)-28(onding)-38toas)-383(the)-38smallrgest val(as)-384(in)-383(t

14.2 Reading motifs

```
AACGTGacagccctcc
>MA0004 ARNT 19
AACGTGcacatcgtcc
>MA0004 ARNT 20
aggaatCGCGTGc
```

The parts of the sequence in capital letter.are334(of)-333(themotifquence)-sta(ses334(of)-at334(ofw)27(ere334(offound334(of

```
>>> from Bio.motifs.jaspar.db import JASPAR5
>>>
>>> JASPAR_DB_HOST = <hostname>
>>> JASPAR_DB_NAME = <db_name>
>>> JASPAR_DB_USER = <user>
>>> JASPAR_DB_PASS = <passord>
>>>
>>> jdb = JASPAR5(
...     host=JASPAR_DB_HOST,
...     name=JASPAR_DB_NAME,
...     user=JASPAR_DB_USER,
...     password=JASPAR_DB_PASS
```

```
>>> motif.pseudocounts = motifs.jaspar.calculate_pseudocounts(motif)
```

```
********************************************************************************
MEME - Motif discovery tool
********************************************************************************
MEME version 3.0 (Release date: 2004/08/18 09:07:01)
...
```

Further down, the input set of training sequences is recapitulated:

```
********************************************************************************
TRAINING SET
********************************************************************************
DATAFILE= INO_up800.s
ALPHABET= ACGT
Sequence name            Weight Length  Sequence name            Weight Length
-------------            ------ ------  -------------            ------ ------
```

```
12
>>> pvalue = motif.instances[0].pvalue
>>> print("%5.3g" % pvalue)
1.85e-08
```

To parse a TRANSFAC file, use

```
>>> with open("transfac.dat") as handle:
...     record = motifs.parse(handle, "TRANSFAC")
...
```

The overall version number, if available, is stored as

Table 14.2: Fields used to store references in TRANSFAC files

RN	Reference number
RA	Reference authors
RL	Reference data
RT	Reference title
RX	PubMed ID

07	46	0	0	0	A
08	1	0	0	45	T

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
<BLANKLINE>

```
>>> for pos, seq in r.instances.search(test_seq):
...     print("%i %s" % (pos, seq))
...
6 GCATT
20 GCATT
```

```
>>> distribution = pssm.distribution(background=background, precision=10**4)
```

The distribution object can be used to determine a number of di erent thresholds. We can specify the requested false-positive rate (probability of \ nding" a motif instance in background generated sequence):

```
>>> threshold = distribution.threshold_fpr(0.01)
>>> print("%5.3f" % threshold)
4.009
```

or the false-negative rate (probability of \not nding" an instance generated from the motif):

```
>>> threshold = distribution.threshold_fnr(0.1)
>>> print("%5.3f" % threshold)
-0.510
```

```
obob>>> threshold = distribution.threshold_fnralw0J0-11.955Td[(>>>)-525(print("%5.3f")-525(%)-525(thresl
```

```
        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
<BLANKLINE>
>>> 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
<BLANKLINE>

>>> print(motif.pss=t1i2wm)
```

```
>>> 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
<BLANKLINE>
```

You can also set the `.pseudocounts` to a dictionary over the four nucleotides if you want to use di erent pseudocounts for them. Setting `motif.pseudocounts` to `None` resets it to its default value of zero.

The position-speci c scoring matrix depends on the background distribution, which is uniform by default:

```
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.background[letter]))
...
A: 0.25
C: 0.25
G: 0.25
T: 0.25
```

Again, if you modify the background distribution, the position-speci c scoring matrix is recalculated:

```
>>> motif.background = {'A'youououou
>>> print(motif.pssm)
        0      1      2      3      4      5
A:      1782  -1.92  -1.92  -1.92  -1.92
C:
G:  -168:  -1.62  -168:   1.62  -168:   1.62
T:  -1.92  -1.92  -1.92  -1.92   1852  -1.92
<BLANKLINE>
```

Setting the background to uniform distribi

```
>>> motif.background = None
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.background[letter]))
...
A: 0.25
C: 0.25
G: 0.25
T: 0.25
```

motif.background unerpreateo thoon-2(tren)28(t:)]TJ 0 g 0 G 0 g 0 G /F29 9.9626 Tf -31.340 -19.541 Td [(>>>)-525(motif.ba

```
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.background[letter]))
...
A: 0105
C: 0405
G: 0405
 0105
```

tato youcsn(o)28wo tto of(tto)-933PSSMo scodes over tto backgrounds(whic)78(h)-92(ito)-933(w)28(s5)]TJ 0 -11.955

```
>>> m_reb1.pseudocounts = {'A':0.6, 'C': 0.4, 'G': 0.4, 'T': 0.6}
>>> m_reb1.background = {'A':0.3,'C':0.2,'G':0.2,'T':0.3}
>>> pssm_reb1 = m_reb1.pssm
>>> print(pssm_reb1)
        0      1      2      3      4      5      6      7      8
A:   0.00  -5.67  -5.67   1.72  -5.67  -5.67  -5.67  -5.67  -0.97
C:  -0.97  -5.67  -5.67  -5.67   2.30   2.30   2.30  -5.67  -0.41
G:   1.30  -5.67  -5.67  -5.67  -5.67  -5.67  -5.67   1.57   1.44
T:  -1.53   1.72   1.72  -5.67  -5.67  -5.67  -5.67   0.41  -0.97
<BLANKLINE>
```

```
.version
.command
```

The motifs returned by the MEME Parser can be treated exactly like regular Motif objects (with in-

linear congruential generators, two (integer) seeds are needed for initialization, for which we use the system-supplied random number generator `rand` (in the C standard library). We initialize this generator by calling `srand` with the epoch time in seconds, and use the first two random numbers generated by `rand` as seeds for the uniform random number generator in `Bio.Cluster`.

15.1 Distance functions

In order to cluster items into groups based on their similarity, we should first define what exactly we mean by *similar*. `Bio.Cluster` provides eight distance functions, indicated by a single character, to measure similarity, or conversely, distance:

`'e'`: Euclidean distance;

`'b'`: City-block distance.

`'c'`: Pearson correlation coefficient;

`'a'`: Absolute value of the Pearson correlation coefficient;

`'u'`

where

$$\sigma_x^{(0)} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2},$$

$$\sigma_y^{(0)} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} y_i^2}.$$

```
>>> from Bio.Cluster import clustercentroids
>>> cdata, cmask = clustercentroids(data)
```

where the following arguments are de ned:

`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==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, then 5r[ng

```
7 mask     None          5051c]ng  ult:4lusteridtranspose8 1.9626 Array of 16 follo d(u)=4 (s)=1 1.9626 T 1 49.234 0 [(8 ]49 data
```

`index1` (default: `0`)

transpose

- as a list containing the rows of the left-lower part of the distance matrix:

```
distance = [array([]|,
            array([1.1]),
            array([2.3, 4.5])
           ]
```

These three expressions correspond to the same distance matrix.

`nclusters` (default: `2nclusters`

In pairwise average-linkage clustering, the distance between two nodes is de ned 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 de ned as the distance

```
>>> node.right = 2
>>> node.distance = 0.73
>>> node
(6, 2): 0.73
```

An error is raised if

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

- { `method=='m'`: pairwise maximum- (or complete-) linkage clustering
- { `method=='c'`: pairwise centroid-linkage clustering
- { `method=='a'`: pairwise average-linkage clustering

`dist`

15.5 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 [31].

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 considrocol96(Maps)f3(ecto3(the)-333)-333(is)-334(giv)28(en)-333(b)28(y)]T
f the iteo step:

$$= \text{init} \quad 1 \quad \frac{i}{\overline{n}} \quad ;$$

init isalue aseci edy *i* isumber and *n* is

`transpose` (default:

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

gweight
The weights that are to be used to calculate the distance in expression pro le between genes. If not present in the data le, gweight is set to None.

gorder
The preferred order in which genes should be stored in an output le. If not present in the data le, gorder

`transpose` (default:

{ `method=='a'`: pairwise average-linkage clustering

`dist` (default: `'e'`, Euclidean distance)
De nes the distance function to be used f 24626 Tf 41.234 0 Td [433(Euclidean)-0 rg an 8 9.96 g 0 G /F29 9.9626 Tf

`transpose` (default: `0`)
Determines if rows (`transpose` is `0`) or columns (`transpose` is `1`

15.8 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 [22, Hihara *et al.*, 2001].

```
>>> 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=251
>>> record.save("cya222result"222r25(genet22e,)-525222rtree)cyano_result.gt_K_G2_A233(will)-334(create)-3.25452
```

The logistic regression model gives us appropriate values for the parameters $_0$, $_1$, $_2$ using two sets of example genes:

OP: Adjacent genes, on the same strand of DNA, known to belong to the same operon;

NOP: Adjacent genes, on the same strand of DNA, known to belong to di erent operons.

In the logistic regression model, the probability of belonging to a class depends on the score via the logistic function. For the two classes OP and NOP, we can write this as

$$\Pr(\text{OP}|x_1;x_2) = \frac{\exp(\ _0 + \ _1x_1 + \ _2x_2)}{1 + \exp(\ _0 + \ _1x_1 + \ _2x_2)} \tag{16.2}$$

$$\Pr(\text{NOP}|x_1;x_2) = \frac{1}{1 + \exp(\ _0 + \ _1x_1 + \ _2x_2)} \tag{16.3}$$

Using a set of gene pairs for which it is known whether they belong to the same operon (class OP) or to

2, $_1$, $_1$

```
[85, -193.94],
[16, -182.71],
[15, -180.41],
[-26, -181.73],
[58, -259.87],
[126, -414.53],
[191, -249.57],
[113, -265.28],
[145, -312.99],
[154, -213.83],
[147, -380.85],
[93, -291.13]]
```

```
Iteration: 2 Log-likelihood function: -5.76877209868
Iteration: 3 Log-likelihood function: -5.11362294338
```

0, corresponding to class OP and class NOP, respectively. For example, let's consider the gene pairs *yxcE*, *yxcD* and *yxiB*, *yxiA*:

Table 16.2: Adjacent gene pairs of unknown operon status.

showing that the prediction is correct for all but one of the gene pairs. A more reliable estimate of the prediction accuracy can be found from a leave-one-out analysis, in whi72hout thnb8(delout)-330(in)-3coralcultimound from

In Biopython, the *k*-nearest neighbors method is available in `Bio.kNN`. To illustrate the use of the *k*-nearest neighbor method in Biopython, we will use the same operon data set as in section 16.1.

16.2.2 Initializing a *k*-nearest neighbors model

Using the data in Table 16.1, we create and initialize a *k* 16.1

```
...
>>> x = [6, -173.143442352]
>>> print("yxcE, yxcD:", kNN.classify(model, x, weight_fn = weight))
yxcE, yxcD: 1
```

```
print("True:", ys[i], "Predicted:", kNN.classify(model, xs[i]))
```

Chapter 17

Graphics including GenomeDiagram

The `Bio.Graphics` module depends on the third party Python library ReportLab. Although focused on

Figure 17.1: Simple linear diagram for *Yersinia pestis biovar Microtus* plasmid 2

17.1.4 A bottom up example

Now let's produce exactly the same figures, but using the bottom up approach. This means we create the different objects directly (and this can be done in almost any order) and then combine them.

```
from reportlab.lib import colors
from reportlab.lib.units import cm
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO
record = SeqIO.read("NC_005816.gb", "genbank")

#Create the feature set and its feature objects,
gd_feature_set = GenomeDiagram.FeatureSet()
for feature in record.features:
    if feature.type != "gene":
        #Exclude this feature
        continue
    if len(gd_feature_set) % 2 == 0:
        color = colors.blue
    else:
        color = colors.lightblue
    gd_feature_set.add_feature(feature, color=color, label=True)
```

```
gds_features = gdt_features.new_set()

#Add three features to show the strand options,
feature = SeqFeature(FeatureLocation(25, 125), strand=+1)
feature = SeqFeature(FeatureLocati5oorward"2w'setU), Nonerand=+1)
gds_features.add_feature(feature, name="Forward", label=Tn41)
gds_features.add_feature(feature, label=Tn41)
feature = SeqFeature(FeatureLocati7on(2537-525(125),)-525(-trand=+1))]TJ0-11.955Td[(gds_features.add_fea
```

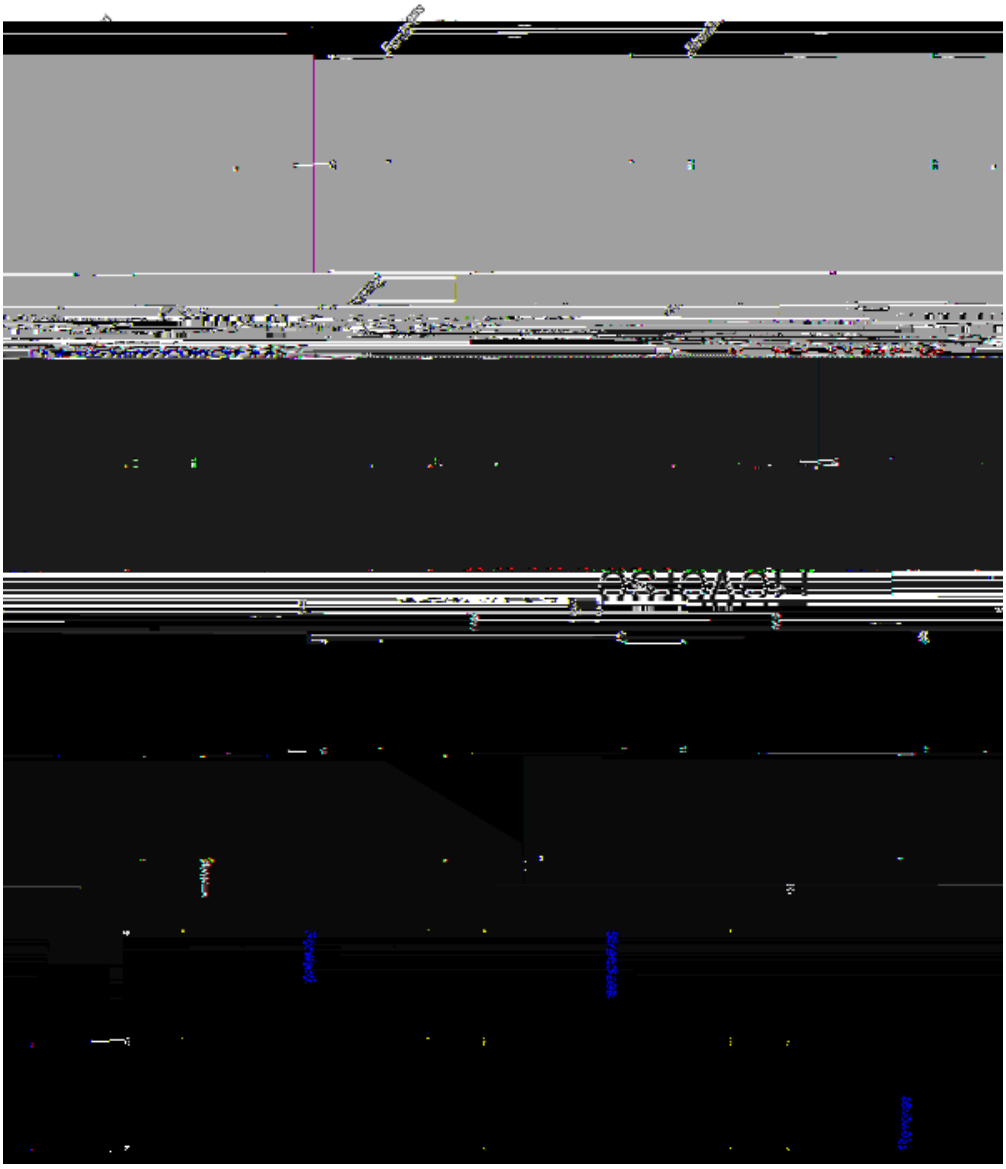


Figure 17.3: Simple GenomeDiagram showing label options. The top plot in pale green shows the default label settings (see Section 17.1.5) while the rest show variations in the label size, position and orientation (see Section 17.1.6).

17.1.7 Feature sigils

The examples above have all just used the default sigil for the feature, a plain box, which was all that was

Figure 17.4: Simple GenomeDiagram showing di erent sigils (see Section 17.1.7)

```
gd_feature_set.add_feature(feature, sigil="BIGARROW")
```

All the shaft and arrow head options shown above for the sigil.Tf ARROW.Tf 251.4470Td[=.13470Td[(sigir)-300cawne for tp

```
                start=0, end=len(record))
gd_diagram.write("plasmid_linear_nice.pdf", "PDF")
gd_diagram.write("plasmid_linear_nice.eps", "EPS")
gd_diagram.write("plasmid_linear_nice.svg", "SVG")
```

Figure 17.8: Circular diagram for

You can download these using Entrez if you like, see Section

```
        i+=1

gd_diagram.draw(format="linear", pagesize='A4', fragments=1,
                start=0, end=max_len)
gd_diagram.write(name + ".pdf", "PDF")
gd_diagram.write(name + ".eps", "EPS")
gd_diagram.write(name + ".svg", "SVG")
```

The expected output is shown in Figure 17.9. I did wonder why in the original manuscript there were no

Figure 17.9: 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 17.1.10).

red or orange genes marked in the bottom phage. Another important point is here the phage are shown with di erent lengths - this is because they are all drawn to the same scale (they *are* di erent lengths).

Continuing the example from the previous section inspired by Figure 6 from Proux *et al.* 2002 [5], we

```
    (30, "orf53", "lin2567"),
    (28, "orf54", "lin2566"),
]
```

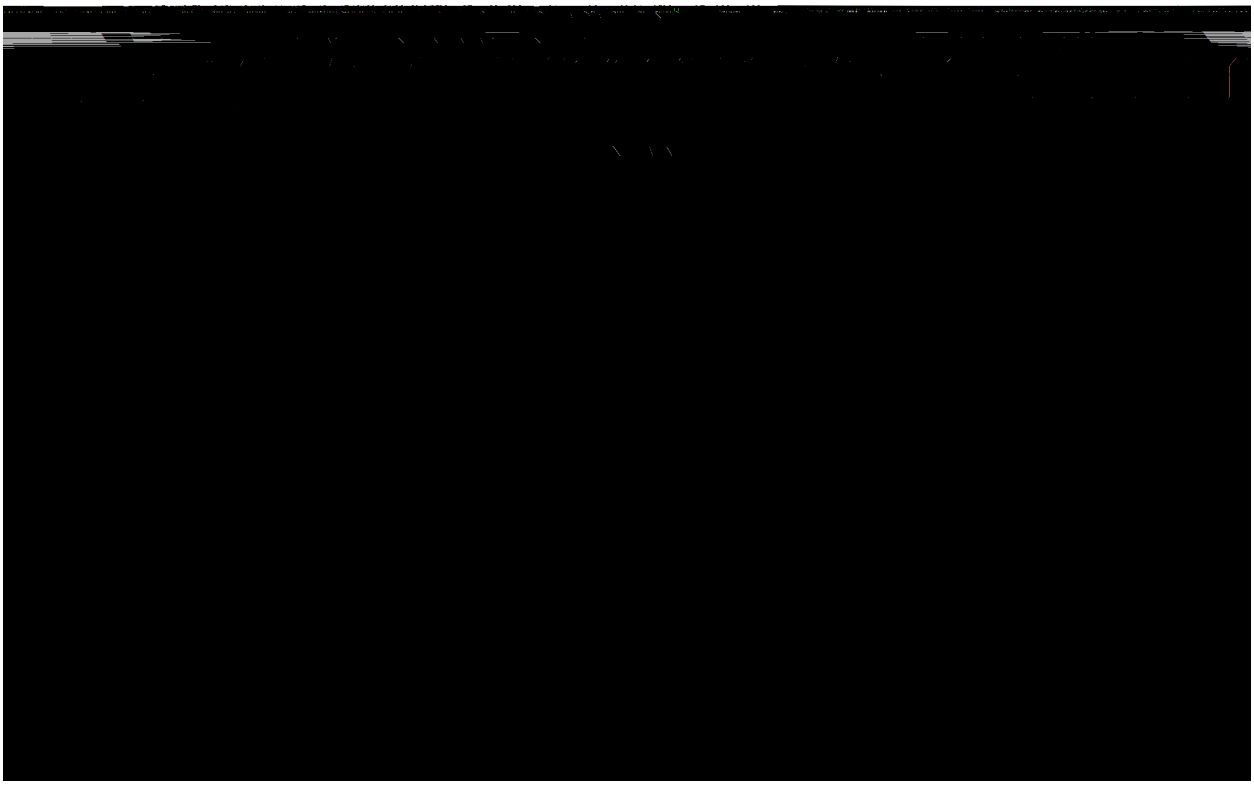


Figure 17.10: Linear diagram with three tracks for Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from *Listeria innocua* Clip11262 (NC_003212) plus basic cross-links shaded by percentage identity (see Section

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

These options are not covered here yet, so for now we refer you to the User Guide (PDF) included with

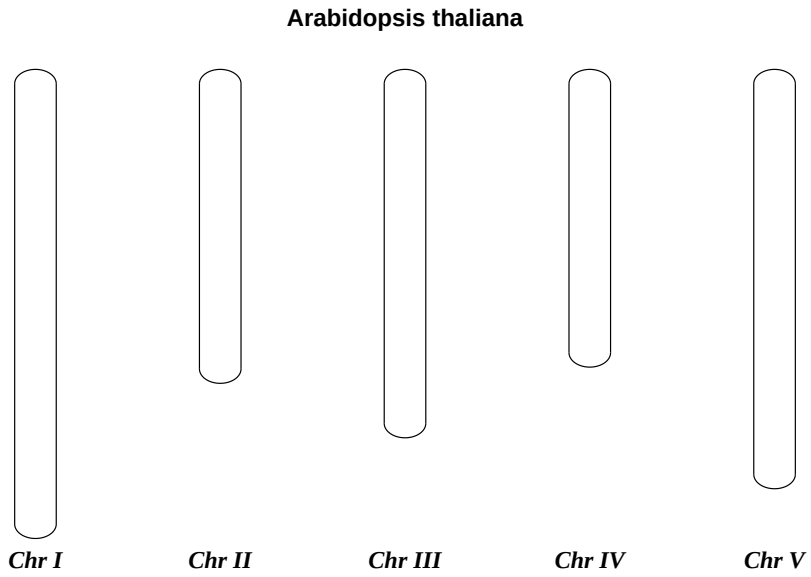


Figure 17.12: Simple chromosome diagram for

```
chr_diagram.draw("simple_chrom.pdf", "Arabidopsis thaliana")
```

```
#Add a closing telomere
end = BasicChromosome.TelomereSegment(inverted=True)
end.scale = telomere_length
cur_chromosome.add(end)

#This chromosome is done
```

Chapter 18

```
print("There are %d repair pathways and %d repair genes. The genes are:" % \
      (len(repair_pathways), len(repair_genes)))
print(", ".join(repair_genes))
```

The KEGG API wrapper is compatible with all endpoints. Usage is essentially replacing all slashes in the url with commas and using that list as arguments to the corresponding method in the KEGG module. Here are a few examples from the api documentation (http://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")
```

uppercase character from A to H, while columns are indicated by a two digit number, from 01 to 12. There

```
>>> 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)
[...]
```

```
>>> corrected = record.subtract_control(control='A01')
>>> record['A01'][63]
336.0
>>> corrected['A01'][63]
0.0
```

19.1.2.4 Parameters extraction

Those wells where metabolic activity is observed show a sigmoid behavior for the colorimetric data. To allow

```
area    4414.38
average_height  61.58
lag     48.60
max     143.00
min     12.00
plateau 120.02
slope   4.99
```

19.1.3 Writing Phenotype Microarray data

PlateRecord objects can be written to file in the form of JSON files, a format compatible with other software packages such as opm or DuctApe.

```
>>> phenotype.write(record, "out.json", "pm-json")
1
```

```
if count < lenuantedn:
```

Now, in order to use Bio.SeqIO to output the shu ed sequence, we need to construct a new SeqRecord with a new Seq object using this shu ed list. In order to do this, we need to turn the list of nucleotides

20.1.5 Sorting a sequence le

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

As a bonus, because it doesn't parse the data into `SeqRecord` objects a second time it should be faster. If you only want to use this with FASTA format, we can speed this up one step further by using the low-level FASTA parser to get the record identifiers and lengths:

```
from Bio.SeqIO.FastaIO import SimpleFastaParser
from Bio import SeqIO

# Get the lengths and ids, and sort on length
with open("ls_orchid.fasta") as in_handle:
    len_and_ids = sorted((len(seq), title.split(None, 1)[0]) for
                         title, seq in SimpleFastaParser(in_handle))
ids = reversed([id for (length, id) in len_and_ids])
del len_and_ids  # free this memory

record_index = SeqIO.index("ls_orchid.fasta", "fasta")
with open("sorted.fasta", "wb") as out_handle:
    for id in ids:
        out_handle.write(record_index.get_raw(id))
```

20.1.6 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.

This pulled out only 14580 reads out of the 41892 present. A more sensible thing to do would be to quality

```
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

This takes longer, as this time the output le contains all 41892 reads. Again, we're used a generator expression to avoid any memory problems. You could alternatively .94-14.948f.in, w rhe

```
original_reads = SeqIO.parse("SRR020192.fastq", "fastq")
trimmed_reads = trim_adaptors(original_reads, "GATGACGGTGT")
```

$$Q_{\text{PHRED}} = 10 \log_{10}(P_e) \quad (20.1)$$

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*

20.1.10 Converting FASTA and QUAL files into FASTQ files

FASTQ files hold *both* sequences and their quality strings. FASTA files hold *just* sequences, while QUAL files hold *just* the qualities. Therefore a single FASTQ file can be converted to or from *paired* FASTA and QUAL files.

Going from FASTQ to FASTA is easy:

```
from Bio import SeqIO
SeqIO.convert("example.fastq", "fastq", "example.fasta", "fasta")
```

Going from FASTQ to QUAL is also easy:

```
from Bio import SeqIO
SeqIO.convert("example.fastq", "fastq", "example.qual", "qual")
```

However, the reverse is a little more tricky. You can use `Bio.SeqIO.parse()`

```
['SRR020192.38240', 'SRR020192.23181', 'SRR020192.40568', 'SRR020192.23186']
>>> fq_dict["SRR020192.23186"].seq
```

20.1.13 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 3.9 about translation).

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.fna", "fasta")
>>> table = 11
>>> min_pro_len = 100
```

Here is a neat trick using the Seq object's `split` method to get a list of all the possible ORF translations in the six reading frames:

```
>>> for strand, nuc in [(+1, record.seq), (-1, record.seq.reverse_complement())]:
```

```
table = 11
min_pro_len = 100

def find_orfs_with_trans(seq, trans_table, min_protein_length):
    answer = []
    seq_len = len(seq)
```

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 le (as visualised in Section 17.1.9).

If however all you want to nd 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

Figure 20.1: Histogram of orchid sequence lengths.

20.2.2 Plot of sequence GC%

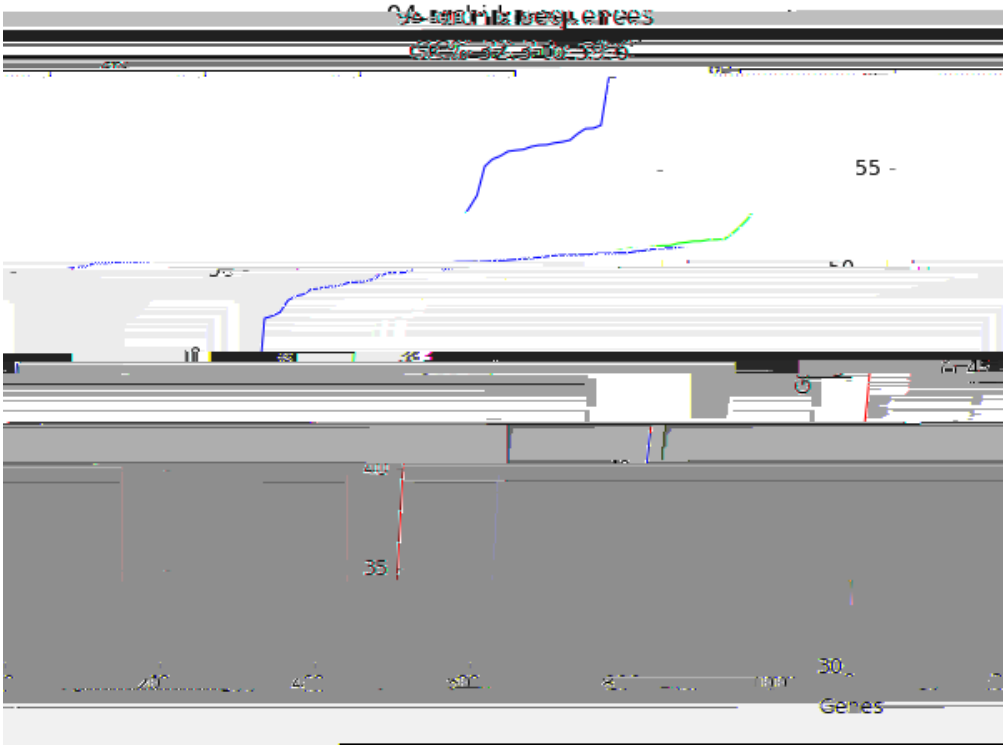


Figure 20.2: Histogram of orchid sequence lengths.

20.2.3 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 mis-match threshold. Here for simplicity we'll only look forfollerfecshcesf(eshonk Figure

orchidAST

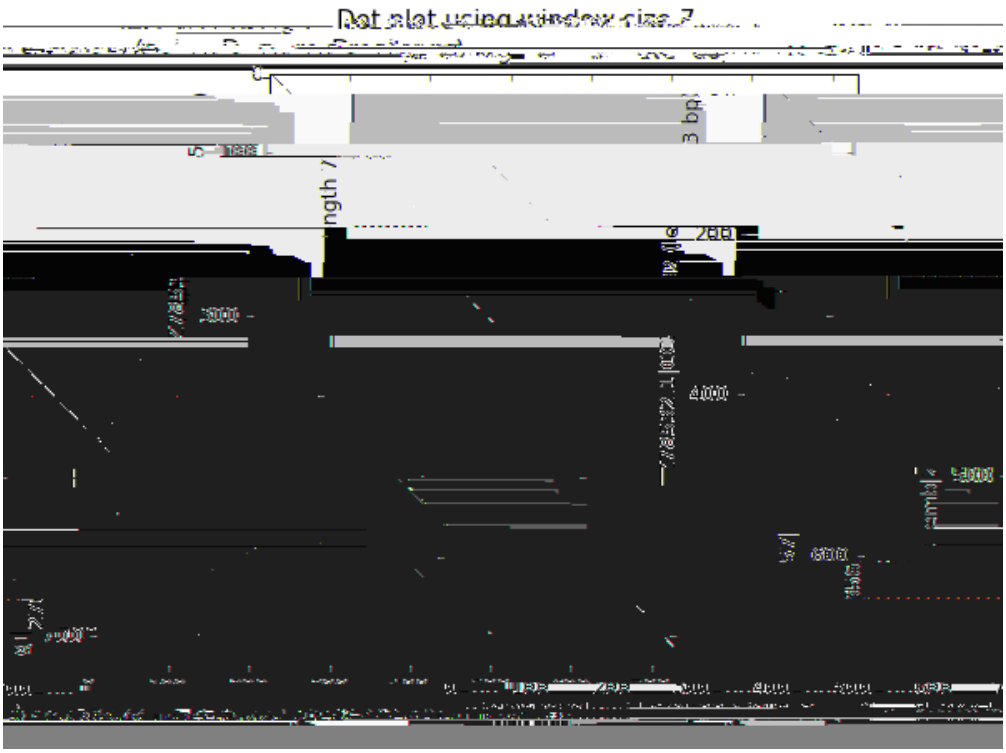


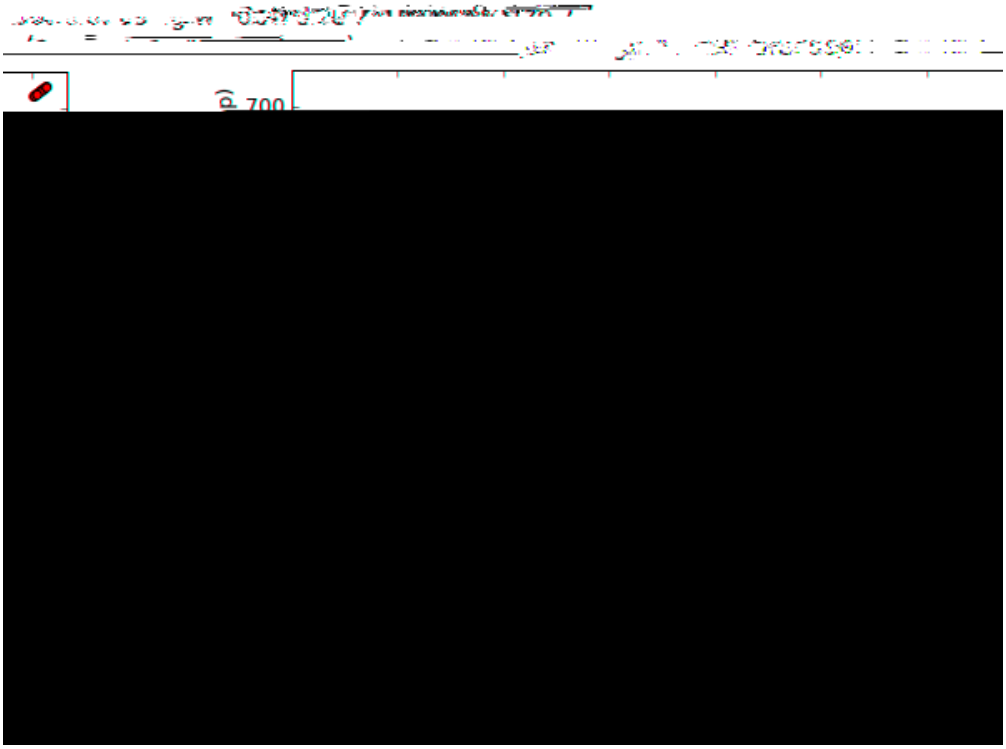
Figure 20.3: Nucleotide dot plot of two orchid sequence lengths (using pylab's imshow function).

Note that we have *not* checked for reverse complement matches here. Now we'll use the matplotlib's

```
dict_two = {}
for (seq, section_dict) in [(str(rec_one.seq).upper(), dict_one),
                            (str(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
#(Python 2.3 would require slightly different code here)
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* co-ordinates:

```
#Create lists of xr:
forTd[(section)-525(inen(myError:)]TJ20.921-11.955Td[(for)-525(i)-525(inet(din_dict[seow):)]TJ20.922-11
```



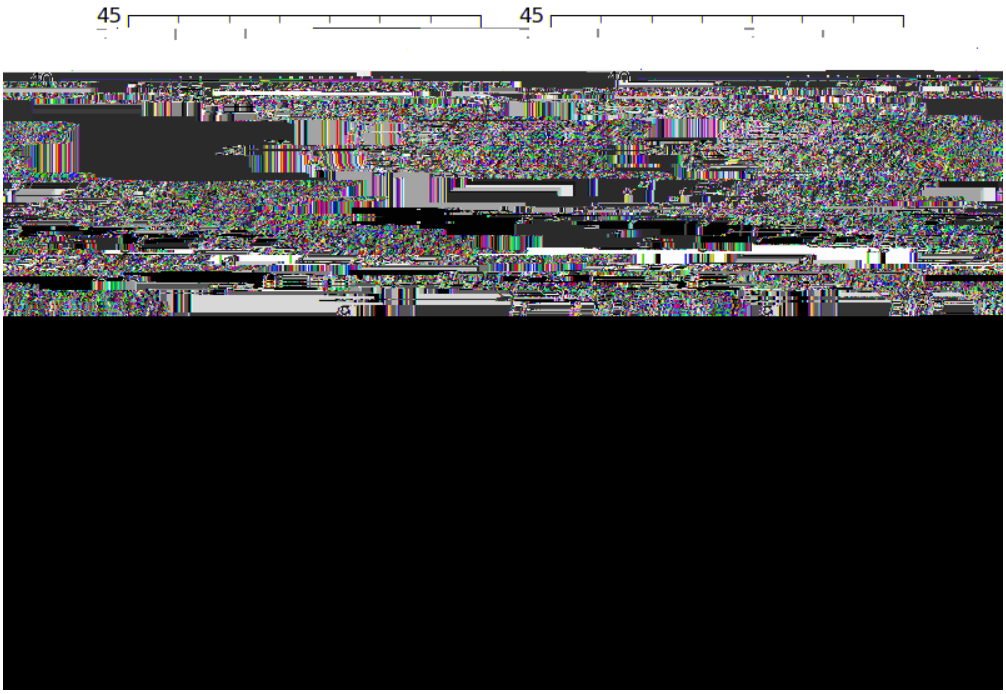


Figure 20.5: Quality plot for some paired end reads.

```
consensus = summary_align.dumb_consensus()
```

As the name suggests, this is a really simple consensus calculator, and will just add up all of the residues

Sections 6.4.1 and 20.3.1 contain more information on doing this.

20.5 BioSQL { storing sequences in a relational database

Chapter 21

The Biopython testing framework

Biopython has a regression testing framework (the file `run_tests.py`) based on unittest

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.

1. `test_Biospam.py` { The actual test code for your module.
2. `Biospam` [optional]{ A directory where any necessary input les will be located. If you have any output

(a) The long way:

online documentaion for unittest. If you are familiar with the unittest

to execute the tests when the script is run by itself (rather than imported from

```
$ 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 21.3), 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)
```

Chapter 22

Advanced

22.1 Parser Design

Many of the older Biopython parsers were built around an event-oriented design that includes Scanner and

(a) `__init__(self,data=None,alphabet=None, mat_name='', build_later=0):`
    i.

i. Full matrix size: N*N
ii. Half matrix size: N(N+1)/2

(a) `acc_rep_mat`: user provided accepted replacements matrix

(b) `exp_freq_table`: expected frequencies table. Used if provided, if not, generated from the `acc_rep_mat`.

(c) `logbase`

Summing up to 1.

When passing a dictionary as an argument, you should indicate whether it is a count or a frequency

Chapter 23

Where to go from here { contributing to Biopython

23.1 Bug Reports + Feature Requests

Getting feedback on the Biopython modules is very important to us. Open-source projects like this bene t greatly from feedback, bug-reports (and patches!) from a wide variety of contributors.

23.7 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 stu you want to work on. Normally, we try to have

Chapter 24

Appendix: Useful stu about Python

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 le 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 les, under Python 3 it is essential to use gzip in text mode.
See Section 5.2 for more examples like this, including reading bzip2 compressed les.

24.1.1 Creating a handle from a string

One useful thing gzip193(to)-285boe to n(fomaction)-285coenlaised ne string neto handle Tne example from the (Python)-333astndards lbrary:.

```
n(fe)-525(=)-525'A= strin\in
f.elen(rec'A= strin\in
```

Bibliography

[1] Peter J. A. Cock, Tiago Antao, Je rey T. Chang, Brad A. Chapman, Cymon J. Cox, Andrew Dalke, Iddo

[11] Douglas R. Cavener: \Comparison of the consensus sequence anking translational start sites in Drosophila and vertebrates." *Nucleic Acids Research* 15

[29] Robin Sibson: \SLINK: An optimally e cient algorithm for the single-link cluster method". *The Computer Journal* 16 (1): 30{34 (1973). https://doi.org/10.1093/comjnl/16.1.30

[30] George W. Snedecor, William G. Cochran: *Statistical methods*