Biopython Tutorial and Cookbook

Jeff Chang, Brad Chapman, Iddo Friedberg, Thomas Hamelryck,

Contents

2 Quick Start { What can you do with Biopython?

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 scientific publication?*

14. *Why doesn't* `Bio.Entrez.parse()`

28. *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 file.

For more general questions, the Python FAQ pages `http://www.python.org/doc/faq/` may be useful.

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

jecatdiersd(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 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 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, defined 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 files defined 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 file 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 file directly you'll find 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

Chapter 5

Sequence Input/Output

In this chapter we'll discuss in more detail the `Bio.SeqIO` module, which was briefly introduced in Chapter

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 [illegible] ereh7342(describ)-27(edh73

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

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

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

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

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

In either case, the result is:

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

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

Now, let's suppose you wanted to extract a list of the species from a FASTA file, rather than the

5.2 Parsing sequences from compressed files

In the previous section, we looked at parsing sequence data from a file. Instead of using a filename, you can give `Bio.SeqIO` a handle (see Section 24.1), and in this section we'll use handles to parse sequence from compressed files.

As you'll have seen above, we can use `Bio.SeqIO.read()` or `Bio.SeqIO.parse()` with a filename - for

5.3 Parsing sequences from the net

```
handle = Entrez.efetch(db="nucleotide", rettype="gb", retmode="text",
                       id="6273291,6273290,6273289")
```

`Bio.SeqIO.to_dict()` is the most flexible but also the most memory demanding option (see Section 5.4.1

5.4.1.1 Specifying the dictionary keys

Using the same code as above, but for the FASTA file instead:

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

This time the keys are:

```
['gi|2765596|emb|Z78471.1|PDZ78471', 'gi|2765646|emb|Z78521.1|CCZ78521', ...
 ..., 'gi|2765613|emb|Z78488.1|PTZ78488', 'gi|2765583|emb|Z78458.1|PHZ78458']
```

You should recognise these strings from when we parsed the FASTA file earlier in Section 2.4.1. Suppose

This should give:

```
Z78533.1 JUEoWn6DPhgZ9nAyowsgtoD9TTo
Z78532.1 MN/s0q9zDoCVEEc+k/IFwCNF2pY
...
Z78439.1 H+JfaShya/4yyAj7IbMqgNkxdxQ
```

```
>>> from Bio import SeqIO
```

```
>>> print(gb_vrl[``AB811634.1''].description)
```

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()`

```
                    +"SSAC", generic_protein),
                 id="gi|14150838|gb|AAK54648.1|AF376133_1",
                 description="chalcone synthase [Cucumis sativus]")

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

rec3 = SeqRecord(Seq("MVTVEEFRRAQCAEGPATVMAIGTATPSNCVDQSTYPDYYFRITNSEHKVELKEKFKRMC" \
                    +"EKSMIKKRYMHLTEEILKENPNICAYMAPSLDARQDIVVVEVPKLGKEAAQKAIKEWGQP" \
                    +"KSKITHLVFCTTSGVDMPGCDYQLTKLLGLRPSVKRFMMYQQGCFAGGTVLRMAKDLAEN" \
                    +"NKGARVLVVCSEITAVTFRGPNDTHLDSLVGQALFGDGAAAVIIGSDPIPEVERPLFELV" \
                    +"SAAQTLLPDSEGAIDGHLREVGLTFHLLKDVPGLISKNIEKSLVEAFQPLGISDWNSLFW" \
                    +"IAHPGGPAILDQVELKLGLKQEKLKATRKVLSNYGNMSSACVLFILDEMRKASAKEGLGT" \
                    +"TGEGLEWGVLFGFGPGLTVETVVLHSVAT", generic_protein),
                 id="gi|13925890|gb|AAK49457.1|",
                 description="chalcone synthase [Nicotiana tabacum]")

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

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

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

And if you open thg 0 Wg 0V6wweth1(AST)8shoul Tf 0 -lohak4(w)28(e)-3 format file:

5.5.1 Round trips

Chapter 6

Multiple Sequence Alignment objects

6.1.1 Single Alignments

As an example, consider the following annotation rich protein alignment in the PFAM or Stockholm file

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> print("Alignment length %i" % alignment.get_alignment_length())
Alignment length 52
>>> for record in alignment:
...     print("%s - %s" % (record.seq, record.id))
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA - COATB_BNmE/30-851
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVKRLFKKVSRKA -52
 - COATB_BN22/32-831
AGDDP---AKAAFNDSQASYATYIGYAWAMTVVIVGATIGVKRLFKKTSSKA - COATB_BM13/24-721
AGDDP---AKAAFDDSQASYATYIGYAWAMTVVIVGATIGVKRLFKKASSKA - COATB_BZJ2/1-49:
AGDDP---AKAAFDDSQASYATYIGYAWAMTVVIVGATIGVKRLFKKTSSKA -
```

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

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

6.2 Writing Alignments

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

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

texist,theo hle

```
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])
SingleLetterAlphabet() alignment with 7 rows and 3 columns
```

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

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

```
>>> 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 file, but you *can* tell it to write the output to stdout instead (useful ian

```
>>> from Bio import pairwise2
>>> from Bio import SeqIO
>>> from Bio.SubsMat.MatrixInfo import blosum62
>>> seq1 = SeqIO.read("alpha.faa", "fasta")
>>> seq2 = SeqIO.read("beta.faa", "fasta")
>>> alignments = pairwise2.align.globalds(seq1.seq, seq2.seq, blosum62, -10, -0.5)
>>> len(alignments)
Info>ts = l[(>*>>)-525(le[0]nn(alignments))]TJMV-LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTY...KYR0alignm
```

Chapter 7

BLAST

Hey, everybody loves BLAST right? I mean, geez, how can it get any easier to do comparisons between one of your sequences and every other sequence in the known world? But, of course, this section isn't about how

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

Or, you can use a `for`-loop:

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

Note though that you can step through the BLAST records only once. Usually, from each BLAST record you would save the information that you are interested in. If you want to save all returned BLAST records, you can convert the iterator into a list:

```
>>> blast_records = list(blast_records)
```

Now you can access each BLAST record in the list with an index as usual. If your BLAST file is huge though, you may run into memory problems trying to save them all in a list.

Usually, you'll be,yingeBLASTh aThen(,)-345a'll you to is to h(e)-34 rist
blast_records BLAST record in

```
from mporsd
blast_records =
blast_recors =
or in blast_recor.alignmeotds:
    for in
```

STRecord cone(v)27(ryethia)1gd youmighnte(v)27(rt)-35((w)28(an)78(t)-35((to)-36(exsrac(t)-35((from)-36((the)-335(BLAST)-36(outpute.)
ch iece1 informatiot.
forourl with soml inn ln thuisdh blasn rportt
blast_records = a herenoldn.yinge hist:

v your(e)-333wyndaerawthat sd in a BLAST recor.n

```
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 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

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

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

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

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

(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]-034(noe)-034jusoengso:

```
>>> 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 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
```

```
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`

>>> from Bio

Q[illegible]on

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 Quefunction806um(806ld)-222rhits)

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

t(en)a051plbgramquatricOAccesmanus,Biopn

us,Entrezogrammsing also, Oçonsistsing

9.2 EInfo: Obtaining information about the Entrez databases

EInfo provides field index term counts, last update, and available links for eacu, last NCBI'14eld

```
        <DbName>unigene</DbName>
        <DbName>unists</DbName>
</DbList>
</eInfoResult>
```

Since this is a fairly simple XML file, we could extract the information it contains simply by string

9.3 ESearch: Searching the Entrez databases

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.

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

```
ort os
m Bio import SeqIO
m Bio import Entrez
rez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
```

The `record` variable consists of a Python list, one for each database in which we searched. Since we specified only one PubMed ID to search for, `record` contains only one item. This item is a dictionary containing information about our search term, as well as all the related items that were found:

```
>>> record[0]["DbFrom"]
'pubmed'
>>> record[0]["IdList"]
['19304878']
```

The `"LinkSetDb"`

9.8 EGQuery: Global Query - counts for search terms

EGQuery provides counts for a search term in each of the Entrez databases (i.e. a global query). This

The resulting XML file has a size of 6.1 GB. Attempting `Entrez.read` on this file willsultinghas

```
                ...

...
                </Field>
        </FieldList>
                <Docsum>
        <DocsumList>
                <Docsum>
                <sName>PubDate</<sName>>
                <sType>4</<sType>>
```

9.12.1 Parsing Medline records

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

```
...
A high level interface to SCOP and ASTRAL implemented in python.
GenomeDiagram: a python package for the visualization of large-scale genomic data.
Open source clustering software.
PDB file parser and structure class implemented in Python.
```

Instead of parsing Medline records stored in files, you can also parse Medline records downloaded by `Bio.Entrez.efetch`. For example, let's look at all Medline records in PubMed related to Biopython:

```
>>> from Bio import Entrez
```

9.12.2 Parsing GEO records

GEO (

9.13 Using a proxy

```
...     if row["DbName"]=="nuccore":
...         print(row["Count"])
814
```

So, we expect to find 814 Entrez Nucleotide records (this is the number I obtained in 2008; it is likely 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:

```
>>]8ytep:handle8ytep:=8ytep:Entrez.esearch(db="nucleotide",8ytep:term="Cypripedioideae",8ytep:retmax=81
>>]8ytep:record8ytep:=8ytep:Entrez.read(handle)
```

Here,

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

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

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

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

```
>>> gi_list = search_results["IdList"]
>>> count = int(search_results["Count"])
>>> assert count == len(gi_list)
```

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"
>>> pmid = "14630660"
```

Chapter 10

Swiss-Prot and ExPASy

10.1 Parsing Swiss-Prot files

```
>>> from Bio import SwissProt
>>> record = SwissProt.read(handle)
```

This function should be used if the handle points to exactly one Swiss-Prot record. It raises a `ValueError` if no Swiss-Prot record was found, and also if more than one record was found.

We can now print out some information about this record:

```
>>> print(record.description)
'RecName: Full=Chalcone synthase 3; EC=2.3.1.74; AltName: Full=Naringenin-chalcone synthase 3;'
>>> for ref in record.references:
...     print("authors:", ref.authors)
...     print("title:", ref.title)
...
authors: Liew C.F., Lim S.H., Loh C.S., Goh C.J.;
title: "Molecular cloning and sequence analysis of chalcone synthase cDNAs of
Bromheadia finlaysoniana.";
>>> print(record.organism_classification)
['Eukaryota', 'Viridiplantae', 'Streptophyta', 'Embryophyta', ..., 'Bromheadia']
```

```
>>> from Bio import SwissProt
>>> descriptions = []
>>> handle = open("uniprot_sprot.dat")
>>> for record in SwissProt.parse(handle):
...     descriptions.append(record.description)
...
>>> len(descriptions)
468851
```

```
>>> from Bio.SwissProt import KeyWList
>>> handle = open("keywlist.txt")
>>> records = KeyWList.parse(handle)
>>> for record in records:
...     print(record['ID'])
...     print(record['DE'])
```

This prints

```
2Fe-2S.
Protein which contains at least one 2Fe-2S iron-sulfur cluster: 2 iron atoms
complexed to 2 inorganic nordeins nolfur atomt from
```

```
>>> record.name
```

```
CC   -!- Also hydrolyzes diacylglycerol.
PR   PROSITE; PDOC00110;
DR   P11151, LIPL_BOVIN ;  P11153, LIPL_CAVPO ;  P11602, LIPL_CHICK ;
DR   P55031, LIPL_FELCA ;  P06858, LIPL_HUMAN ;  P11152, LIPL_MOUSE ;
DR   O46647, LIPL_MUSVI ;  P49060, LIPL_PAPAN ;  P49923, LIPL_PIG   ;
DR   Q06000, LIPL_RAT   ;  Q29524, LIPL_SHEEP ;
//
```

In this example, the first line shows the EC (Enzyme Commission) number of lipoprotein lipase (second line). Alternative names of lipoprotein lipase are "clearing factor lipase", "diacylglycerol lipase", and "diglyceride lipase" (lines 3 through 5). The line starting with "CA" shows the catalytic activity of this enzyme. Comment lines start with "CC". The "PR" line shows references to the Prosite Documentation records, and the "DR" lines show references to Swiss-Prot records. Not of these entries are necessarily present in an Enzyme record.

In Biopython, an Enzyme record is represented by the `Bio.ExPASy.Enzyme.Record` class. This record derives from a Python dictionary and has keys corresponding to the two-letter codes used in Enzyme files. To read an Enzyme file containing one Enzyme record, use the `read` function in `Bio.ExPASy.Enzyme`:

```
>>> from Bio.ExPASy import Enzyme
>>> with open("lipoprotein.txt") as handle:
...     record = Enzyme.read(handle)
...
>>> record["ID"]
'3.5I11.955T55Td>>> record["L-525(with)955T56Td[.') as ha11.955T55Td>>> record[['C3(lipase)40.95are  hy
```

10.5.2 Searching Swiss-Prot

Now, you may remark that I knew the records' accession numbers beforehand. Indeed, `get_sprot_raw()`

```
6
>>> result[0]
{'signature_ac': u'PS50948', 'level': u'0', 'stop': 98, 'sequence_ac': u'USERSEQ1', 'start': 16, 'score
>>> result[1]
{'start': 37, 'stop': 39, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00005'}
>>> result[2]
{'start': 45, 'stop': 48, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00006'}
>>> result[3]
{'start': 60, 'stop': 62, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00005'}
>>> result[4]
{'start': 80, 'stop': 83, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00004'}
>>> result[5]
{'start': 106, 'stop': 111, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00008'}>>>>>>>>>=25('sequEQ1
```

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

11.1.4 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 (

This is the way many structural biologists/bioinformaticians think about structure, and provides a simple but efficient way to deal with structure. Additional stuff is essentially added when needed. A UML diagram of the `Structure` object (forget about the `Disordered` classes for now) is shown in Fig. 11.1. Such a data

```
>>> child_entity = parent_entity[child_id]
```

You can also get a list of all child Entities of a parent Entity object. Note that this list is sorted in a specific way (e.g. according to chain identifier for Chain objects in a Model object).

```
>>> child_list = parent_entity.get_list()
```

You can also get the parent from a child:

```
>>> parent_entity = child_entity.get_parent()
```

11.2.2 Model

The id of the Model object is an integer, which is derived from the position of the model in the parsed file (they are automatically numbered starting from 0). Crystal structures generally have only one model (with

The reason for the hetero-flag is that many, many PDB files use the same sequence identifier for an amino acid and a hetero-residue or a water, which would create obvious problems if the hetero-flag was not used.

Unsurprisingly, a Residue object stores a set of Atom children. It also contains a string that specifies the

11.3 Disorder

DisorderedResidue

```
...         for residue in chain:
...             for atom in residue:
...                 print(atom)
...
```

There is a shortcut if you want to iterate over all atoms in a structure:

```
>>> atoms = structure.get_atoms()
>>> for atom in atoms:
...     print(atom)
...
```

Similarly, to iterate over all atoms in a chain, use

```
>>> atoms = chain.get_atoms()
>>> for atom in atoms:
...     print(atom)
...
```

Iterating over all residues of a model

or if you want to iterate over all residues in a model:

```
>>> residues = model.get_residues()
>>> for residue in residues:
...     print(residue)
...
```

You can also use the `Selection.unfold_entities` function to get all residues from a structure:

```
>>> res_list = Selection.unfold_entities(structure, 'R')
```

or to get all atoms from a chain:

```
>>> atom_list = Selection.unfold_entities(chain, 'A')
```

Obviously, `A=atom`, `R=residue`, `C=chain`, `M=model`, `S=structure`. You can use this to go up in the hierarchy, e.g. to get a list of (unique) `Residue` or `Chain` parents from a list `e`,

```
>>> atom_list = Selec0 -19.262ld_entities(strget_residues()
```

Print out the coordinates of all CA atoms in a structure with B factor greater than 50

```
>>> for model in structure.get_list():
...     for chain in model.get_list():
...         for residue in chain.get_list():
...             if residue.has_id("CA"):
...                 ca = residue["CA"]
...                 if ca.get_bfactor() > 50.0:
```

taets perform neighbor lookup. The neighbor lookup is done using a KD tree module

in C (see

11.7 Common problems in PDB files

It is well known that many PDB files contain semantic errors (not the structures themselves, but their representation in PDB files). Bio.PDB tries to handle this in two ways. The PDBParser object can behave in two ways: a restrictive way and a permissive way, which is the default.

Example:

```
# Permissive parser
>>> parser = PDBParser(PERMISSIVE=1)
>>> parser = PDBParser() # The same (default)
# Strict parser
>>> strict_parser = PDBParser(PERMISSIVE=0)
```

In the permissive state (DEFAULT), PDB files that obviously contain errors "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 [18, 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 that this atom is probably shared by Ser and Pro 22, as Ser 22 misses the N atom. Again, this points to a problem in the file: the N atom should be present in both the Ser and the Pro residue, in both cases associated with a suitable altloc identifier.

11.7.2 Automatic correction

Some errors are quite common and can be easily corrected without much risk of making a wrong interpretation. These cases are listed below.

11.7.2.1 A blank altloc for a disordered atom

Normally each disordered atom should have ahouldank altloc identifier-419(Helo)28(v)288(v)28r,(b)-226(tr8(e)-256r8(e)-25ma

11.7.2.1

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`

11.9 General questions

11.9.1 How well tested is Bio.PDB?

Pretty well, actually. Bio.PDB has been extensively tested on nearly 5500 structures from the PDB - all structures seemed to be parsed correctly. More details can be found in the Bio.PDB Bioinformatics article. Bio.PDB has been used/is being used in many research projects as a reliable tool. In fact, I'm using Bio.PDB almost daily for research purposes and continue working on improving it and adding new features.

11.9.2 How fast is it?

The `PDBParser` performance was tested on about 800 structures (each belonging to a unique SCOP superfamily). This takes about 20 minutes, or on average 1.5 seconds per structure. Parsing the structure of the large ribosomal subunit (1FKK), which contains about 64000 atoms, takes 10 seconds on a 1000 MHz PC. In short: it's more than fast enough for many applications.

Chapter 12

Bio.PopGen: Population genetics

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

Chapter 13

Phylogenetics with Bio.Phylo

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

The

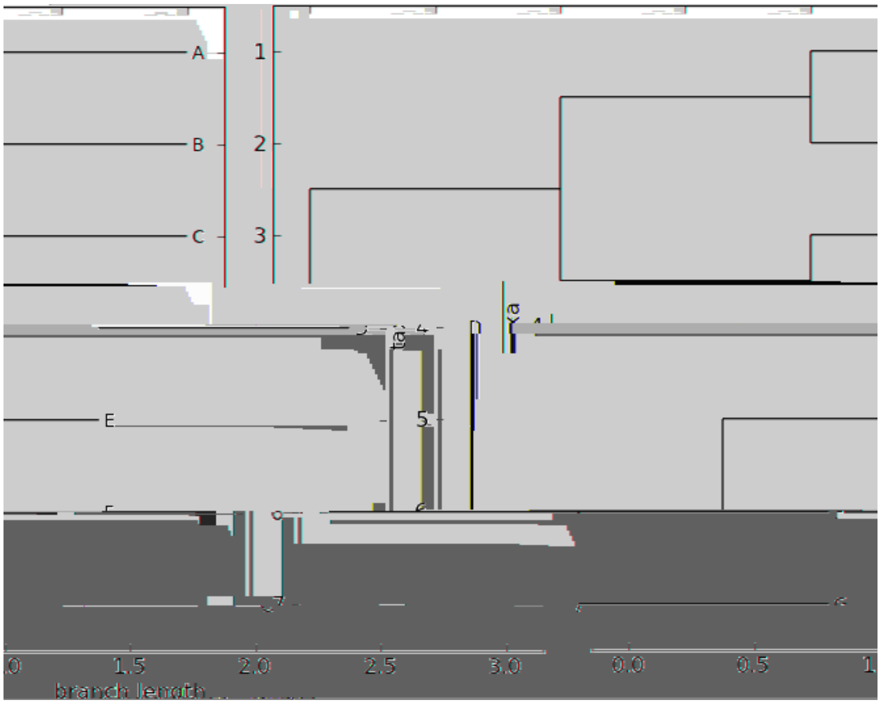


Figure 13.1: A rooted tree drawn with `Phylo.draw`.

Note that the file 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.

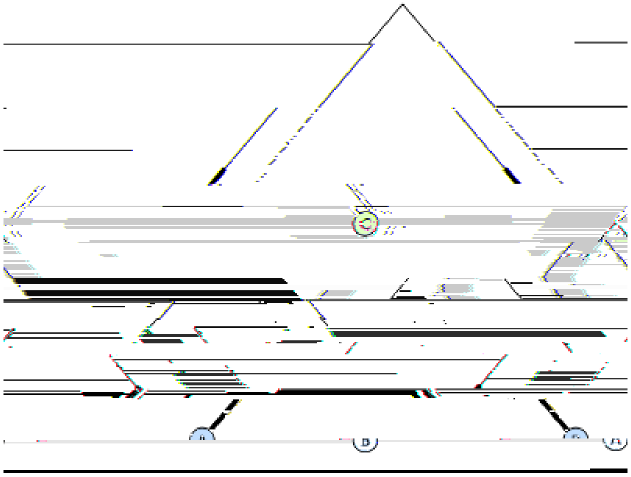


Figure 13.4 /Im8 Do Q Q 0 g 0 G 1 0 0 1 -189 -166.6(8eeted)-3 [()1 -wnted wihe

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 floating-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, 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:

```
>>> 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

```
>>> 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
********************************************************************************
```

To parse a TRANSFAC file, use

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>
```

```
>>> background = {'A':0.3,'C':0.2,'G':0.2,'T':0.3}
>>> 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 different thresholds. We can specify the requested false-positive rate (probability of \finding" 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 finding" 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(thres`

```
        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 different pseudocounts for them. Setting `motif.pseudocounts` to `None` resets it to its default value of zero.

The position-specific 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-specific 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 motif.background to None resets it 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 with hex 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 instances), they also provide some extra functionality, by adding additional information about the instances.

```
>>> motifsM[0].consensus
Seq('CTCAATCGTA', IUPACUnambiguousDNA())
>>> motifsM[0].instances[0].sequence_name
'SEQ10;'
```

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

$$
\begin{aligned}
{}^{(0)}_{x} &= \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}; \\
{}^{(0)}_{y} &= \sqrt{\frac{1}{n}\sum_{i=1}^{n} y_i^2}:
\end{aligned}
$$

```
>>> 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==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]ngl t:4lusteridtranspose8 1.9626 Array.f164f0lTcd[(u)-424(th-f11.9626 Tf 149.234 0 [7(8]49ata

`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 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

```
>>> 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.

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):

```
distance = array([[0.0, 1.1, 2.3],
                  [1.1, 0.0, 4.5],
```

The parameter is a parameter that decreases at each iteration step. We have used a simple linear function of the iteration step:

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

${}_{\text{init}}$ is the initial value of 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

`components`
The principal components.

`eigenvalues`
The eigenvalues corresponding to each of the principal components.

The original matrix `data` can be recreated by calculating appropriate gene and sample. The 5.8 in row 2 column 4 means that the observed value for gene YAL001C 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:

Calculating the distance matrix

To calculate the distance matrix between the items stored in the record, use

```
>>> 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==0`), or between the columns of `data` (`transpose==1`).

`dist` (default: `'e'`, Euclidean distance)
Defines the distance function to be used (see 15.1).

transpose

Saving the clustering result

This will create the files `cyano_result_K_G2_A2.cdt`, `cyano_result_K_G2.kgg`, and `cyano_result_K_A2.kag`.

15.9 Auxiliary functions

`median(data)` returns the median of the 1D array `data`.

`mean(data)` returns the mean of the 1D array `data`.

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 different 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

$$\Pr(\mathrm{OP}|x_1, x_2) = \frac{\exp({}_0 + {}_1 x_1 + {}_2 x_2)}{1 + \exp({}_0 + {}_1 x_1 + {}_2 x_2)} \tag{16.2}$$

$$\Pr(\mathrm{NOP}|x_1, x_2) = \frac{1}{1 + \exp({}_0 + {}_1 x_1 + {}_2 x_2)} \tag{16.3}$$

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.

Gene pair	Intergene distance x_1	Gene expression score x_2
yxcE | *yxcD*	6	-173.143442352
yxiB | *yxiA*		

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 pPCP1.

Figure 17.2: Simple circular diagram for *Yersinia pestis biovar Microtus* plasmid pPCP1.

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(FeatureLocat150orward"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
```

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 different sigils (see Section 17.1.7)

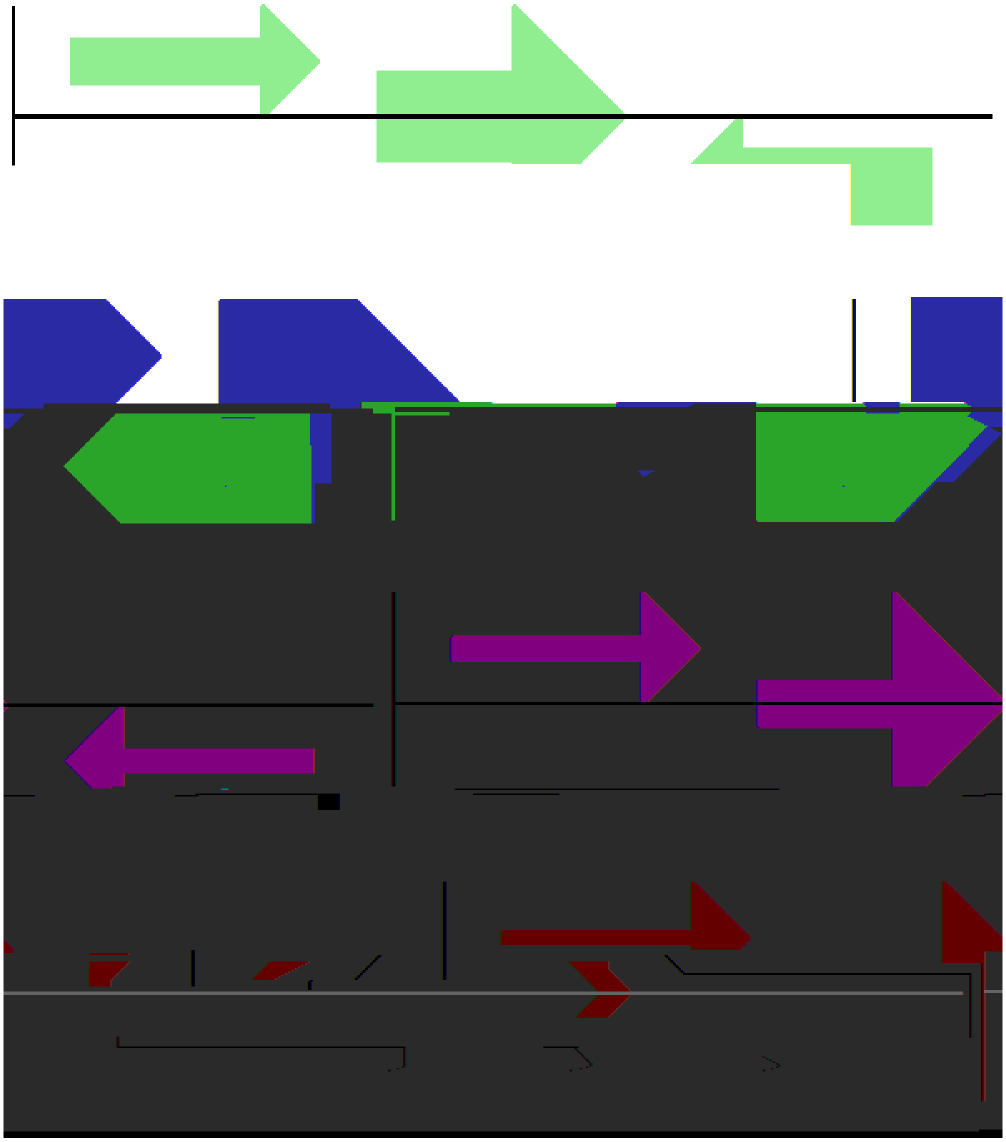


Figure 17.5: Simple GenomeDiagram showing arrow shaft options (see Section 17.1.8)

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

All the shaft and arrow head options shown above for the sigil ARROW.Tf251.4470Td[=.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")

gd_diagram.draw(format="circular", circular=True, pagesize=(20*cm,20*cm),
                start=0, end=len(record), circle_core = 0.5)
gd_diagram.write("plasmid_circular_nice.pdf", "PDF")
gd_diagram.write("plasmid_circular_nice.eps", "EPS")
gd_diagram.write("plasmid_circular_nice.svg", "SVG")
```

The expected output is shown in Figures 17.7 and 17.8.

Figure 17.7: Linear diagram for *Yersinia pestis biovar Microtus* plasmid pPCP1 showing selected restriction digest sites (see Section 17.1.9).

17.1.10 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) [5]. We'll need the GenBank files for the following three phage:

`NC_002703` { Lactococcus phage Tuc2009, complete genome (38347 bp)

`AF323668` { Bacteriophage bIL285, complete genome (35538 bp)

`NC_003212` { *Listeria innocua*

Figure 17.8: Circular diagram for *Yersinia pestis biovar Microtus* plasmid pPCP1 showing selected restriction

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 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).

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).

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.

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

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

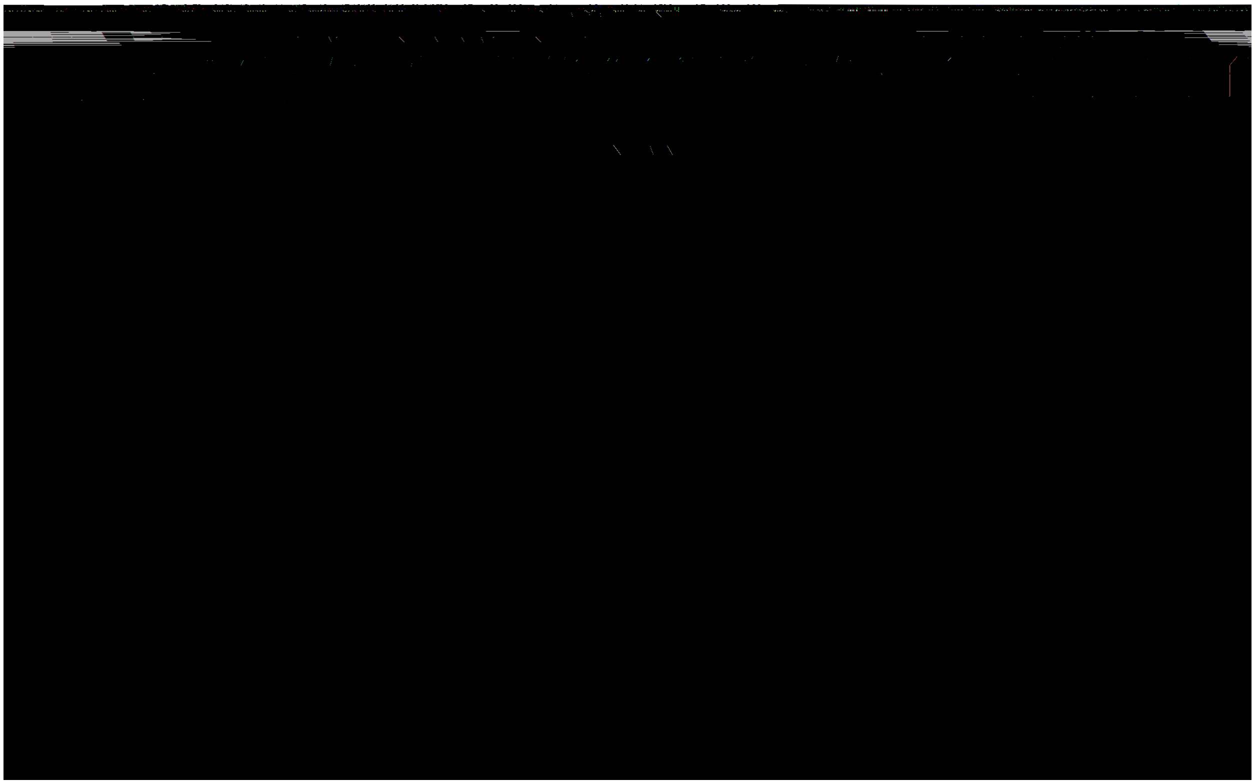


Figure 17.10: Linear diagram with three tracks for Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from *Listeria innocua* Clip11262 (NC_003212) plus basic cross-links shaded by percentage identity (see Section 17.1.11).

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`

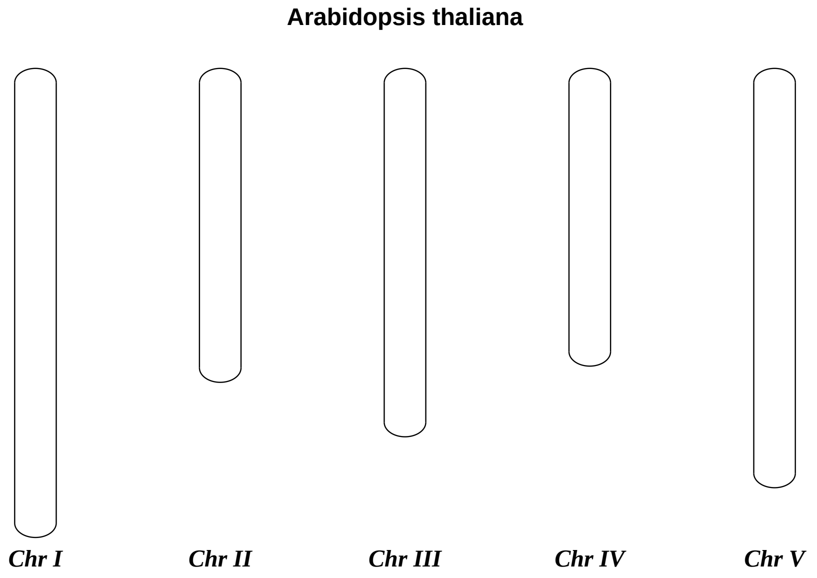


Figure 17.12: Simple chromosome diagram for

```
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

KEGG

KEGG (`http://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.

```
>>> 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.

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)
[...]
```

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.

19.1.2.2 Accessing interpolated data

To make it easier to compare different experiments and in general to allow a more intuitive handling of

```
>>> 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
```

Chapter 20

Cookbook { Cool things to do with it

Personally I prefer the following version using a function to shuffle the record and a generator expression

First we scan through the file once using, recording the record identifiers and theirthis sorted list of identifiersallows us to retrieve the recor

```
              if min(rec.letter_annotations["phred_quality"]) >= 20)
count = SeqIO.write(good_reads, "good_quality.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

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

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
```

```
>>> fq_dict.keys()[:4]
['SRR020192.38240', 'SRR020192.23181', 'SRR020192.40568', 'SRR020192.23186']
>>> fq_dict["SRR020192.23186"].seq
Seq('GTCCCAGTATTCGGATTTGTCTGCCAAAACAATGAAATTGACACAGTTTACAAC...CCG', SingleLetterAlphabet())
```

When testing this on a FASTQ file with seven million reads, indexing took about a minute, but record access was almost instant.

The example in Section 20.1.5 show how you can use the `Bio.SeqIO.index()` function to sort a large FASTA file – this could also be used on FASTQ files.

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 file (as visualised in Section 17.1.9).

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

94 orchid sequences

[redacted]

Figure 20.1: Histogram of orchid sequence lengths.

(ue)334(coulde)-333doewithelo:op,ebute

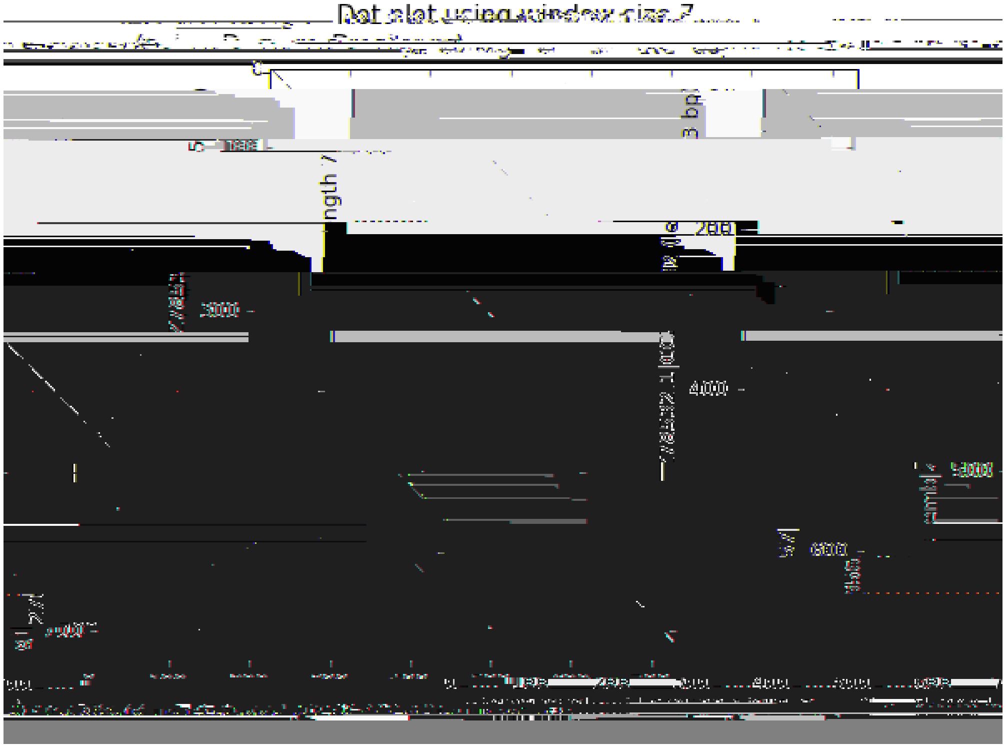


Figure 20.3: Nucleotide dot plot of two orchid sequence lengths (using pylab's imshow function).

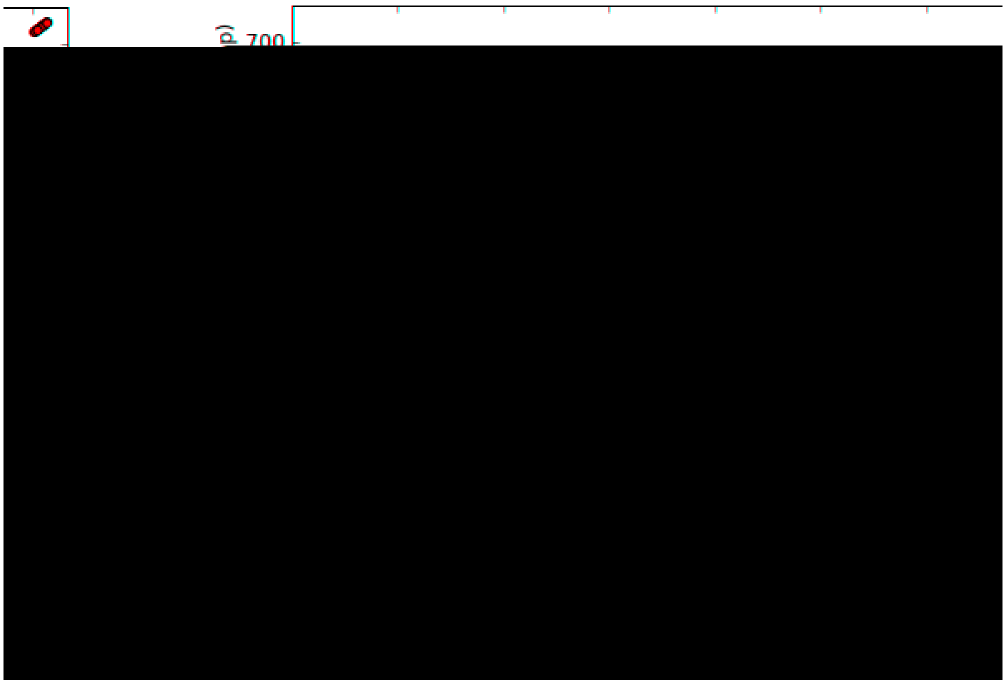
Note that we have *not*

```
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
```

[illegible]


700

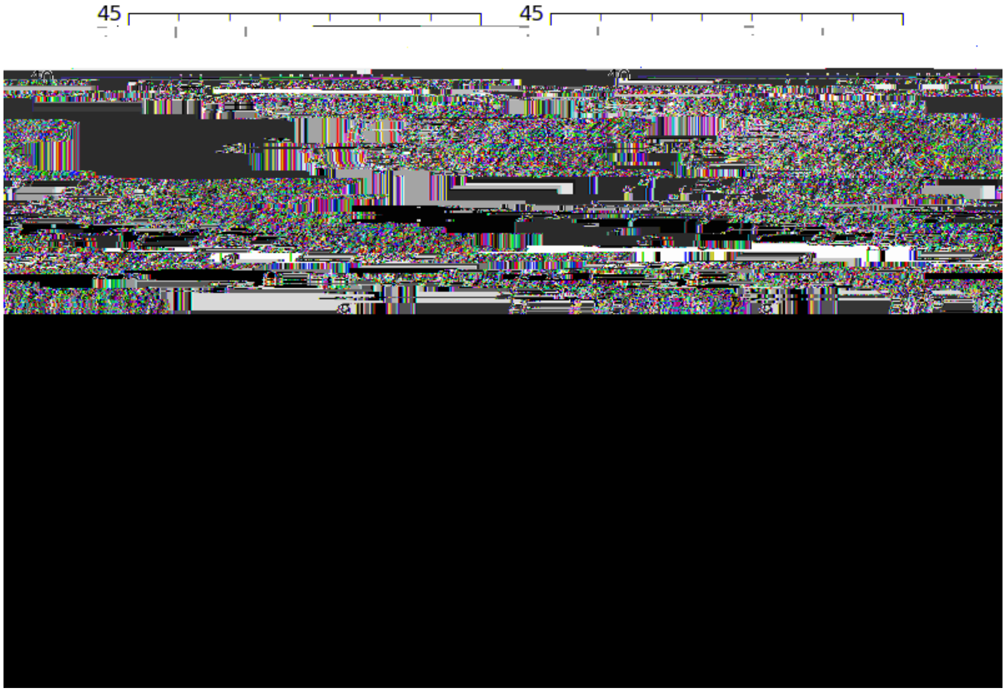


Figure 20.5: Quality plot for some paired end reads.

```
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

Q_i { The expected frequency of a letter i

20.4 Substitution Matrices

Substitution matrices are an extremely important part of everyday bioinformatics work. They provide the

```
>>> from Bio import SubsMat
>>> my_arm = SubsMat.SeqMat(replace_info)
```

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 files 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.

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 benefit 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 stuff you want to work on. Normally, we try to have

Chapter 24

Appendix: Useful stuff about Python

Bibliography

[1]

[12] Timothy L. Bailey and Charles Elkan: \Fitting a mixture model by expectation maximization to discover motifs in biopolymers",

[30] Pablo Tamayo, Donna Slonim, Jill Mesirov, Qing Zhu, Sutisak Kitareewan, Ethan Dmitrovsky, Eric S. Lander, Todd R. Golub: \Interpreting patterns of gene expression with self-organizing maps: Methods and application to hematopoietic differentiation". *Proceedings of the National Academy of Science USA* 96 (6): 2907{2912 (1999). doi:10.1073/pnas.96.6.2907

[31] Robert C. Tryon, Daniel E. Bailey: *Cluster analysis*. New York: McGraw-Hill (1970).

[32]