Biopython Tutorial and Cookbook

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

Contents

l.95e3-5th 0 g 0 logg.rægress(.)-s.. . . . . .3S500(.)-499(.)-5004-300 G 100(.41.mz9-500(.)-50

Chapter 1

Introduction

1.1 What is Biopython?

The Biopython Project is an international association of developers of freely available Python (http://www.python.org) tools for computational molecular biology. Python is an object oriented, interpreted, exible

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 if appropriate, in particular as a reference for speci c

If the `\import Bio"` line fails, Biopython is not installed. Note that those are double underscores

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 le.

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

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 Parsing sequence file formats

A large part of much bioinformatics work involves dealing with the many types of file formats designed to

2.4.2 Simple GenBank parsing example

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

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

This should give:

```
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', IUPACAmbiguousDNA())
740
...
Z78439.1
```

Chapter 3

Sequence objects

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

However, where possible you should specify the alphabet explicitly when creating your sequence objects - in this case an unambiguous DNA alphabet object:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("AGTACACTGGT", IUPAC.unambiguous_dna)
>>> my_seq
Seq('AGTACACTGGT', IUPACUnambiguousDNA())
>>> my_seq.alphabet
IUPACUnambiguousDNA()
```

Unless of course, this really is an amino acid sequence:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_prot = Seq("AGTACACTGGT", IUPAC.protein)
>>> my_prot
Seq('AGTACACTGGT', IUPACProtein())
>>> my_prot.alphabet
IUPACProtein()
```

3.2 Sequences act like strings

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

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCG", IUPAC.unambiguous_dna)
>>> for index, letter in enumerate(my_seqe:PAC
>>>stabet[(ate(my_seqe051))]TJ0-11.955Td5051

>>>stabet(my_[0]051>> lett51
>>>stabet(my_[2]051>>rdex, lett51
>>>stabet(my_[-1]051>> lett51
```

The Seq object has a .count()

```
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Seq import Seq
>>> protein_seq = Seq("EVRNAK", IUPAC.protein)
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> protein_seq + dna_seq
Traceback _seqosrt
```

```
>>> from Bio.Alphabet import
>>> protein_se.aAlphabet
>>> dna_se.aAlphabet
>>> protein_seq + dna_seq
```

```
  >>> from Bio.Seq import Seq
  >>> from Bio.Alphabet import
  >>> from Bio.Alphabet import IUPAC
  >>>dna_seq = SeqGATCGATGCACGT",
  >>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
  >>>dna_seq
  SeqGATCGATGCACG'",
  >>> dna_seq
  SeqRNA"ACG'", IUPAUunambiguouDNAnaa)
  >>>dna_seq + dna_seq
  >>>>>> dna_se.aAlphabrhis,d2.96PAC
  >>>dna_seq = SeqGATCGATGCACGT",
  >>>SeqGATCGATGCACGT",
```

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

For example, you might argue that the two DNA Seq objects Seq("ACGT", IUPAC.unambiguous_dna) and Seq("ACGT", IUPAC.ambiguous_dna) should be equal, even though they do have di erent alphabets. Depending on the context this could be important.

This gets worse { suppose you think Seq("ACGT", IUPAC.unambiguous_dna) and Seq("ACGT") (i.e. the default generic alphabet) should be equal. Then, logically, Seq("ACGT", IUPAC.protein) and Seq("ACGT") should also be equal. Now, in logic if $A = B$ and $B = C$, by transitivity we expect $A = C$. So for logical consistency we'd require Seq("ACGT", IUPAC.unambiguous_dna) and Seq("ACGT", IUPAC.protein) to be equal { which most people would agree is just not right. This transitivity also has implications for using Seq objects as Python dictionary keys.

Now, in everyday use, your sequences will probably all have the same alphabet, or at least all be the same type of sequence (all DNA, all RNA, or all protein). What you probably want is to just compare the sequences as strings { which you can do explicitly:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> seq1 = Seq("ACGT", IUPAC.unambiguous_dna)
>>> seq2 = Seq("ACGT", IUPAC.ambiguous_dna)
>>> str(seq1) == str(seq2)
True
>>> str(seq1) == str(seq1)
True
```

So, what does Biopython do? Well, as of Biopython 1.65, sequence comparison only looks at the sequence, essentially ignoring the alphabet:

```
>>> seq1 == seq2
True
>>> seq1 == "ACGT"
True
```

As an extension to this, using sequence objects as keys in a Python dictionary is now equivalent to using

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous_dna)
```

Observe what happens if you try to edit the sequence:

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

>>> from Bio.Seq importSeq
>>> from Bio.Alphabet import IUPAC
>>>my_seq =("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA"

3.13 UnknownSeq objects

The UnknownSeq object is a subclass of the basic Seq

Chapter 4

.annotations

```
>>> record.seq


 from theq-362(LOCUSq-361(line,q-369(while)-362(theq)]T/FF349.9626Tf174.41160Td[idq)]T/F849.9626Tf14.062
>>> recorddescriptionq



>>> recordletter_annotationsq



>>> recorddbxrefsq



                      i454i454gets-333(aburesqq)]TJ00he.g.141mplete>
```

```
>>> my_location.sTt

>>>my_location.sTt)

>>> my_locatioend

>>>my_locatioend)
```

```
>>> 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 Sequence described by a feature or location

A SeqFeature or location object doesn't directly contain a sequence, instead the location (see Section 4.3.2)

4.4 References

Another common annotation related to a sequence is a reference to a journal or other published work

```
>>> len(sub_record)
500
>>> len(sub_record.features)
2
```

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

```
>>> print(sub_record.features[0])
type: gene
location: [42:480](+)
qualifiers:
    Key: db_xref, Value: [':rID:2767712']51
    Key: geef, Value: pim']51
    Key:lus_tagef, Value: [(YP_pPCP']51)]T-J20.921-11.955Td<BLANKLINE>(:)]TJ0g0G0g0TJ0-20.726Td[(>>>)-5
    Key: db_xref, Value: [I:45478716'ef,[':rID:2767712']51
    Key: geef, Value: pim']51


    Key: Value: NP_91.971.1']51
    Key: Value: 11']51
    Key: Value: MGGGMISKLFCLALIFLSSSGLAEKNTYTAKDILQNLELNTFGNSLSH...']51
```

```
>>> edited = record[:20] + record[21:]
```

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

4.8 Reverse-complementing SeqRecord objects

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

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

5.1.4 Extracting data

The SeqRecord object and its annotation structures are described more fully in Chapter 4. As an example of how annotations are stored, we'll look at the output from parsing the first record in the GenBank file ls_orchid.gbk.

```
from Bio import SeqIO
record_iterator = SeqIO.parse("ls_orchid.gbk", "genbank")
first_record = next(record_iterator)
print(first_record)
```

That should give something like this:

```
ID: Z78533.1
Name: Z78533
Descriuh8sin.atrapeatu,l[qfirch2(8s)(f3o1rjNA(5Tr0)'-5e-[3(2(Pyt34g)-3(1Ts1i(3fg1)534(f)86T9(2H4(f-3(0)Nc.7(8523(.1e)
             print(first_recordFannot83lgs)
```

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 comprehendefrom Bio import SeqIO

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 22.1

5.3 Parsing sequences from the net

`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', ...
```

This should give:

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

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index("ls_orchid.fasta", "fasta")
>>> len(orchid_dict)
94
>>> orchid_dict.keys()
```

5.4.3 Sequence les as Dictionaries { Database indexed les

Biopython 1.57 introduced an alternative, `Bio.SeqIO.index_db()`, which can work on even extremely large les since it stores the record information as a le on disk (using an SQLite3 database) rather than in memory. Also, you can index multiple les together (providing all the record identi ers are unique).

The `Bio.SeqIO.index()` function takes three required arguments:

Index lename, we suggest using something ending `.idx`

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

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

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

6.1.1 Single Alignments

As an example, consider the following annotation rich protein alignment in the PFAM or Stockholm le format:

```
# STOCKHOLM 1.0
#=GS COATB_BKcE/30-8.1
#=GS COATB_BKcE/30-8.1 iflM
#=GS
#=GS COATB_BK22/32-831
#=GS COATB_BM13/24-751
#=GS COATB_BM13/24-751
#=GS COATB_BM13/24-751
#=GS COATB_BZJ2/()491          CS
#=GS         CS
#=GSnh4M
#=GS COATB_BIF1/22-731
#=GS COATB_BIF1/22-731 ifkM
COATB_BKcE/30-8.1
#=RS COATB_BKcE/30-8.1 HHHHHHHHHHHHHH--HHHHHHHHH--HHHHHHHHHHHHHHHHHHHHHH----0

COATB_BK22/32-831
COATB_BM13/24-751
#=RS COATB_BM13/24-751 S-T...CHCHHHHCCCCTCCCTTCHHHHHHHHHHHHHHHHHHHHHHCTT--0
COATB_BZJ2/()491

#=RS -...-HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH--0
COATB_BKF1/22-731
#=RS COATB_BKF1/22-731 HHHH--HHHHHHH--HHHHHHHH--HHHHHHHHHHHHHHHHHHHHHHH---0
#=CScon
#=CScon
```

```
>>> 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))
```

```
Epsilon   CCCAAC
...
    5     6
Alpha     AAAACC
Beta      ACCCCC
Gamma     AAAACC
Delta     CCCCAA
Epsilon   CAAACC
```

If you wanted to read thi5-333(read)-333(thi5-333(read)ed)-333(thi5-333(read)-333(thi5-3Cusingy)27(oAAC)]TJ 0 -11154

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

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

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 manipulation like removing certain rows or columns.

Suppose yo(to)-484(w)28(an)2T3neows-4-484ho27(ts-4-484m(an)2Ty-4-484lignmen)272T3so

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 4.5, 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()`

DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRL...SKA COATB_BPI22/32-83

```
>>> 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
```

```
>>> 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
```

```
>>> from Bio.Emboss.Applications import NeedleCommandline
>>> needle_cline NeedleCommand(r"C:\EMBOSS\525(ne.exe",line)]TJ0-11.955T...
```

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 qblast function can return the BLAST results in various formats, which you can choose with the optional format_type keyword: "HTML", "Text", "ASN.1", or "XML". The default is "XML", as that is

```
>>> save_file = open("my_blast.xml", "w")
>>> save_file.write(result_handle.read())
>>> save_file.close()
>>> result_handle.close()
```

Or, you can use a for-loop:

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

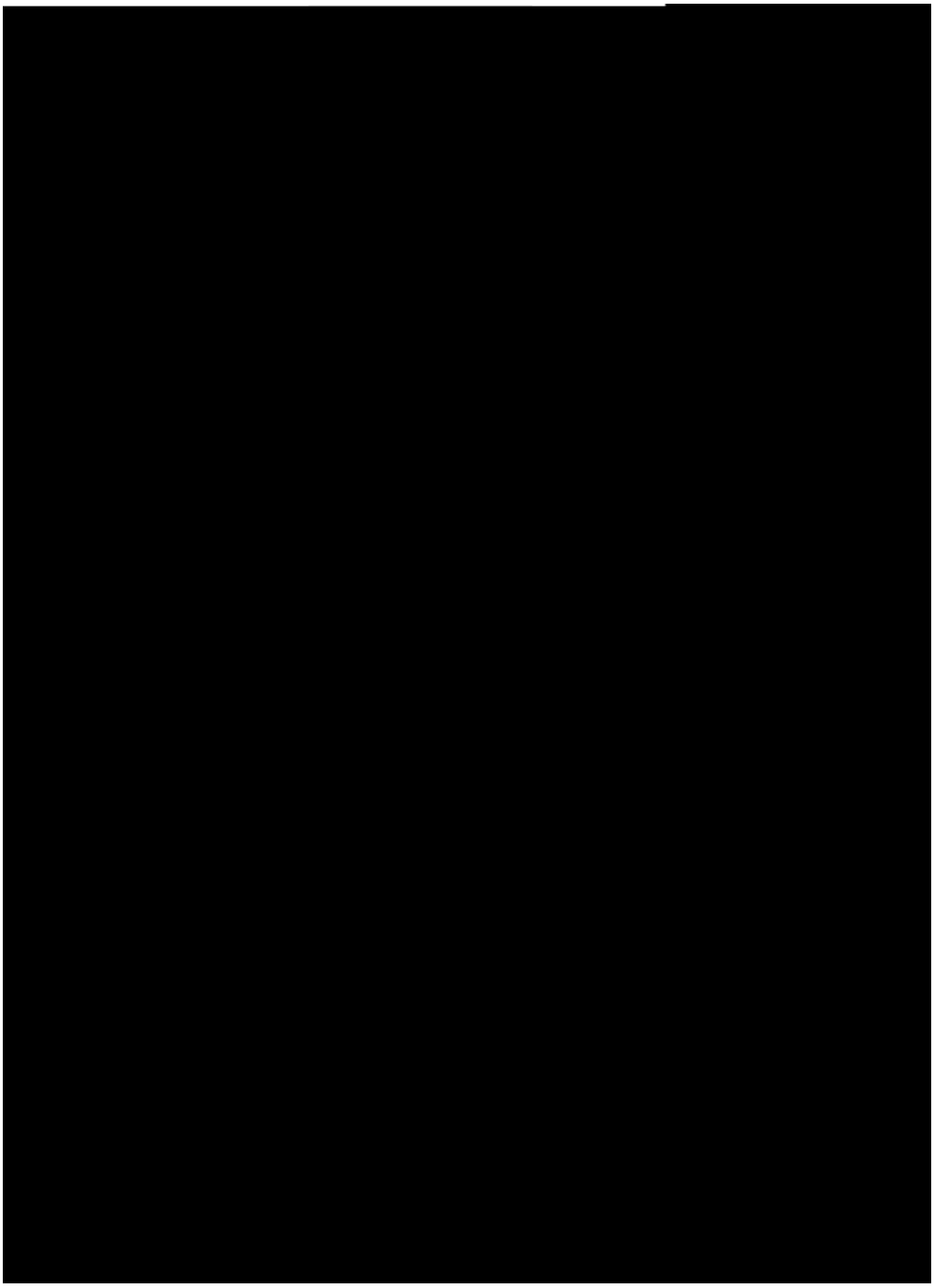
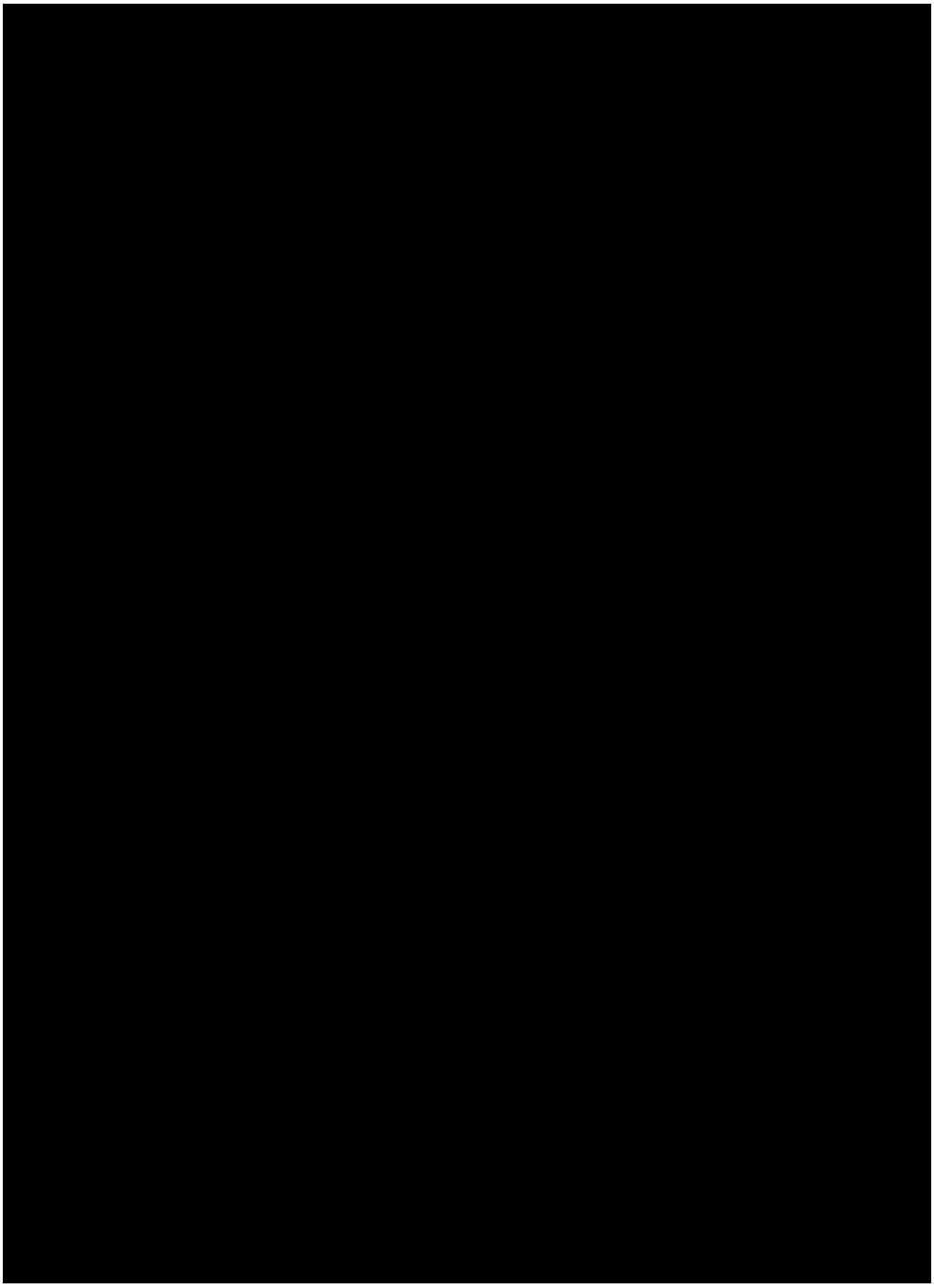


Figure 7.1: Class diagram for the Blast Record class representing all of the info in a BLASThe



```
...             print('****Alignment****')
...             print('sequence:', alignment.title)
...             print('length:', alignment.length)
...             print('e value:', hsp.expect)
...             print(hsp.query[0:75] + '...')
...             print(hsp.match[0:75] + '...')
...             print(hsp.sbjct[0:75] + '...')
```

If you also read the section 7.3 on parsing BLAST XML output, you'll notice that the above code is identical to what is found in that section. Once you parse something into a record class you can deal with it independent of the format of the original BLAST info you were parsing. Pretty snazzy!

Sure, parsing one record is great, but I've got a BLAST le with tons of records { how can I parse them all? Well, fear not, the answer lies in the very next section.

7.5.2 Parsing a plain-text BLAST le full of BLAST runs

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

- `item[1]` The id of the input record that caused the error. This is really useful if you want to record all of the records that are causing problems.

As mentioned, with each error generated, the BlastErrorParser will write the offending record to the specified `error_handle`. You can then go ahead and look and these and deal with them as you see fit.

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

```
>>> blat_qresult = SearchIO.read('my_blat.psl', 'blat-psl')
>>> print(blat_qresult)
Program: blat (<unknown version>)
  Query: mystery_seq (61)
         <unknown description>
 Target: <unknown target>
```

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

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)- [

Check out the HSP documentation

```
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

The last one is on strand and reading frame values. For strands, there are only four valid choices: `1` (plus strand), `-1`

need to access only a few of the queries. This is because parse

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` [illegible]

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 NCBI's databases such as PubMed, GenBank, GEO, and many others. You can access Entrez from a web browser to manually enter queries, or you can use Biopython's `Bio.Entrez` module for programmatic access to Entrez. The latter allows you for example to search PubMed or download GenBank records from within a Python script.

The `Bio.Entrez` module makes use of the Entrez Programming Utilities (also known as EUtils), consisting

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 database names accessible through the Entrez utilities:

```
>>resultesm(ai:)-125=h,tses
>>:tre.em(ai:)-125=:thr@example.com"paifaiwh-125Bioy3(y)5BioareEn(trez)]TJ0-11.955Td[(>>h,)-ccesm(ai:)-1
```

<DbName>unigene</DbName>
<DbName>unists</DbName>

```
                     EDKFLHLNYVSDLLIPHPIHLEILVQILQCRIKDVPSLHLLRLLFHEYHNLNSLITSK
                     KFIYAFSKRKKRFLWLLYNSYVYECEYLFQFLRKQSSYLRSTSSGVFLERTHLYVKIE
                     HLLVVCCNSFQRILCFLKDPFMHYVRYQGKAILASKGTLILMKKWKFHLVNFWQSYFH
                     FWSQPYRIHIKQLSNYSFSFLGYFSSVLENHLVVRNQMLENSFIINLLTKKFDTIAPV
                     ISLIGSLSKAQFCTVLGHPISKPIWTDFSDSDILDRFCRICRNLCRYHSGSSKKQVLY
                     RIKYILRLSCARTLARKHKSTVRTFMRRLGSGLLEEFFMEEE"
ORIGIN
        1 atttttacg aacctgtgga aatttttggt tatgacaata aatctagttt agtacttgtg
       61 aaacgtttaa ttactcgaat gtatcaacag aatttttga tttcttcggt taatgattct
      121 aaccaaaaag gattttgggg gcacaagcat tttttttctt ctcatttttc ttctcaaatg
      181 gtatcagaag gttttggagt cattctggaa attccattct cgtcgcaatt agtatcttct
      241 cttgaagaaa aaaaaatacc aaaatatcag aatttatc45(aaaaaataMattttc)-52aaaat(attctg)]TJ0-11.955Td301a
      61 atatctcggt

      61atatcttc atgaaatacc 5(tattcgtc)-525((aatcattct)-52t(cagtgtcact)]TJ-5.23-11.955Td[021at)-525(gac
```

The record variable consists of a Python list, one for each database in which we searched. Since we speci ed 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 will result in a `MemoryError` on many computers.

The XML file `Homo_sapiens.xml` consists of a list of Entrez gene records, each corresponding to one Entrez gene in human. `Entrez.parse` retrieves these gene records one by one. You can then print out or store the relevant information in each record by iterating over the records. For example, this script iterates over the Entrez gene records and prints out the gene numbers and names for all current genes:

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

...</Field></FieldList><DocsumList><Docsum><DsName>PubDate</DsName><DsType>4</DsType><DsTypeName>string</DsTypeName><

9.12.1 Parsing Medline records

```
...
A high level interface to SCOP and ASTRAL implemented in python.
```

9.12.2 Parsing GEO records

GEO (

```
PROTSIM     ORG=9986; PROTGI=126722851; PROTID=NP_001075655.1; PCT=76.90; ALN=288
...
PROTSIM     ORG=9598; PROTGI=114619004; PROTID=XP_519631.2; PCT=98.28; ALN=288

UpUNT      388
EQUENCEMACC=BC0267185.1; TIDg455013076; EQTYPE=mRNA.
EQUENCEMACC=NMP_0_0051.2; TIDg11629-5606; EQTYPE=mRNA.
EQUENCEMACC=D190025.1; TIDg2194166; EQTYPE=mRNA.
EQUENCEMACC=D190005.1; TIDg2194126; EQTYPE=mRNA.
EQUENCEMACC=BC0158785.1; TIDg146184206; EQTYPE=mRNA.
EQUENCEMACC=CR4079631.1; TIDg4711_5196; EQTYPE=mRNA.
EQUENCEMACC=BG5692935.1; EQTYPE=EST6;
...
EQUENCEMACC=AU0995345.1; EQTYPE=EST.
```

9.13 Using a proxy

NOTE - We've just done a separate search and fetch here, the NCBI much prefer you to take advantage of their history support in this situation. See Section 9.15.

Keep in mind that `records` is an iterator, so you can iterate through the records only once. If you want to save the records, you can convert them to a list:

```
>>> records = list(records)
```

Let's now iterate over the records to print out some information about each record:

```
>>> for record in records:
...     print("title:", record.get("TI", "?"))
...     print("authors:", record.get("AU", "?"))
...     print("source:", record.get("SO", "?"))
...     print("")
```

The output for this looks like:

```
title: Sex pheromone mimicry in the early spider orchid (ophrys sphegodes):
```

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

So, we expect to nd 814 Entrez Nucleotide records (this is the number I obtained in 2008; it is likely to increase in the future). If you nd 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:

```
814 from Bio imd[(>>>)-525(from)-525(BioTEnide)-[(really)-333(w)27(5(Bio)-52handleio)-52=5(BioTEnide).es
```

```
>>> 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"
>>> results = Entrez.read(Entrez.elink(dbfrom="pubmed", db="pmc",
...                                    LinkName="pubmed_pmc_refs", from_uid=pmid))
```

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
```

Because this is such a large input file, either way takes about eleven minutes on my new desktop computer (using the uncompressed `uniprot_sprot.dat` file as input).

It is equally easy to extract any kind of information you'd like from Swiss-Prot records. To see the members of a Swiss-Prot record, use

```
>>> dir(record)
['__doc__', '__init__', '__module__', 'accessions', 'annotation_update',
'comments', 'created', 'cross_references', 'data_class', 'description',
'entry_name', 'features', 'gene_name', 'host_organism', 'keywords',
'molecule_type', 'organelle', 'organism', 'organism_classification',
'references', 'sequence', 'sequence_length', 'sequence_update', 'taxonomy_id']
```

```
>>> 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 sulfides and 4 sulfur atoms of cysteines from the
protein.
...
```

10.2 Parsing Prosite records

```
>>> record.name
'PKC_PHOSPHO_SITE'
>>> record.pdoc
'PDOC00005'
```

and so on. If you're interested in how many Prosite records there are, you could use

```
>>> from Bio.ExPASy import Prosite
>>> handle = open("prosite.dat")
>>> records = Prosite.parse(handle)
>>> n = 0
>>> for record in records: n+=1
...
>>> n
2073
```

To read exactly one Prosite from the handle, you can use the `read` function:

```
>>> from Bio.ExPASy import Prosite
>>> handle = open("mysingleprositerecord.dat")
>>> record = Prosite.read(handle)
```

This function raises a ValueError if no Prosite record is found, and also if more than one Prosite record is found.

10.3 Parsing Prosite documentation records

In the Prosite example above, the `record.pdoc` accession numbers `'PDOC00001'`, `'PDOC00004'`, `'PDOC00005'` and so on refer to Prosite documentation. The Prosite documentation records are available from ExPASy as individual files, and as one file (`prosite.doc`) containing all Prosite documentation records.

We use the parser in `Bio.ExPASy.Prodoc` to parse Prosite documentation records. For example, to create a list of all accession numbers of Prosite documentation record, you can use

```
>>> from Bio.ExPASy import Prodoc
>>> handle = open("prosite.doc")
>>> records = Prodoc.parse(handle)
>>> accessions = [record.accession for record in records]
```

Again a

```
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  hyd
```

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'}
```

Other ScanProsite parameters can be passed as keyword arguments; see the

[

11.2 Coalescent simulation

11.2.1.2 Chromosome structure

We strongly recommend reading Fastsimcoal2 documentation to understand the full potential available in modeling chromosome structures. In this subsection we only discuss how to implement chromosome structures using the Biopython interface, not the underlying Fastsimcoal2 capabilities.

We will start by implementing a single chromosome, with 24 SNPs with a recombination rate immediately on the right of each locus of 0.0005 and a minimum frequency of the minor allele of 0. This will be speci ed by the following list (to be passed as second parameter to the function generate_simcoal_from_template):

toype and the 06some(on)1(d

npops Number of populations existing in nature. This is really a \guestimate". Has to be lower than 100.

In practice, when the number of populations is low, the mutation model is stepwise and the sample size increases, fdist will not be able to simulate an acceptable approximate average F_{st}.

To address that, a function is provided to iteratively approach the desired value by running several fdists in sequence. This approach is computationally more intensive than running a single fdist run, but yields good results. The following code runs fdist approximating the desired F_{st}:

```
sim_fst = ctrl.run_fdist_force_fst(npops = 15, nsamples = fd_rec.num_pops,
    fst = fst, sample_size = samp_size, mut = 0, num_sims = 40000,
    limit = 0.05)
```

The only new optional parameter, when comparing with run_fdist, is limit which is the desired maximum error. run_fdist can (and hould5) be eplacend with run

Chapter 12

Phylogenetics with Bio.Phylo

The Bio.Phylo module was introduced in Biopython 1.54. Following the lead of SeqIO and AlignIO, it aims

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

The

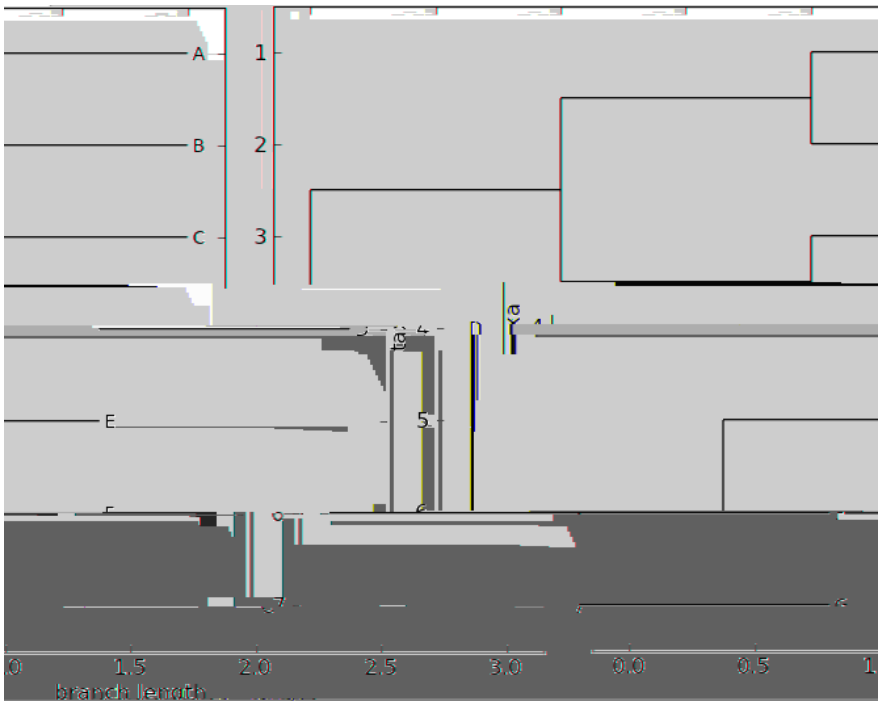


Figure 12.1: A rooted tree drawn with `Phylo.draw`.

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

```
>>> tree.clade[0, 1].color = "blue"
```

Finally, show our work (see Fig. 12.2):

```
>>> Phylo.draw(tree)
```

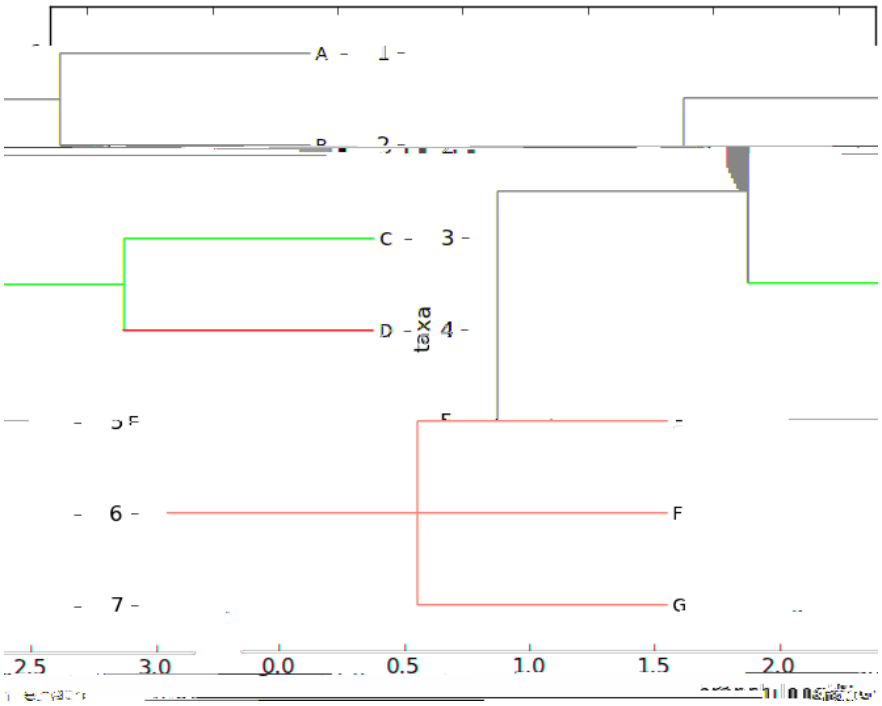


Figure 12.2: A colorized tree drawn with `Phylo.draw`.

14lthename(a)29 olorh | (a)29 hegure4(w)2thestanzard(a)29 output,ee45357oessethethatw7(ould(a)29 b)-c oesf30(orm9 w

Figure 12.7: A larger tree, using `neato`

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

12.4.2 Information methods

prune

12.6 PAML integration

Biopython 1.58 brought support for PAML (

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 12.4.4).

then we can create a Motif object as follows:

```
>>> 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)
```

13.2 Reading motifs

Creating motifs from instances by hand is a bit boring, so it's useful to have some I/O functions for reading

```
>MA0052.1 MEF2A
A  [ 1  0 57  2  9  6 37  2 56  6 ]
C  [50  0  1  1  0  0  0  0  0  0 ]
G  [ 0  0  0  0  0  0  0  0  2 50 ]
T  [ 7 58  0 55 49 52 21 56  0  2 ]
```

The motifs are read as follows:

```
>>> fh = open("jaspar_motifs.txt")
>>> for m in motifs.parse(fh, "jaspar"))
...     print(m)
TF name  Arnt
Matrix ID MA0004.1
Matrix:
        0      1      2      3      4      5
A:   4.00  19.00   0.00   0.00   0.00   0.00
C:  16.00   0.00  20.00   0.00   0.00   0.00
G:   0.00   1.00   0.00  20.00   0.00  20.00
T:   0.00   0.00   0.00   0.00  20.00   0.00



TF name  RUNX1
Matrix ID MA0002.1
Matrix:
        0      1      2      3      4      5      6      7      8      9     10
A:  10.00  12.00   4.00   1.00   2.00   2.00   0.00   0.00   0.00   8.00  13.00
C:   2.00   2.00   7.00   1.00   0.00   8.00   0.00   0.00   1.00   2.00   2.00
G:   3.00   1.00   1.00   0.00  23.00   0.00  26.00  26.00   0.00   0.00   4.00
T:  11.00  11.00  14.00  24.00   1.00  16.00   0.00   0.00  25.00  16.00   7.00



TF name  MEF2A
Matrix ID MA0052.1
Matrix:
        0      1      2      3      4      5      6      7      8      9
A:   1.00   0.00  57.00   2.00   9.00   6.00  37.00   2.00  56.00   6.00
C:  50.00   0.00   1.00   1.00   0.00   0.00   0.00   0.00   0.00   0.00
G:   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   2.00  50.00
T:   7.00  58.00   0.00  55.00  49.00  52.00  21.00  56.00   0.00   2.00
```

```
>>> 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(
```

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

****

To parse a TRANSFAC file, use

Table 13.2: Fields used to store references in TRANSFAC files

RN	Reference number
RA	Reference authors
RL	Reference data
RT	Reference title

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
```

13.6.2 Searching for matches using the PSSM score

It's just as easy to look for positions, giving rise to high log-odds scores against our motif:

```
>>> for position, score in pssm.search(test_seq, threshold=3.0):
...     print("Position %d: score = %5.3f" % (position, score))
...
Position 0: score = 5.622
Position -20: score = 4.601
Position 10: score = 3.037
Position 13: score = 5.738
Position -6: score = 4.601
```

The negative positions refer to instances of the motif found on the reverse strand of the test sequence, and follow the Python convention on negative indices. Therefore, the instance of the motif at `pos` is located at `test_seq[pos:pos+len(m)]os`

```
        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 /F34 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

```
>>> print("%f" % motif.pssm.mean(motif.background))
4.703928
```

as well as its standard deviation:

```
>>> print("%f" % motif.pssm.std(motif.background))
3.290900
```

and its distribution:

```
>>> 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
```

.command

13.10 Useful links

Sequence motif in wikipedia

PWM in wikipedia

Consensus sequence in wikipedia

Comparison of different motif finding programs

Chapter 14

Cluster analysis

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

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 ofo.ata[i,j]

`transpose` (default: `0`)
Determines if rows (`transpose` is `0`

{ 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: 2)
The number of clusters *k*.

`npass` (default: 1)
The number of times the *k*-medoids clustering algorithm is performed, each time with a di erent (random) initial condition. If `initialid` is given, the value of `npass` is ignored, as the clustering algorithm behaves deterministically in that case.

`initialid` (default: `None`)
Speci es the initial clustering to be used for the EM algorithm. If `initialid==None`, then a di erent random initial clustering is used for each of the `npass` runs of the EM algorithm. If `initialid` is not `None`

M-bedoids deterministic call.-12.9517Td [(Thei)-3394(funcion.-3394(returs)-339(a)-3394(tuple]TJ/F34 9.9626 T

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

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

An error is raised if `left` and `right` are not integers, or if `distance` cannot be converted to a oating-point value.

The Python class `Tree` represents a full hierarchical clustering solution. A `Tree` object can be created from a list of `Node` objects:

```
>>> from Bio.Cluster import Node, Tree
>>> nodes = [Node(1, 2, 0.2), Node(0, 3, 0.5), Node(-2, 4, 0.6), Node(-1, -3, 0.9)]
>>> tree = Tree(nodes)
>>> print(tree)
```

This guarantees that any `Tree` object is always well-formed.

To display a hierarchical clustering solution with visualization programs such as Java Treeview, it is better to scale all node distances such that they are between zero and one. This can be accomplished by

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

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

init is the initial value of as speci ed 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

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

`transpose` (default: `0`)
Determines if the centroids of the rows of `data` are to be calculated (`transpose==0`

transpose

14.8 Example calculation

Chapter 15

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

```
[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 15.2: 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*	309	-271.005880394

The logistic regression model classi es *yxcE*, *yxcD* as belonging to the same operon (class OP), while *yxiB*, *yxiA* are predicted to belong to di erent operons:

```
>>> print("yxcE, yxcD:", LogisticRegression.classify(model, [6, -173.143442352]))
yxcE, yxcD: 1
>>> print("yxiB, yxiA:", LogisticRegression.classify(model, [309, -271.005880394]))
yxiB, yxiA: 0
```

yxcE, 0

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

```
...
>>> x = [6, -173.143442352]
```

```
True: 1 Predicted: 1
True: 1 Predicted: 0
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 1
True: 0 Predicted: 0
```

Chapter 16

Graphics including GenomeDiagram

The `Bio.Graphics` module depends on the third party Python library ReportLab. Although focused on producing PDF files, ReportLab can also create encapsulated postscript (EPS) and (SVG) files. In addition

Figure 16.1: Simple linear diagram for *Yersinia pestis biovar Microtus* plasmid pPCP1.

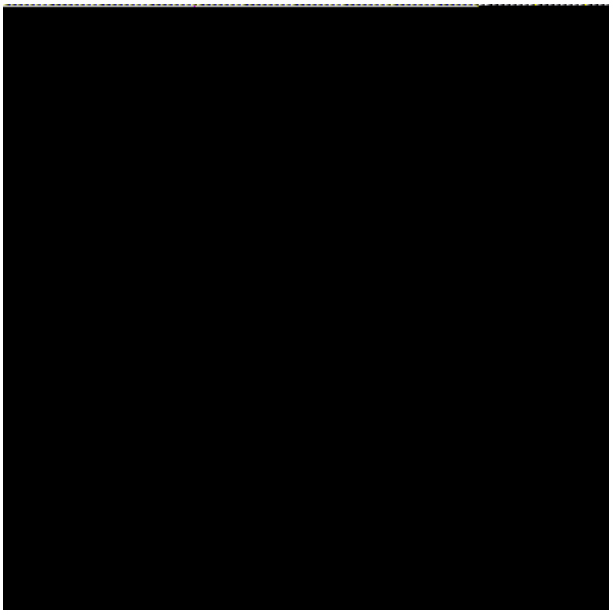


Figure 16.2: Simple circular diagram for *Yersinia pestis biovar Microtus* plasmid pPCP1.

16.1.4 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)
#(this for loop is the same as in the previous example)

#Create a track, and a diagram
gd_track_for_features = GenomeDiagram.Track(name="Annotated Features")
gd_diagram = GenomeDiagram.Diagram("Yersinia pestis biovar Microtus plasmid pPCP1")

#Now have to glue the bits together...
gd_track_for_features.add_set(gd_feature_set)
gd_diagram.add_track(gd_track_for_features, 1)
```

You can now call the `draw` and `write` methods as before to produce a linear or circular diagram, using the code at the end of the top-down example above. The figures should be identical.

16.1.5 Features without a SeqFeature

In the above example we used a `SeqRecord`'s `SeqFeature` objects to build our diagram (see also Section 4.3). Sometimes you

9-356N'tyohabo a

```
gds_features = gdt_features.new_set()

#Add three features to show the strand options,
feature = SeqFeature(FeatureLocation(25, 125), strand=+1)
gds_features.add_feature(feature, name="Forward", label=True)
```

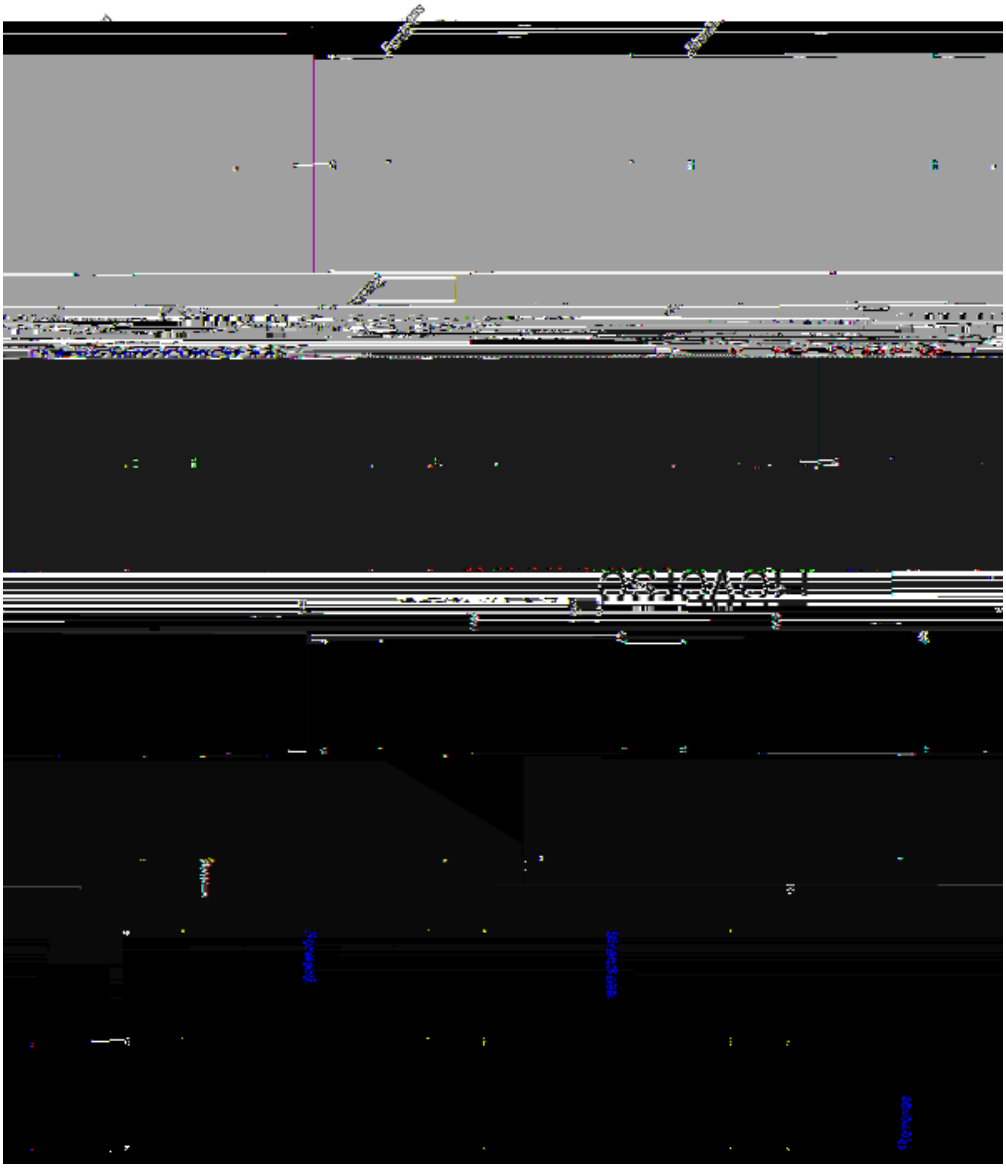


Figure 16.3: Simple GenomeDiagram showing label options. The top plot in pale green shows the default label settings (see Section 16.1.5) while the rest show variations in the label size, position and orientation (see Section 16.1.6).

16.1.7 Feature sigils

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

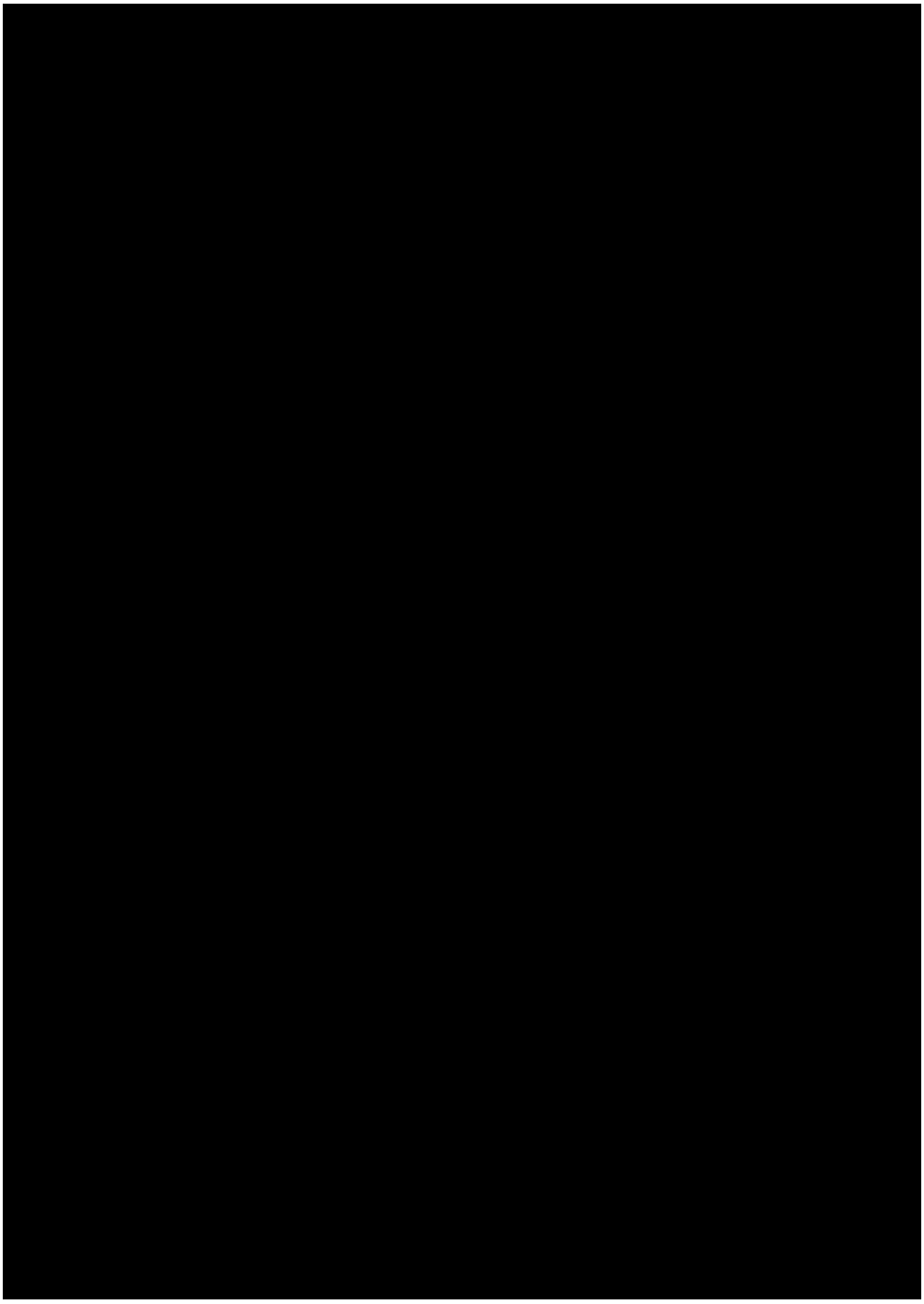


Figure 16.4: Simple GenomeDiagram showing di erent sigils (see Section 16.1.7)

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

```
                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),
```

Figure 16.8: Circular diagram for *Yersinia pestis biovar Microtus* plasmid pPCP1 showing selected restriction digest sites (see Section 16.1.9).

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

```
i+=1
```

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

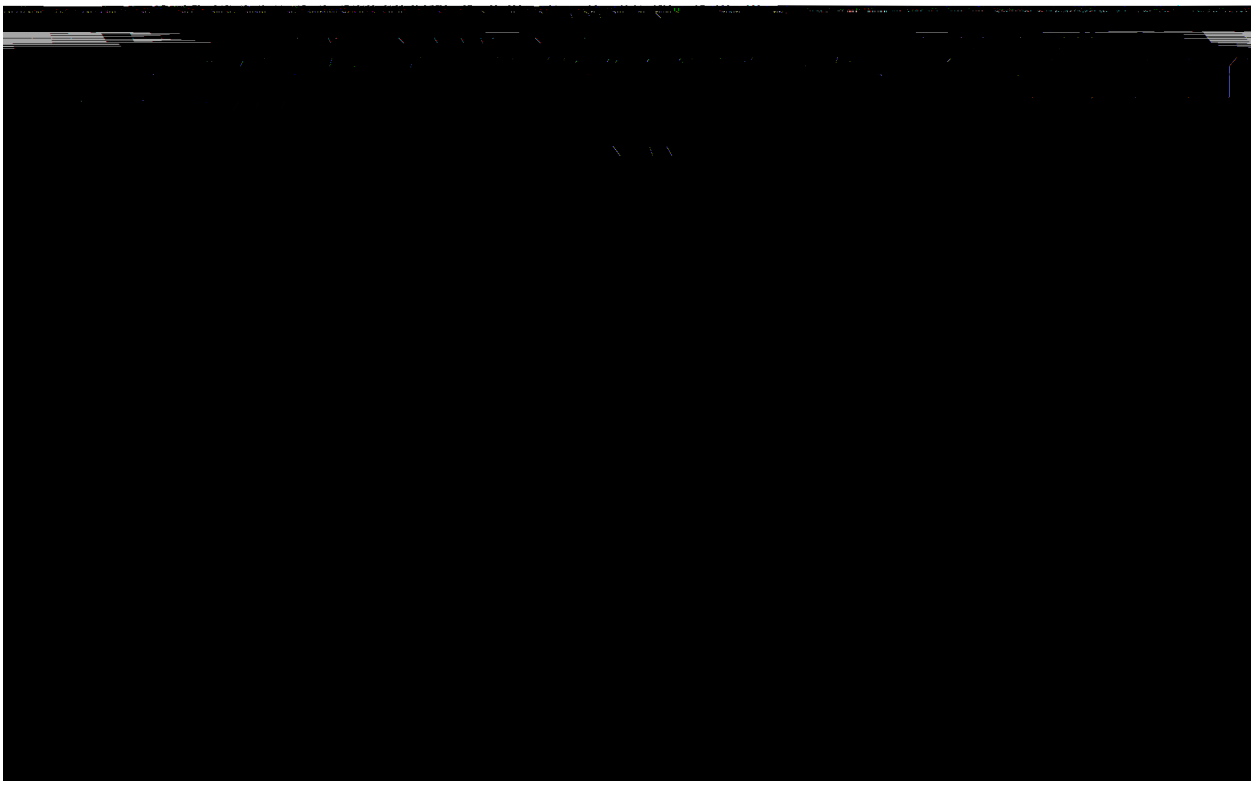


Figure 16.10: Linear diagram with three tracks for Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from *Listeria innocua* Clip11262 (NC_003212) plus basic cross-links shaded by percentage identity (see Section 16.1.11).

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` sigil, which allows us to further reduce the vertical space needed

reexample,eife(y)28(oe)-33(r)-898ur

These options are not covered here yet, so for now we refer you to theere(P

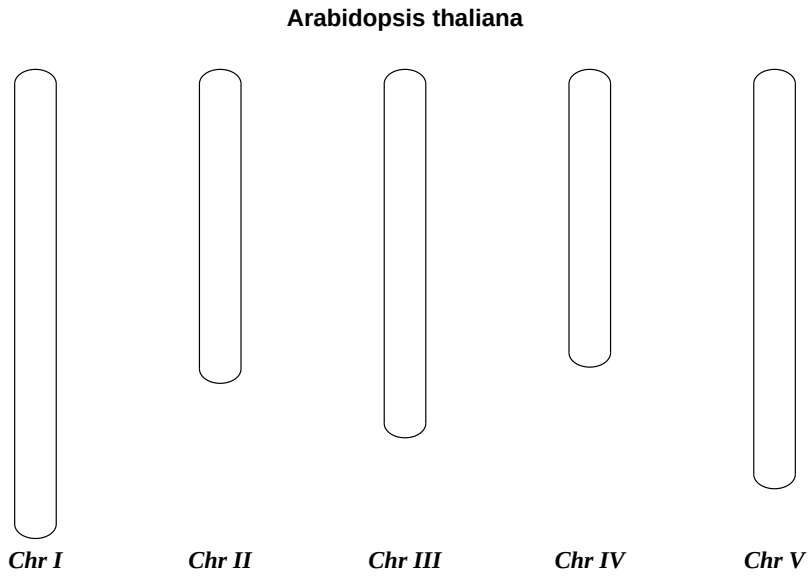


Figure 16.12: Simple chromosome diagram for *Arabidopsis thaliana*.

```
chr_diagram.draw("simple_chrom.pdc_ "Arabidopsis thaliana"w
```

should create a very "simpl PDF le, shown in Figure 16.12 is exaimpl is deliberately short sweet. next exaimpl shows the location of features of interest.

Annotated Cchroosomes

tinuing fhro the previous exaimpl, let's also show the tRNA genes. We'll get their locations by pars-

cabeGenBank Yeast Genome to download these les

```
_II/NC_003071.gbk"w,2Td0-11.9552Td[w(Cch IIIc_ "CHR_III/NC_003074.gbk"w,2Td0-11.9552Td[w(Cch IVc_ "CHR_IV/NC_003075.gb
```

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

Chapter 17

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.

Please note that the KEGG parser implementation in Biopython is incomplete. While the KEGG website hth06.15f the73.28 Tww.kegg.78(a) 9.9626 Tf 9626 Tf 99..798 0 Td [(ht73.28 TG2)-917w.kegg>>>

```
>>> request = REST.kegg_get("ec:5.4.2.2")
>>> open("ec_5.4.2.2.txt", 'w').write(request.read())
>>> records = Enzyme.parse(open("ec_5.4.2.2.txt"))
>>> record = list(records)[0]
>>> record.classname
['Isomerases;', 'Intramolecular transferases;', 'Phosphotransferases (phosphomutases)']
>>> record.entry
'5.4.2.2'
```

Now, here's a more realistic example which shows a combination of querying the KEGG API. This will demonstrate how to extract a unique set of all human pathway gene symbols which relate to DNA repair. The steps that need to be taken to do so are as follows. First, we need to get a list of all human pathways.

```
/list/hsa:10458+ece:Z5100          -> REST.kegg_list(["hsa:10458", "ece:Z5100"])
/find/compound/300-310/mol_weight  -> REST.kegg_find("compound", "300-310", "mol_weight")
/get/hsa:10458+ece:Z5100/aaseq     -> REST.kegg_get(["hsa:10458", "ece:Z5100"], "aaseq")
```

Chapter 18

Cookbook { Cool things to do with it

Personally I prefer the following version using a function to shu e the record and a generator expression instead of the for loop:

```
import random
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO

def make_shuffle_record(record, new_id):
    nuc_list = list(record.seq)
    random.shuffle(nuc_list)
    return SeqRecord(Seq("".join(nuc_list), record.seq.alphabet), \
           id=new_id, description="Based on %s" % original_rec.id)

original_rec = SeqIO.read("NC_005816.gb","genbank")
shuffled_recs = (make_shuffle_record(original_rec, "Shuffled%i" % (i+1)) \
                 for i in range(30))
handle = open("shuffled.fasta", "w")
SeqIO.write(shuffled_recs, handle, "fasta")
handle.close()
```

First we scan through the le once using `Bio.SeqIO.parse()`

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

This takes longer, as this time the output file contains all 41892 reads. Again, we're used a generator expression to avoid any memory problems. You could alternatively use a generator function rather than a generator expression.

```
from Bio import SeqIO
def trim_primers(records, primer):
```

```
trimmed_reads = trim_adaptors(original_reads, "GATGACGGTGT")
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
```

This means a wrong read (Pe= 1) gets a PHRED quality of 0, while a very good read likePe= 0:00001gets a PHRED quality of 50. While fo

18.1.10 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()` to iterate over the records in a *single* file, but in this case we have two input files. There are several strategies possible, but assuming

```
>>> fq_dict.keys()[:4]
['SRR020192.38240', 'SRR020192.23181', 'SRR020192.40568', 'SRR020192.23186']
```

18.1.13 Identifying open reading frames

```
table = 11
min_pro_len = 100

def find_orfs_with_trans(seq, trans_table, min_protein_length):
    answer = []
    seq_len = len(seq)
    for strand, nuc in [(+1, seq), (-1, seq.reverse_complement())]:
        for frame in range(3):
            trans = str(nuc[frame:].translate(trans_table))
            trans_len = len(trans)
            aa_start = 0
            aa_end = 0
            while aa_start < trans_len:
                aa_end = trans.find("*", aa_start)
                if aa_end == -1:
                    aa_end = trans_len
                if aa_end-aa_start >= min_protein_length:
                    if strand == 1:
                        start = frame+aa_start*3
                        end = min(seq_len,frame+aa_end*3+3)
                    else:
                        start = seq_len-frame-aa_end*3-3
                        end = seq_len-frame-aa_start*3
                    answer.append((start, end, strand,
                                   trans[aa_start:aa_end]))
                aa_start = aa_end+1
    answer.sort()
    return answer

orf_list = find_orfs_with_trans(record.seq, table, min_pro_len)
for start, end, strand, pro in orf_list:
    print("%s...%s - length %i, strand %i, %i:%i" \
          % (pro[:30], pro[-3:], len(pro), strand, start, end))
```

And the output:

```
NQIQGVICSPDSGEFMVTFETVMEIKILHK...GVA - length 355, strand 1, 41:1109
WDVKTVTGVLHHPFHLTFSLCPEGATQSGR...VKR - length 111, strand -1, 491:827
KSGELRQTPPASSTLHLRLILQRSGVMMEL...NPE - length 285, strand 1, 1030:1888
RALTGLSAPGIRSQTSCDRLRELRYVPVSL...PLQ - length 119, strand -1, 2830:3190
RRKEHVSKKRRPQKRPRRRRFFHRLRPPDE...PTR - length 128, strand 1, 3470:3857
```

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 16.1.9).

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 Theshe are an pbefule c

thr, complxs , wby of search stoines,hiach are up(p)-28(o

ulmede

Figure 18.1: Histogram of orchid sequence lengths.

18.2.2 Plot of sequence GC%

Another easily calculated quantity of a nucleotide sequence is the GC%. You might want to look at the GC% of all the genes in a bacterial genome for example, and investigate any o]ner1(s)-31whic(v)2h(s)-31coulandv7(ate)-31bl

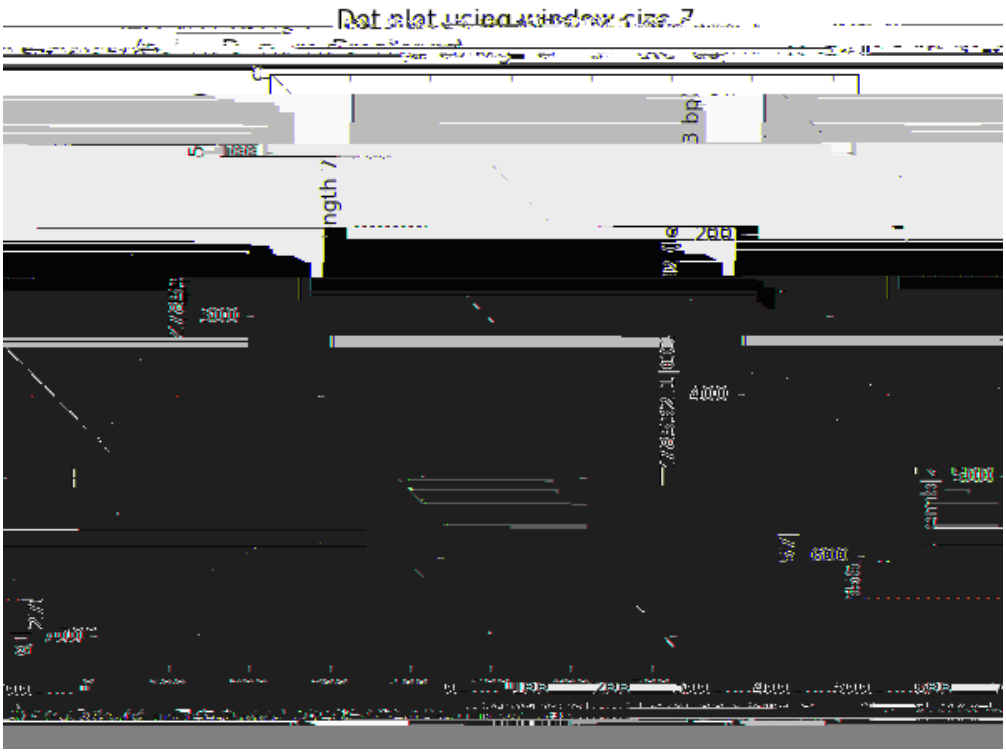


Figure 18.3: Nucleotide dot plot of two orchid sequence lengths (using pylab's imshow function).

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)
```

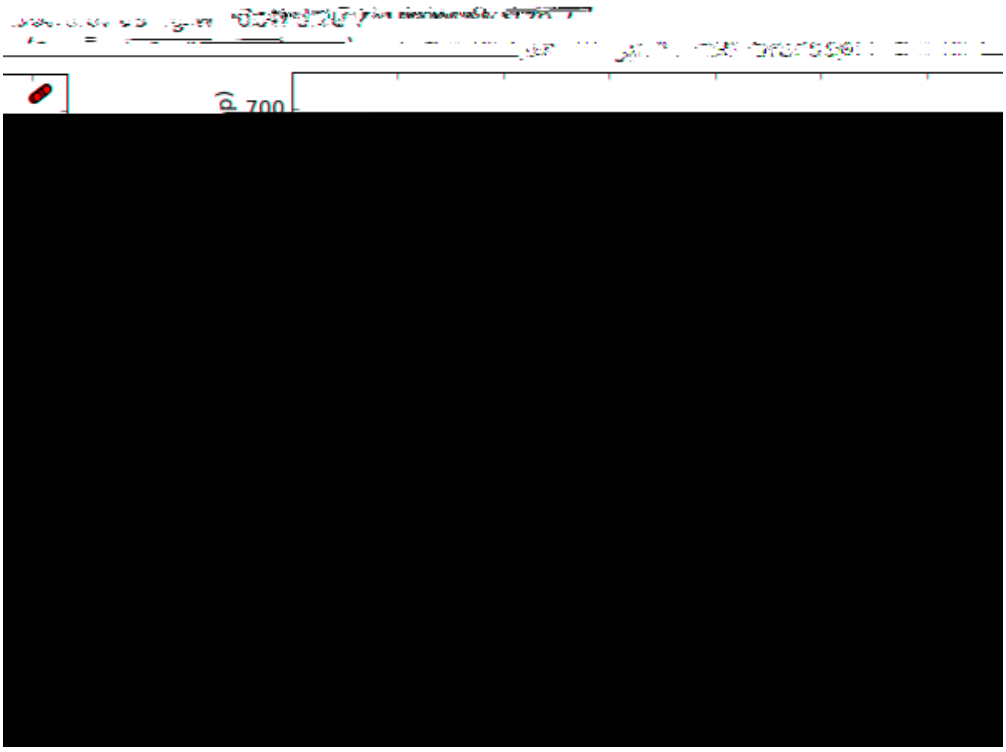


Figure 18.4: Nucleotide dot plot of two orchid sequence lengths (using pylab's scatter function).

```
import pylab
from Bio import SeqIO
for subfigure in [1,2]:
    filename = "SRR001666_%i.fastq" % subfigure
```

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

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

Chapter 19

The Biopython testing framework

Biopython has a regring teng [(Bmew)eng Td ngor34

Manually look at the file test_Biospam to make sure the output is correct. When you are sure it is all right and there are no bugs, you need to quickly edit the test_Biospam file so that the first line is: `test_Biospam' (no quotes).

copy the test_Biospam file to the directory Tests/output

(b) The quick way:

Run python run_tests.py -g test_Biospam.py. The regression testing framework is nifty

```
import unittest
from Bio import Biospam

class BiospamTestAddition(unittest.TestCase):

    def test_addition1(self):
        result = Biospam.addition(2, 3)
        self.assertEqual(result, 5)

    def test_addition2(self):
        result = Biospam.addition(9, -1)
        self.assertEqual(result, 8)

class BiospamTestDivision(unittest.TestCase):

    def test_division1(self):
        result = Biospam.division(3.0, 2.0)
        self.assertAlmostEqual(result, 1.5)

    def test_division2(self):
        result = Biospam.division(10.0, -2.0)
        self.assertAlmostEqual(result, -5.0)


if __name__ == "__main__":
    runner = unittest.TextTestRunner(verbosity = 2)
    unittest.main(testRunner=runner)
```

In the division tests, we use `assertAlmostEqual` insteadtEqualTd [(-11.955 Td [(resu463p_c103-20.923a.5))]TJ -20.92Td

Now let's check division ... ok

Note that if you want to write doctests involving file parsing, defining the file location complicates matters. Ideally use relative paths assuming the code will be run from the `Tests` directory, see the `Bio.SeqIO` doctests for an example of this.

To run the docstring tests only, use

```
$ python run_tests.py doctest
```

Chapter 20

Advanced

20.1 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. `data`: can be either a dictionary, or anr6FeqMat4(anr6instance36 Td [(i.)]TJ -34.205 Tf3.9484 -16.9936 Td [(i

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

The SeqMat constructor automatically generates a half-matrix, if a full matrix is passed. If a half

(a) acc_rep_mat: user provided accepted replacements matrix
(b) exp_freq_table

Summing up to 1.

When passing a dictionary as an argument, you should indicate whether it is a count or a frequency dictionary. Therefore the FreqTable class constructor requires two arguments: the dictionary itself, and FreqTable.COUNT or FreqTable.FREQ indicating counts or frequencies, respectively.

Chapter 21

Where to go from here { contributing to Biopython

21.5 Maintaining a distribution for a platform

21.7 Contributing Code

Chapter 22

Appendix: Useful stu about Python

If you haven't spent a lot of time programming in Python, many questions and problems that come up in using Biopython are often related to Python itself. This section tries to present some ideas and code that come up often (at least for us!) while using the Biopython libraries. If you have any suggestions for useful pointers that could go here, please contribute!

22.1 What the heck is a handle?

Handles are mentioned quite frequently throughout this documentation, and are also fairly confusing (at least to me!). Basically, you can think of a handle as being a \wrapper" around text information.

Handles provide (at least) two bene ts over plain text information:

1. They provide a standard way to deal with information stored in di erent ways. The text information can be in a le, or in a string stored in memory, or the output from a command line program, or at some remote website, but the handle provides a common way of dealing with information in all of these formats.

2. They allow text information to be read incrementally, instead of all at once. This is really important

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()
```

Bibliography

[1]

[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 di erentiation". *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] John W. Tukey: \Exploratory data analysis". Reading, Mass.: Addison-Wesley Pub. Co. (1977).

[33] Ka Yee Yeung, Walter L. Ruzzo: \Principal Component Analysis for clustering gene expression data". *Bioinformatics* 17 (9): 763{774 (2001). doi:10.1093/bioinformatics/17.9.763

[34] Alok Saldanha: \Java Treeview | extensible visualization of microarray data". *Bioinformatics* 20 (17): 3246{3248 (2004). http://dx.doi.org/10.1093/bioinformatics/bth349