

Biopython Tutorial and Cookbook

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

Last Update – 18 August 2011 (Biopython 1.58)

Contents

1	Introduction	7
1.1	What is Biopython?	7
1.2	What can I find in the Biopython package	7
1.3	Installing Biopython	8
1.4	Frequently Asked Questions (FAQ)	8
2	Quick Start – What can you do with Biopython?	9

4.3.1	SeqFeatures themselves	35
4.3.2	Locations	37
4.3.3	Sequence	38
4.4	Location testing	

7.3	Parsing BLAST output	87
7.4	The BLAST record class	89
7.5	Deprecated BLAST parsers	90
7.5.1	Parsing plain-text BLAST output	90
7.5.2	Parsing a plain-text BLAST file full of BLAST runs	93
7.5.3	Finding a bad record somewhere in a huge plain-text BLAST file	

10 Going 3D: The PDB module	132
10.1 Structure representation	132

13.2.2 AlignAce	171
13.3 Useful links	

16.3.1	Calculating summary information	214
16.3.2	Calculating a quick consensus sequence	214
16.3.3	Position Specific Score Matrices	215
16.3.4	Information Content	216
16.4	Substitution Matrices	218
16.4.1	Using common substitution matrices	218
16.4.2	Creating your own substitution matrix from an alignment	218
16.5	BioSQL – storing sequences in a relational database	219

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>)

12. *What file formats do Bio.SeqIO and Bio.AlignIO*

Chapter 2

Quick Start – What can you do with

followed by what you would type in:

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

What we have here is a sequence object with a *generic* alphabet - reflecting the fact we have *not* spec-

2.4 Parsing sequence file formats

A large part of much bioinformatics work involves dealing with the many types of file formats designed to hold biological data. These files are loaded with interesting biological data, and a special challenge is parsing these files into a format so that you can manipulate them with some kind of software.

2.6 What to do next

Chapter 3

Sequence objects

Biological sequences are arguably the central object in Bioinformatics, and in this chapter we'll introduce the Biopython mechanism for dealing with sequences, the Seq object. Chapter 4 will introduce the related SeqRecord

```
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq
Seq(' AGTACACTGGT', Al phabet())
>>> my_seq.al phabet
Al phabet()
```

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

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

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.

```
Seq('TAGCTAAGAC',obiUPACUnamb
```


3.8 Transcription


```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG", IUPAC.unambiguous_rna)
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
>>> messenger_rna.back_transcribe()
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
```

Note: The Seq object's `transcribe` and `back_transcribe` methods were added in Biopython 1.49. For older releases you would have to use the `Bio.Seq` module's functions instead, see Section [3.14](#).

3.9 Translation

3.10 Translation Tables

In the previous sections we talked about the Seq object translation method (and mentioned the equivalent function in the Bio.Seq module – see [Section 3.14](#))

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

So, what does Biopython do? Well, the equality test is the default for Python objects – it tests to see if

Chapter 4

Sequence Record objects

Chapter

annotations

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

location – The location of the SeqFeature

You could take the parent sequence, slice it to extract 5:18, and then take the reverse complement:

```
>>> feature_seq = example_parent[5:18].reverse_complement()  
>>> print feature_seq  
AGCCTTTGCCGTC
```

This is a simple example so this isn't too bad – however once you have to deal with compound features


```
>>> record
SeqRecord(seq=Seq(' TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG',
IUPACAmbiguousDNA()), id='NC_005816.1', name='NC_005816',
description='Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.',
dbxrefs=['Project: 10638'])
```

```
>>> len(record)
```

```
9609
```

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

```
29
```

For this example we're going to `foc1ca31n1(goinon1(gointheTd[(2mBT/F359.9205.70760511.91pimTd[(29)]TJ/F89.968.6`

Chapter 5

Sequence Input/Output

In this chapter we'll discuss in more detail the Bio.SeqIO module, which was briefly introduced in Chapter 2 and also used in Chapter 4

```
from Bio import SeqIO
for seq_record in SeqIO.parse("Is_orchid.fasta", "fasta"):
```


5.2 Parsing sequences from the net

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 20.1), and in this section we'll use handles to internet connections to download sequences.

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

5.2.1 Parsing GenBank records from the net

Section 8.6 talks about the Entrez EFetch interface in more detail, but for now let's

```

handle = Entrez.efetch(db="nucleotide", rettype="gb", \
                        id="6273291,6273290,6273289")
for seq_record in SeqIO.parse(handle, "gb"):
    print seq_record.id, seq_record.description[:50] + "... "
    print "Sequence length %i," % len(seq_record),
    print "%i features" % len(seq_record.features)

```

- `Bio.SeqIO.to_dict()` is the most flexible but also the most memory demanding option (see Section 5.3.1)

5.3.1.1 Specifying the dictionary keys

This should give:

Z78533.1 JUeWn6DPhgZ9nAyowsgtoD9TTo

Z78532.1 MN/s0q9zDoCvEEc+k/I FwCNF2pY

...

Z78439.1 H+JfaShya/4yyAj 7I bMqgNkxdxQ

Now, recall the


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

5.3.2.1 Specifying the dictionary keys

SeqRecord will write them to a FASTA format like:

```
rec3 = SeqRecord(Seq("MVTVEEFRAQCAEGPATVMAI GTATPSNCVDQSTYQI YFRI TNSEHKVELKEKFKRMC" =
```


Chapter 6

Multiple Sequence Alignment objects

This chapter is about Multiple Sequence Alignments, by which we mean a collection of multiple sequences which have been aligned together – usually with the insertion of gap characters, and addition of leading or trailing gaps – such that all the sequence strings are the same length. Such an alignment can be regarded as a matrix of letters, where each row is held as a SeqRecord object internally.

We will introduce the MultipleSeqAlignment object which holds this kind of data, and the Bio.AlignIO

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

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

```
from Bio import AlignIO
alignments = AlignIO.parse("resampled.phy", "phylip")
for alignment in alignments:
    print alignment
    print
```

This would give the following output, again abbreviated for display:

```
SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAACCA Alpha
AAACCC Beta
ACCCCA Gamma
CCCAAC Delta
CCCAAA Epsilon
```

```
SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAACAA Alpha
AAACCC Beta
ACCCAA Gamma
CCCACC Delta
CCCAAA Epsilon
```

```
SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAAAAC Alpha
AAACCC Beta
AACAAC Gamma
CCCCCA Delta
CCCAAC Epsilon
```

...

```
SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAAACC Alpha
ACCCCC Beta
AAAACC Gamma
CCCCAA Delta
CAAACC Epsilon
```

As with the function `Bio.SeqIO.parse()`, using `Bio.AlignIO.parse()` returns an iterator. If you want to keep all the alignments

Its more common to want to load an existing alignment, and save that, perhaps after some simple manipulation like removing certain rows or columns.

Q9T008_BPI KE/1-52
COATB_BPI 22/32-83
COATB_BPM13/24-72

RA
KA
KA

First of all, in some senses the alignment objects act like a Python list of SeqRecord objects (the rows). With this model in mind hopefully the actions of len() (the number of rows) and iteration (each row as a SeqRecord) make sense:

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> print "Number of rows: %i" % len(alignment)
Number of rows: 7
>>> for record in alignment:
...     print "%s - %s" % (record.seq, record.id)
AEPNAATNYATEAMDSLKTQAI DLI SQTWPVVTTVVAGLVI RLFKKFSSKA - COATB_BPI KE/30-81
AEPNAATNYATEAMDSLKTQAI DLI SQTWPVVTTVVAGLVI KLFKKFVSRA - Q9T0Q8_BPI KE/1-52
DGTSTATSYATEAMNSLKQATDLI DQTWPVVTSTAVAGLAI RLFKKFSSKA - COATB_BPI 22/3-E57("%s)-525(-)a525(%) -525(-)----A
```


DGTSTAATEAMNSLKTQATDLI DQTWPVVTSVAVAGLAI RLFKKFSSKA COATB_BPI 22/32-83
AEGDDPAKAAFNSLQASATEYI GYAWAMVVVI VGATI GI KLFKKFTSKA COATB_BPM13/24-72
AEGDDPAKAAFDSLQASATEYI GYAWAMVVVI VGATI GI KLFKKFASKA COATB_BPZJ2/1-49
AEGDDPAKAAFDSLQASATEYI GYAWAMVVVI VGATI GI KLFKKFTSKA Q9T0Q9_BPF1/1-49
FAADDAKAAAFDSLTAQATEMSGYAWALVVLVGATVGI KLFKKFVSRA COATB_BPI F1/22-73

Another common use of alignment addition would be to combine alignments for several different genes

6.4.1 ClustalW


```
>>> from Bio.Align.Applications import MuscleCommandline
>>> muscle_cline = MuscleCommandline(clwstrict=True)
>>> print muscle_cline
muscle -051
```

```
>>> 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
```


For more about the optional BLAST arguments, we refer you to the NCBI's own documentation, or that built into Biopython:

```
>>> from Bio.Blast import NCBIWWW
>>> help(NCBIWWW.qblast)
...
```

Note that the default settings on the NCBI BLAST website are not quite the same as the defaults on QBLAST. If you get different results, you'll need to check the parameters (e.g. the expectation value

After doing this, the results are in the file

This section will show briefly how to use these tools from within Python. If you have already read or tried the alignment tool examples in Section 6.4 this should all seem quite straightforward. First, we construct a

versions of BLAST. Not only is the XML output more stable than the plain text and HTML output, it is


```

...         print 'e value:', hsp.expect
...         print hsp.query[0:75] + '...'
...         print hsp.match[0:75] + '...'
...         print hsp.subject[0:75] + '...'

```

This will print out summary reports like the following:

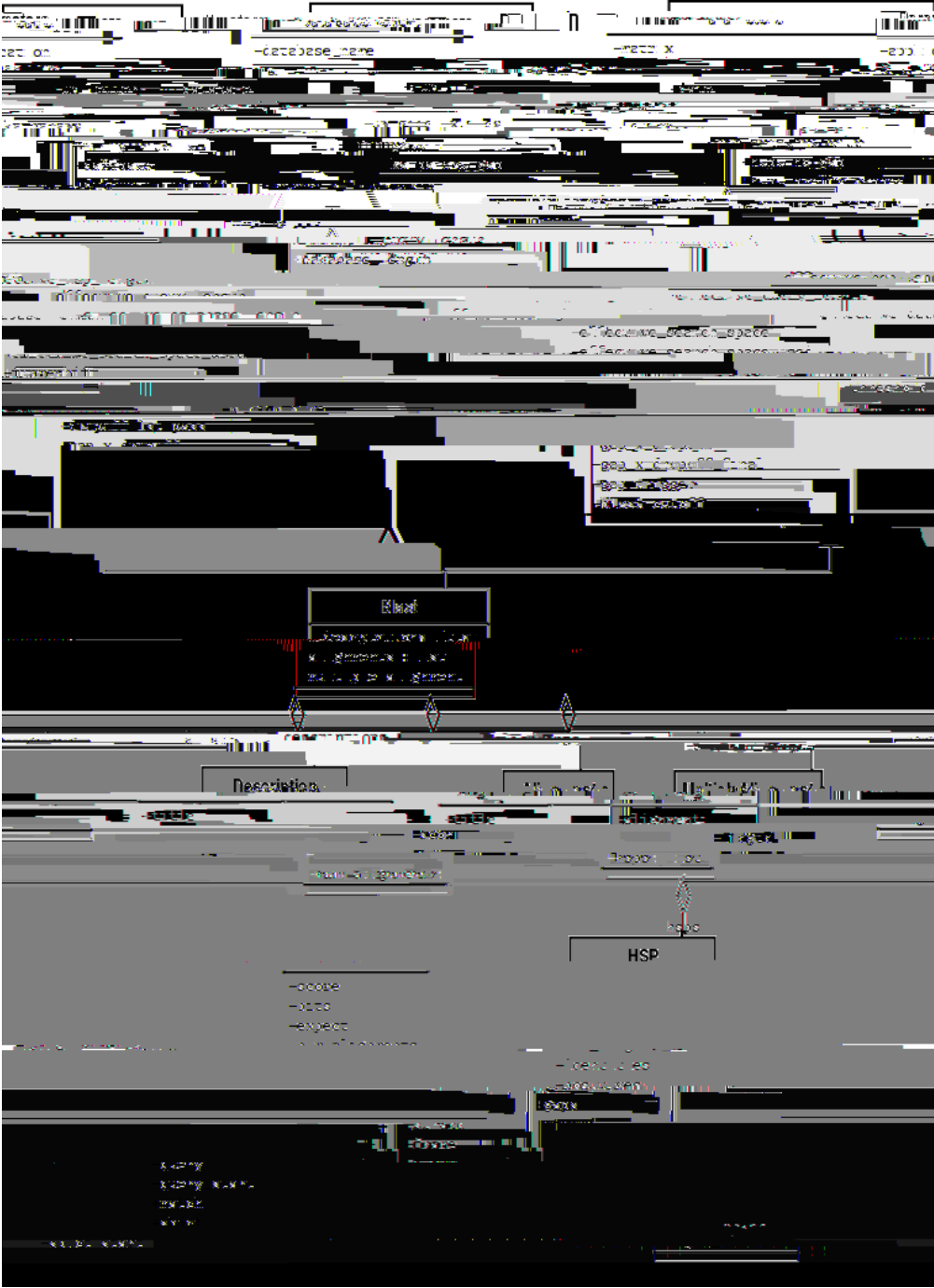
```

****Alignment****
sequence: >gb|AF283004.1|AF283004 Arabidopsis thaliana cold acclimation protein WCOR413-like protein
alpha form mRNA, complete cds
length: 783
e value: 0.034
tacttgttgatattggatcgaacaaactggagaaccaacatgctcacgtcacttttagtcccttacatattcctc...
||||||| | ||||||||| || |||| | | ||||||| ||||| | | ||||||| ||| ||...
tacttgttggtgttgatcgaaccaattggaagacgaatatgctcacatcacttctcattccttacatcttctc...

```

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

An important consideration for extracting information from a BLAST report is the type of objects that



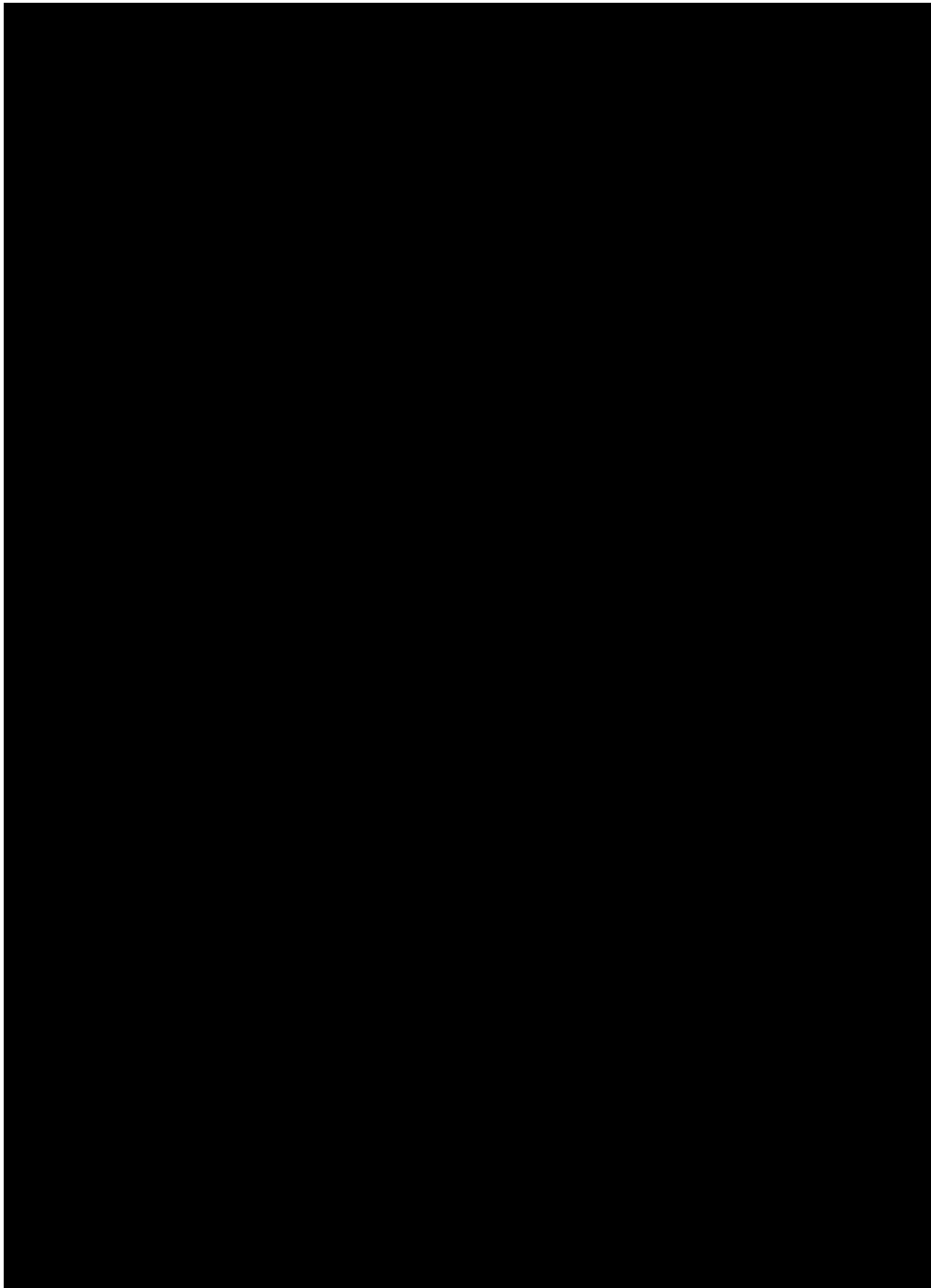


Figure 7.2: Class diagram for the PSIBlast Record class.

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

8.2 EInfo: Obtaining information about the Entrez databases

```
      <DbName>uni gene</DbName>
      <DbName>uni sts</DbName>
</DbLi st>
</el nfoResul t>
```


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

EDKFLHLNYVSDLLI PHPI HLEI LVQI LQCRI KDVPSLHLLRLLFHEYHNLNSLI TSK


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

The "LinkSetDb"


```
>>> from Bio import Entrez
>>> handle = open("Homo_sapiens.xml")
>>> records = Entrez.parse(handle)

>>> for record in records:
...     status = record['Entrezgene_track-info']['Gene-track']['Gene-track_status']
...     if status.attributes['value']=='discontinued':
...         continue
...     geneid = record['Entrezgene_track-info']['Gene-track']['Gene-track_geneid']
...     genename = record['Entrezgene_gene']['Gene-ref']['Gene-ref_locus']
...     print geneid, genename
```

This will print:

```
1 A1BG
2 A2M
3 A2MP
8 AA
9 NAT1
10 NAT2
11 AACP
```

When your file is in the XML format but is corrupted (for example, by ending prematurely), the parser will raise a `CorruptedXMLError`. Here is an example of an XML file that ends prematurely:


```
>>> from Bio import Geo
>>> handle = open("GSE16.txt")
>>> records = Geo.parse(handle)
>>> for record in records:
...     print record
```

You can search the "gds" database (GEO datasets) with ESearch:

```
SEQUENCE ACC=D90042.1; NID=g219415; PID=g219416; SEQTYPE=mRNA
SEQUENCE ACC=D90040.1; NID=g219411; PID=g219412; SEQTYPE=mRNA
SEQUENCE ACC=BC015878.1; NID=g16198419; PID=g16198420; SEQTYPE=mRNA
SEQUENCE ACC=CR407631.1; NID=g47115198; PID=g47115199; SEQTYPE=mRNA
SEQUENCE ACC=BG569293.1; NID=g13576946; CLONE=IMAGE: 4722596; END=5' ; LID=6989; SEQTYPE=EST; TRACE=44
...
SEQUENCE ACC=AU099534.1; NID=g13550663; CLONE=HSI 08034; END=5' ; LID=8800; SEQTYPE=EST
//
```

```
import os
os.environ["http_proxy"] = "http://proxyhost.example.com:8080"
```

See the [urllib documentation](#)

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

The output for this looks like:

```
>>> from Bio import Entrez
>>> handle = Entrez.esearch(db="nucleotide", term="Cypripedium", retmax=814)
>>> record = Entrez.read(handle)
```

Here,

However, you also get given two additional pieces of information, the WebEnv session cookie, and the QueryKey:

```
>>> webenv = search_results["WebEnv"]  
>>> query_key = search_results["QueryKey"]
```

Having stored these values in variables `session_cookie` and `query_key` we can use them as parameters to `Bio.Entrez.efetch()`

```
fetch_handle.close()
out_handle.write(data)
out_handle.close()
```

At the time of writing, this gave 28 matches - but because this is a date dependent search, this will of course vary. As described in Section 8.12.1 above, you can then use Bio.Medline to parse the saved records.

8.15.3 Searching for citations

Back in Section 8.7 we mentioned ELink can be used to search for citations of a given paper. Unfortunately this only covers journals indexed for PubMed Central (doing it for all the journals in PubMed would mean a lot more work for the NIH). Let's try this for the Biopython PDB parser paper, PubMed ID 14630660:

```
>>> 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 9

Swiss-Prot and ExPASy

9.1 Parsing Swiss-Prot files

Swiss-Prot (<http://www.expasy.org/>)

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

```
>>> from Bio720bort(Bio7SwissProt)]TJ1-11.95510373Td[(>>>)-descriptions(>>>)-=(>>>)-[]
>>>>>>>>
>>>>>>Bio72n(Bio7SwissProt.parse(handle:)]TJ1-11.95510373...
>>>
```

```
>>> from Bio.SwissProt import KeyWList
>>> handle = open("keywlist.txtk41st
>>> = KeyWLi.parseseen(handtk41st)]TJ0-11.9251Td[(>>>)-52for>>
```



```

CC  -! - Also hydrolyzes diacylglycerol .
PR  PROSITE; PDOC00110;
DR  P11151, LIPL_BOVIN ; P11153, LIPL_CAVPO ; P11602, LIPL_CHICK ;
DR  P55031, LIPL_FELCA ; P06858, LIPL_HUMAN ; P11152, LIPL_MOUSE ;
DR  046647, LIPL_MUSVI ; P49060, LIPL_PAPAN ; P49923, LIPL_PIG ;
DR  Q06000, LIPL_RAT ; Q29524, LIPL_SHEEP ;
//

```

In this example, the first line shows the EC (Enzyme Commission) number of lipoprotein lipase (sec-

9.5 Accessing the ExPASy server


```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prosite_entry('PS00001')
>>> html = handle.read()
>>> output = open("myprosite_record.html", "w")
>>> output.write(html)
>>> output.close()
```

6

```
>>> result[0]
```

```
{'signature_ac': u'PS50948', 'level': u'0', 'stop': 98, 'sequence_ac': u'USERSEQ1', 'start': 16, 'score': 1.0}
```

```
>>> result[1]
```

Chapter 10

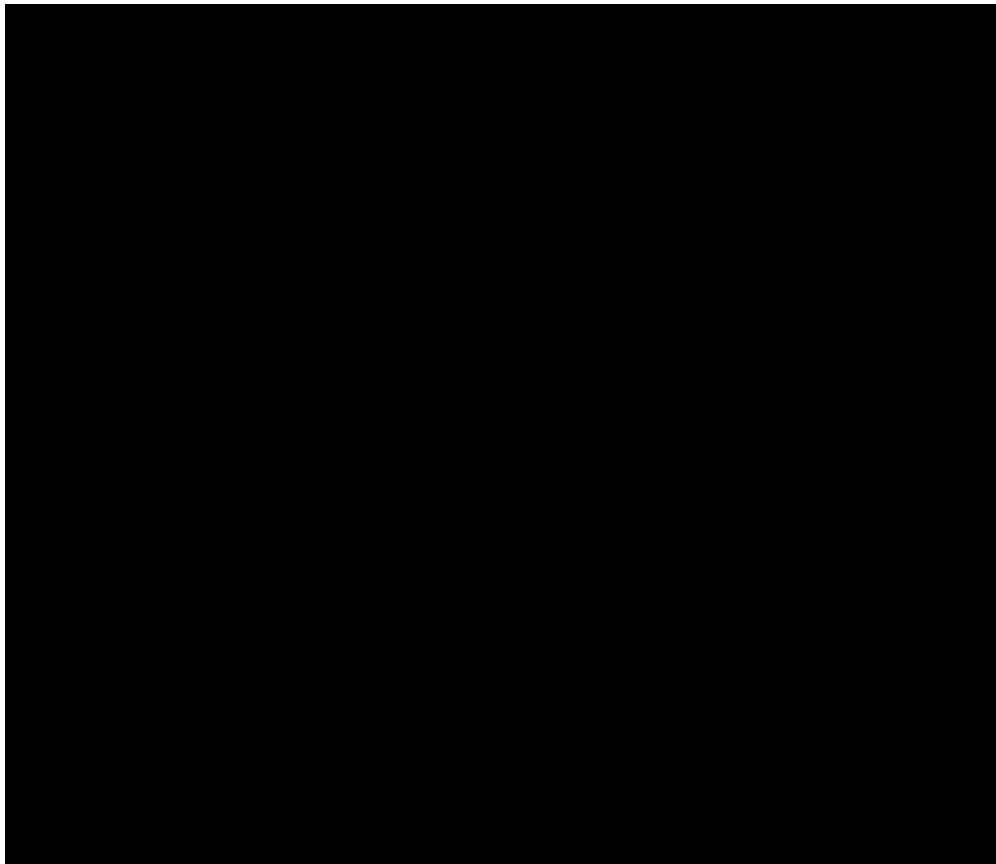


Figure 10.1: UML diagram of the SMCRA data structure used to represent a macromolecular structure.

```
full_id=residue.get_full_id()  
print full_id  
("1abc", 0, "A", (" ", 10, "A"))
```

This corresponds to:

- The Structure with id "1abc"
- The Model with id 0
- The Chain with id "A"
- The Residue with id (" ", 10, "A").

The

```
filename="pdb1fat.ent"
```

```
s=p.get_structure(structure_id, filename)
```

The PERMISSIVE flag indicates that a number of common problems (see

10.2 Disorder

10.5.1.1 Duplicate residues

One structure contains two amino acid residues in one chain with the same sequence identifier (resseq 3) and icode. Upon inspection it was found that this chain contains the residues Thr A3, ..., Gly A202, Leu A3, Glu A204. Clearly, Leu A3 should be Leu A203. A couple of similar situations exist for structure 1FFK

Chapter 11

Bio.PopGen: Population genetics

11.2 Coalescent simulation

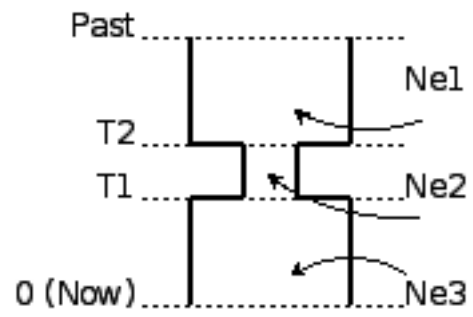


Figure 11.1: A bottleneck

In practice, when the number of populations is low, the mutation model is stepwise and the sample size

Chapter 12

Phylogenetics with Bio.Phylo


```

      Clade(branch_length=1.0, name="C")
      Clade(branch_length=1.0, name="D")
    Clade(branch_length=1.0)
      Clade(branch_length=1.0, name="E")
      Clade(branch_length=1.0, name="F")
      Clade(branch_length=1.0, name="G")

```

The Tree object contains global information about the tree, such as whether it is rooted. It has one root clade, and under it is a nested list of clades all the way down to tips.

The function `draw_ascii`


```
<phy: phyloxml xmlns:phy="http://www.phyloxml.org">
  <phy: phylogeny rooted="true">
    <phy: clade>
      <phy: branch_length>1.0</phy: branch_length>
      <phy: color>
        <phy: red>128</phy: red>
        <phy: green>128</phy: green>
        <phy: blue>128</phy: blue>
      </phy: color>
    </phy: clade>
    <phy: clade>
      <phy: branch_length>1.0</phy: branch_length>
      <phy: clade>
        <phy: branch_length>1.0</phy: branch_length>
        <phy: clade>
          <phy: name>A</phy: name>
          ...
        </phy: clade>
      </phy: clade>
    </phy: clade>
  </phy: phylogeny>
</phy: phyloxml>
```

```
>>> Phyl o.convert("tree1.xml", "phyl oxml", "tree1.dnd", "newi ck")
1
>>> Phyl o.convert("other_trees.xml", "phyl oxml", "other_trees.nex", 'nexus')
12
```

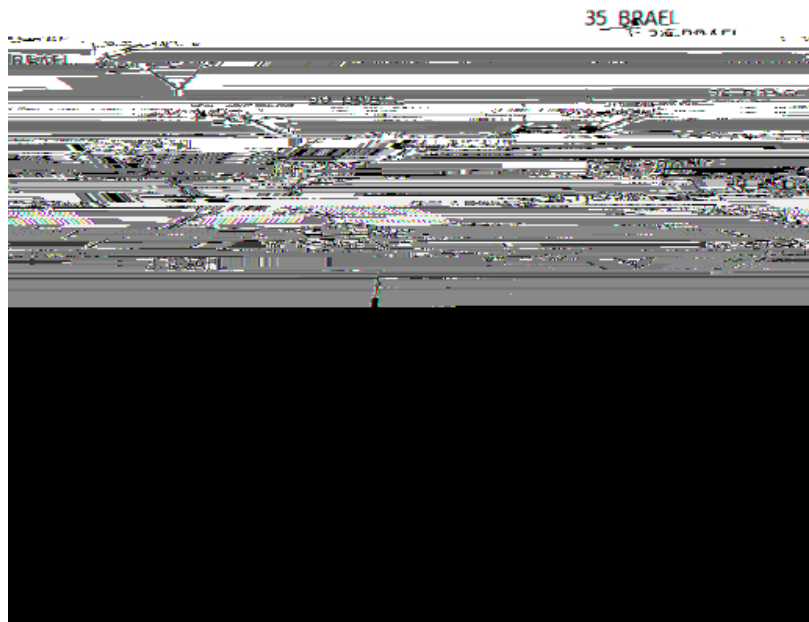



Figure 12.4: A larger tree, using neato

These both wrap a method with full control over tree traversal, `find_clades`. Two more traversal methods, `find_elements` and `find_any`


```
>>> from Bio import Phylo
>>> from Bio.Phylo.Applications import PhymlCommandline
>>> cmd = PhymlCommandline(input='Tests/Phylip/random.phy')
>>> out_log, err_log = cmd()
```

This generates a tree file and a stats file with the names *[input filename]_phyml_tree.txt* and *[input filename]_phyml_stats.txt*. The tree file is in Newick format:

Detailed documentation for this new module currently lives on the Biopython wiki: <http://biopython.org/wiki/PAML>

12.7 Future plans

Bio.Phylo is under active development. Here are some features we might add in future releases:

New methods Generally useful functions for operating on Tree or Clade objects appear on the Biopython wiki first, so that casual users can test them and decide if they're useful before we add them to Bio.Phylo:

Chapter 13

Sequence motif analysis using Bio.Motif

This chapter gives an overview of the functionality of the Bio.Motif package included in Biopython distribution. his ineornpeesaresvvdn(in)-304(anayi)1(s)-1his of s,n sl'lln (s)-1sumef thtfarde


```
' G' : -2.3219280948873622},  
{ ' A' : 1.7655347463629771,  
  ' C' : -2.3219280948873622,  
  ' T' : -2.3219280948873622,  
  ' G' : -2.3219280948873622},  
{ ' A' : -2.3219280948873622,  
  ' C' : -2.3219280948873622,  
  ' T' : 1.7655347463629771,  
  ' G' : -2.3219280948873622},  
{ ' A' : 1.3785116232537298,  
  ' C' : -2.3219280948873622,  
  ' T' : 0.0,  
  ' G' : -2.3219280948873622},  
{ ' A' : 1.7655347463629771,  
  ' C' : -2.3219280948873622,  
  ' T' : -2.3219280948873622,  
  ' G' : -2.3219280948873622}  
]
```

```
>>> arnt.make_counts_from_instances()
```



```
test_seq=Seq("TATGATGTAGTATAATATAATTATAA", m. al phabet)
```

or a threshold satisfying (roughly) the equality between the false-positive rate and the $-\log$ of the

bATTA
TATAA

There are two other functions: `dist_dpq`

13.2.2 AlignAce

We can do very similar things with AlignACE program. Assume, you have your output in the file

The logistic regression model gives us appropriate values for the parameters β_0 , β_1 , β_2 using two sets of

```

[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]]
>>> ys = [1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
0,
0,
0,
0,
0,
0,
0]
>>> model = LogisticRegression.train(xs, ys)

```

Here, xs and ys are the training data: xs contains the predictor variables for each gene pair, and ys

Iteration: 2 Log-likelihood function: -5.76877209868
Iteration: 3 Log-likelihood function: -5.11362294338
Iteration: 4 Log-likelihood function: -4.74870642433
Iteration: 5 Log-likelihood function: -4.50026077146
Iteration: 6 Log-likelihood function: -4.31127773737
Iteration: 7 Log-likelihood function: -4.16015043396
Iteration: 8 Log-likelihood function: -4.03561719785
Iteration: 9 Log-likelihood function: -3.93073282192
Iteration: 10 Log-likelihood function: -3.84087660929
Iteration: 11 Log-likelihood function: -3.76282560605
Iteration: 12 Log-likelihood function: -3.69425027154
Iteration: 13 Log-likelihood function: -3.6334178602
Iteration: 14 Log-likelihood function: -3.57900855837
Iteration: 15 Log-likelihood function: -3.52999671386

0, corresponding to class OP and class NOP, respectively. For example 1(sp)-2et'ss t

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 14.1.

14.2.2 Initializing a k

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

By default, all neighbors are given an equal weight.

To find out how confident we can be in these predictions, we can call the `calculate` function, which

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

The leave-one-out analysis shows that k -nearest neighbors model is correct for 13 out of 17 gene pairs, which

Chapter 15

Graphics including GenomeDiagram

The Bio. Graphics

15.1.3 A top down example

15.1.4 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")
g = GenomeDiagram.FromSeq(record)
```

```
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)
feature = SeqFeature(FeatureLocation(150, 250), strand=None)
gds_features.add_feature(feature, name="Standless", label=True)
feature = SeqFeature(FeatureLocation(275, 375), strand=-1)
gds_features.add_feature(feature, name="Reverse", label=True)

gdd.draw(format='linear', pagesize=(15*cm, 4*cm), fragments=1,
```

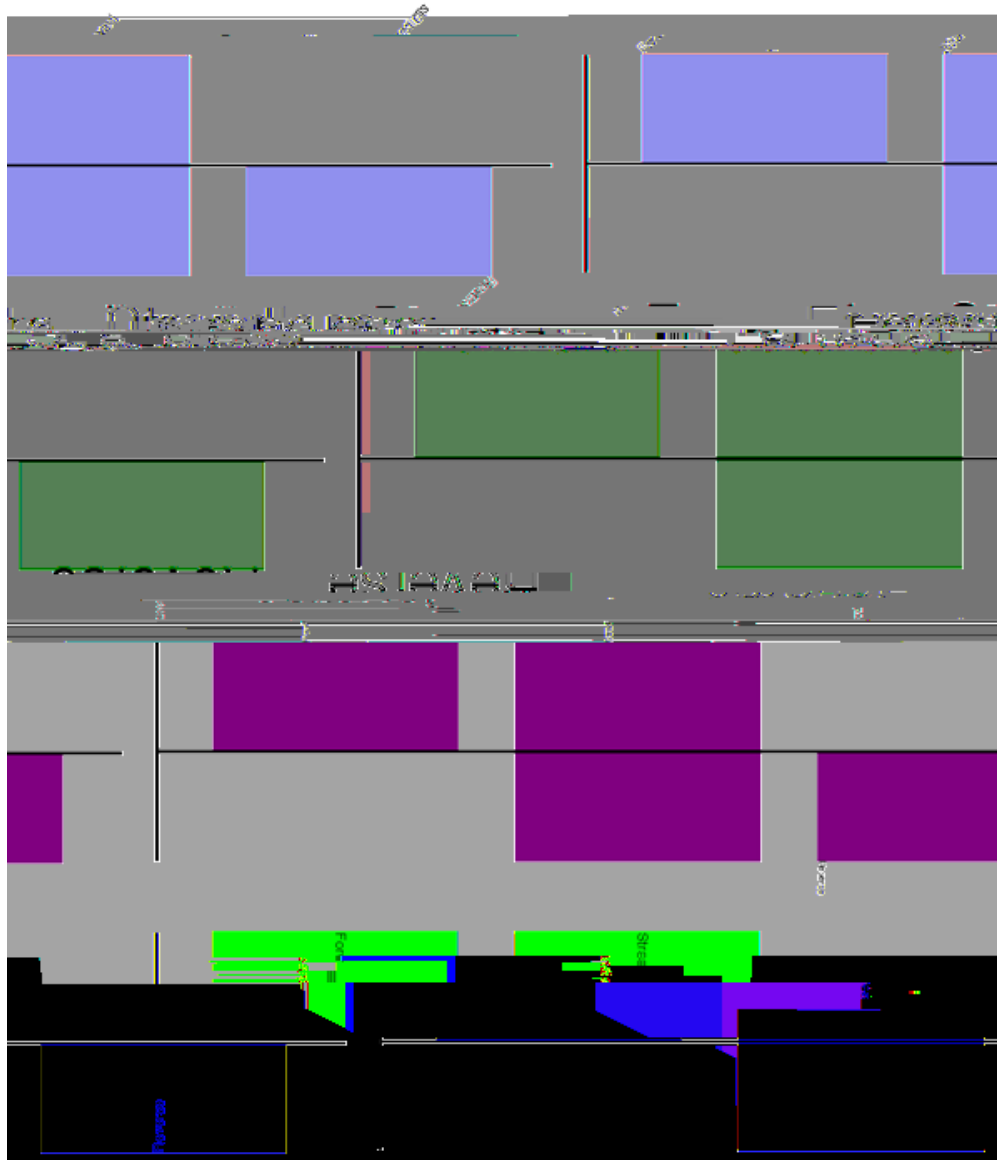


Figure 15.3: Simple GenomeDiagram showing label options. The top plot in pale green shows the default label settings (see Section 15.1.5)

15.1.7 Feature sigils

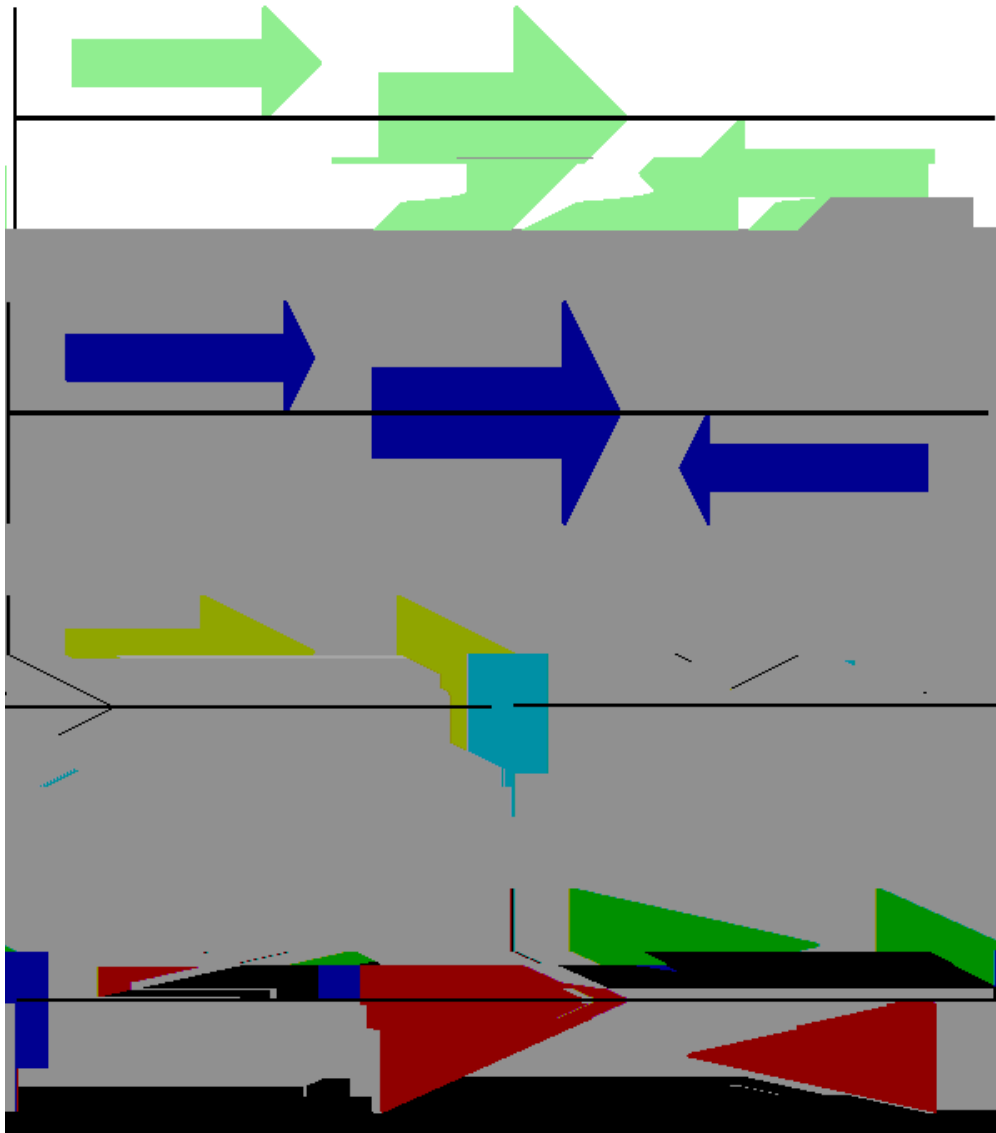


Figure 15.5: Simple GenomeDiagram showing arrow head options (see Section

```
from reportlab.lib import colors
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO
from SeqIO import SeqIO
```

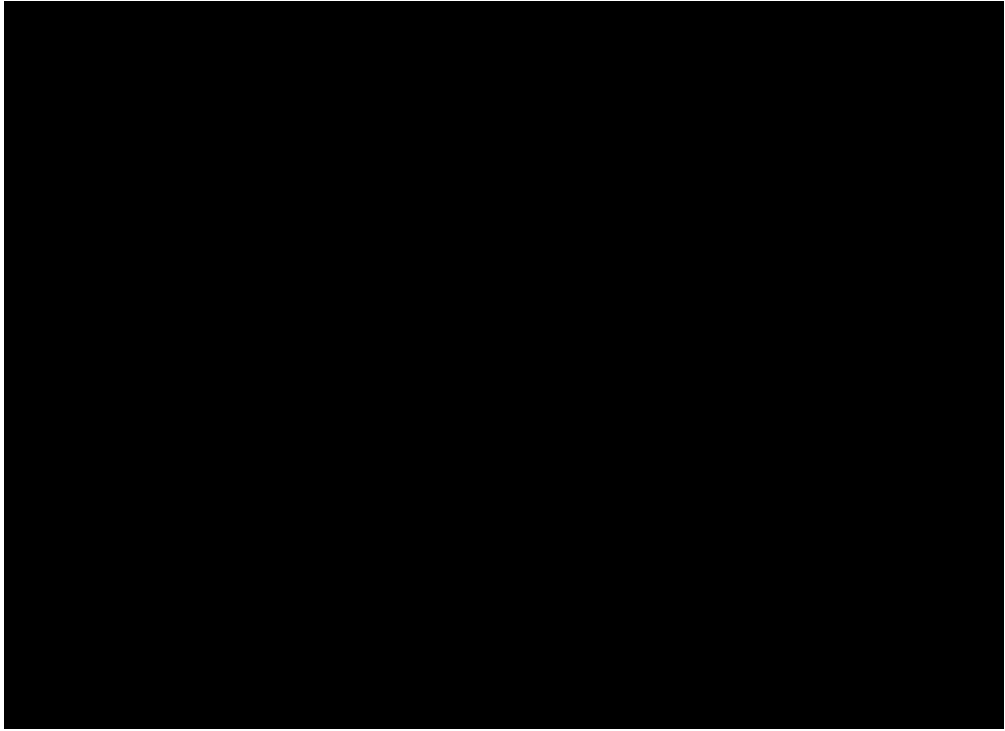


Figure 15.6: Linear diagram for


```

#Add an opening telomere
start = BasicChromosome.TelomereSegment()
start.scale = 0.1 * max_length
cur_chromosome.add(start)

#Add a body - using bp as the scale length here.
body = BasicChromosome.ChromosomeSegment()
body.scale = length
cur_chromosome.add(body)

#Add a closing telomere
end = BasicChromosome.TelomereSegment(inverted=True)
end.scale = 0.1 * max_length
cur_chromosome.add(end)

#This chromosome is done
chr_diagram.add(cur_chromosome)

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

```

This should create a very simple PDF file, shown in Figure 15.7. This example is deliberately short and sweet. One thing you might want to try is showing the location of features of interest - perhaps SNPs or genes. Currently the ChromosomeSegment object doesn't support sub-segments which would be one approach. Instead, you must replace the single large segment with lots of smaller segments, maybe white ones for the boring regions, and codp8ba16460.9713cm0g334a-1(en)2(e)a5a51(e)a5a51(e)orwa5a51interesh.

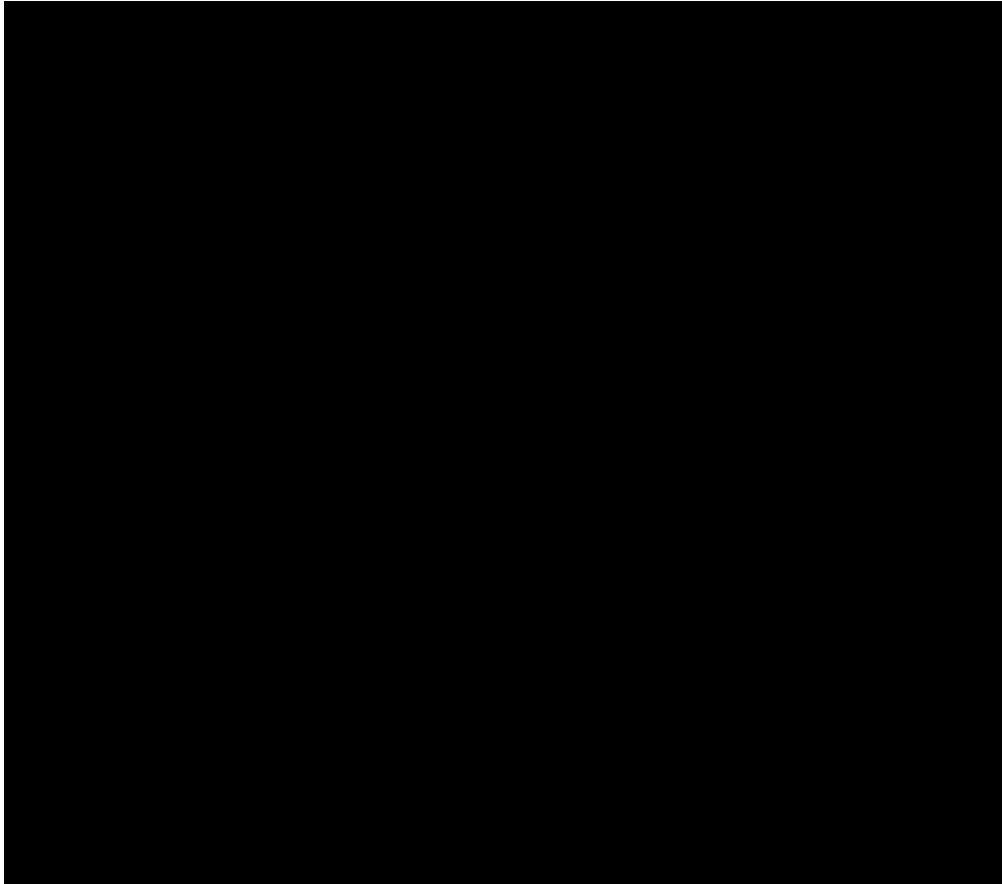


Figure 15.7: Simple chromosome diagram for *Arabidopsis thaliana*.

16.1.2 Producing randomised genomes

Let's suppose you are looking at genome sequence, hunting for some sequence feature – maybe extreme local

Personally I prefer the following version using a function to shuffle 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")
```

16.1.4 Making the sequences in a FASTA file upper case

Often you'll get data from collaborators as FASTA files, and sometimes the sequences can be in a mixture of upper and lower case. In some cases this is deliberate (e.g. lower case for poor quality regions), but usually

First we scan through the file once using `Bi o. SeqIO. parse()`, recording the record identifiers and their

This pulled out only 14580 reads out of the 41892 present. A more sensible thing to do would be to quality trim the reads, but this is intended as an example only.

FASTQ files can contain millions of entries, so it is best to avoid loading them all into memory at once. This example uses a generator expression, which means only one SeqRecord is created at a time - avoiding

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
```

Because we are using a FASTQ input file in this example, the SeqRecord

16.1.10 Converting FASTA and QUAL files into FASTQ files

FASTQ files hold *both* sequences and their quality string. They hold

16.1.13 Identifying open reading frames

```
mi n_pro_l en = 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 52Td[(aa_ew611.9551Td[(i f-(whi l e)-525(aa>=trans_tabl e, )-525(mi n_pr5(
```


If however all you want to find are the locations of the open reading frames, then it is a waste of time

94 orchid sequences

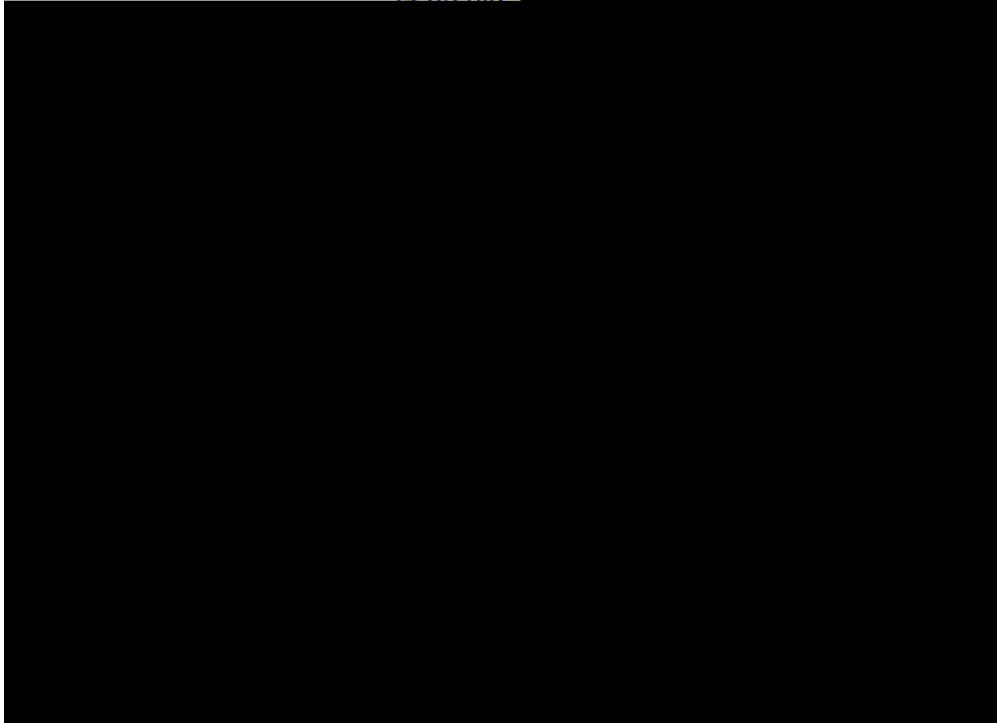


Figure 16.1: Histogram of orchid sequence lengths.

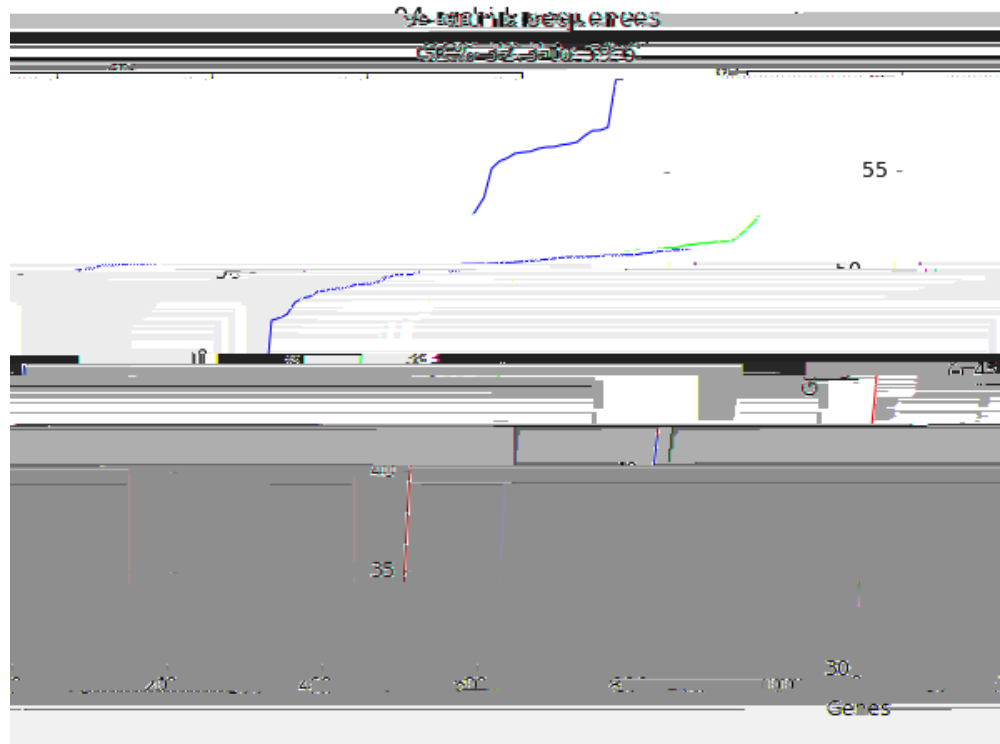


Figure 16.2: Histogram of orchid sequence lengths.

16.2.3 Nucleotide dot plots

A dot plot is a way of visually comparing two nucleotide sequences for similarity to each other. A sliding

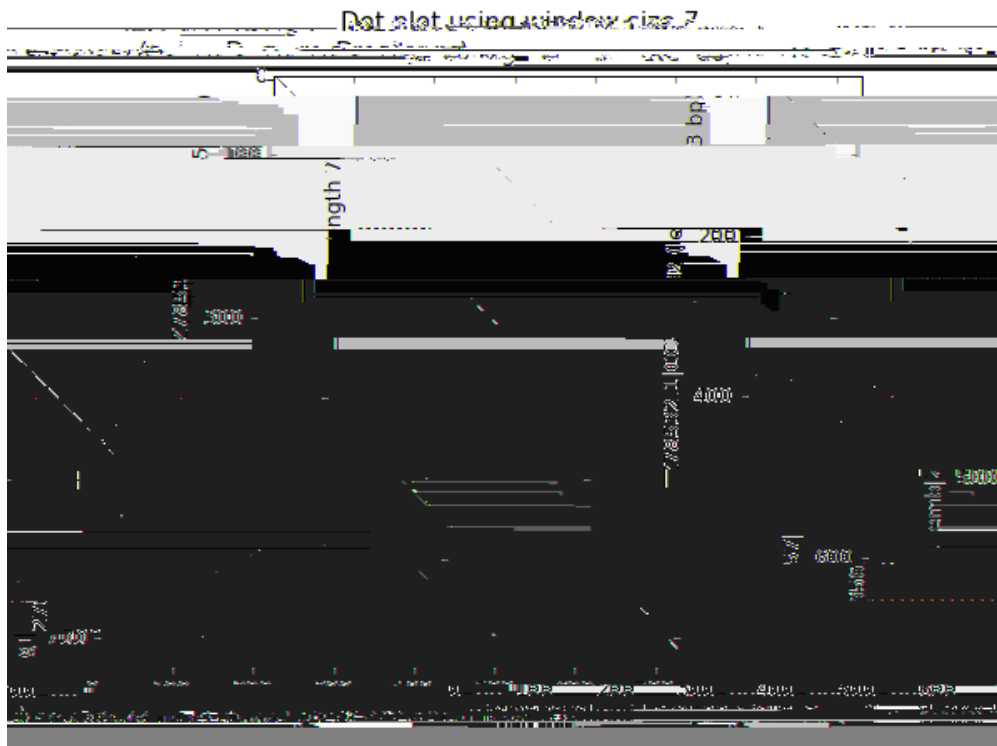


Figure 16.3: Nucleotide dot plot of two orchid sequence lengths (using pylab's imshow function).

Note that we have *not*

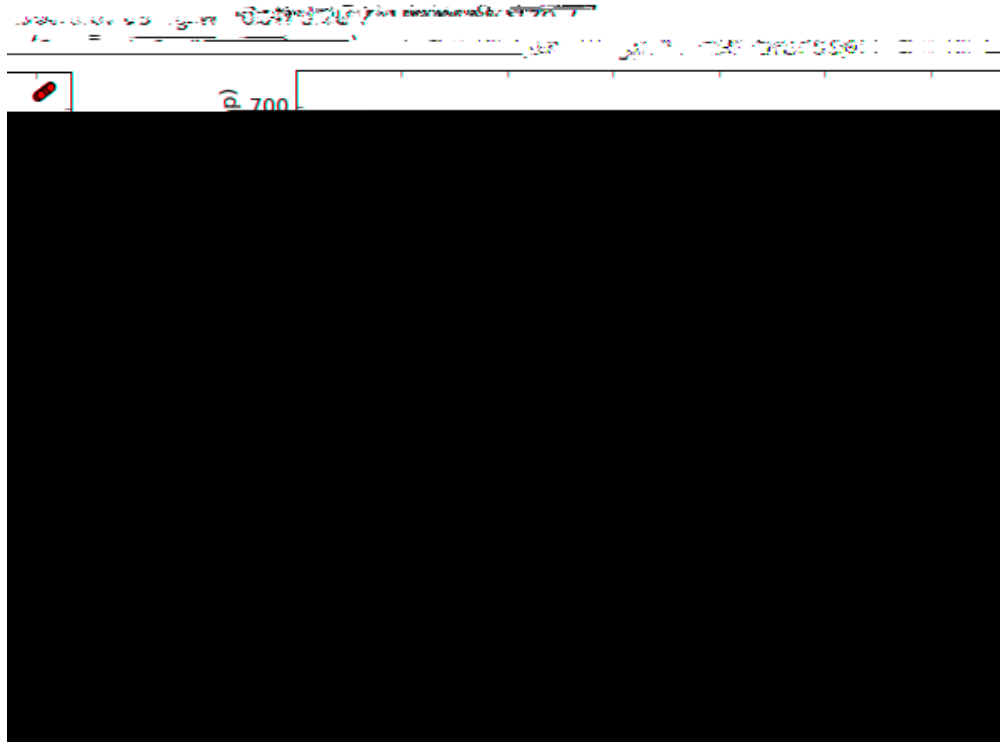


Figure 16.4: Nucleotide dot plot of two orchid sequence lengths (using pylab's scatter function).

```
import pylab
from Bio import SeqIO
for subfigure in [1,2]:
    filename = "SRR001666_%i.fastq" % subfigure
    pylab.subplot(1, 2, subfigure)
    for i,record in enumerate(SeqIO.parse(filename, "fastq")):
        if i >= 50 : break #trick!
        pylab.plot(record.letter_annotations["phred_quality"])
    pylab.ylim(0, 45)
    pylab.ylabel ("PHRED quality score")
    pylab.xlabel ("Position")
pylab.savefig("SRR001666.png")
print "Done"
```

You should note that we are using the Bio. SeqIO format name fastq here because the NCBI has saved

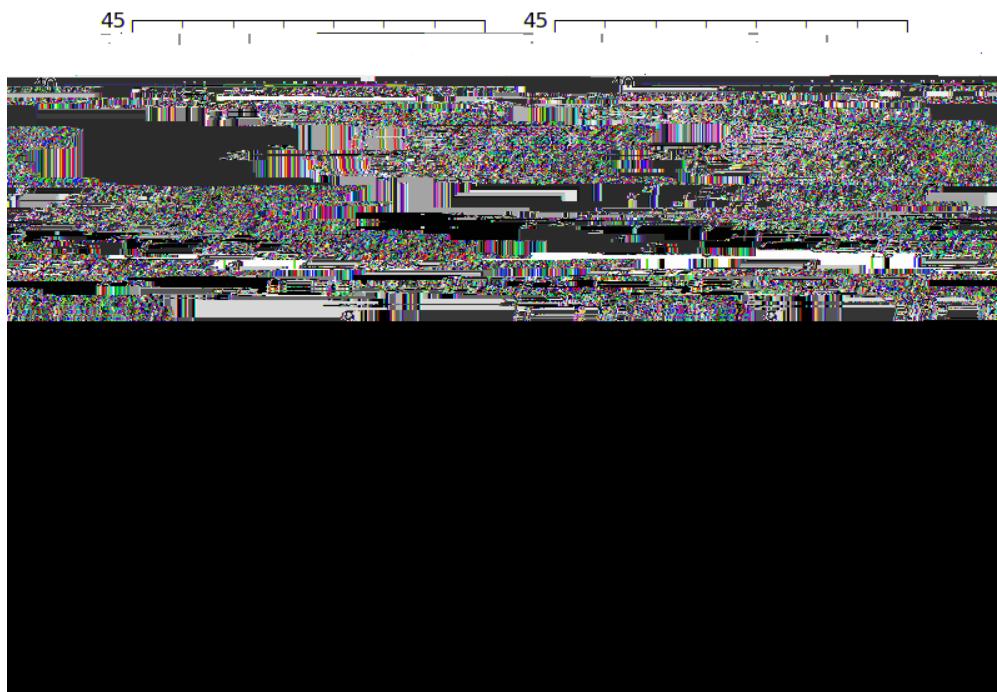


Figure 16.5: Quality plot for some paired end reads.

16.3.1 Calculating summary information

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


- Q_i – The expected frequency of a letter i

16.4 Substitution Matrices

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

Chapter 17

- Simple print-and-compare scripts. These unit tests are essentially short example Python programs, which print out various output text. For a test file named `test_XXX.py` there will be a matching text file called `test_XXX` under the output subdirectory which contains the expected output. All that the test framework does to is run the script, and check the output agrees.
- Standard `unittest`-based tests. These will import `unittest` and then define `unittest.TestCase` classes, each with one or more sub-tests as methods starting with `test_` which check some specific

from Bio imposm


```
"""An addition test"""
result = Biospam.addition(2, 3)
self.assertEqual(result, 5)
```

```
"def addition test"""
result = Biospam.addition9ult, 3)

result = Biospdivi sdi ti on71ult, 3)
```

17.3 Writing doctests

Chapter 18

Advanced

Summing up to 1.

When passing a dictionary as an argument, `yyy1Fdyndi(y)1(icate7(as)-wd[(Wh)1(etd[(Wh)1(ery),)-3ity),asicod[(S)-`

Chapter 20

Appendix: Useful stuff about Python

