

Biopython Tutorial and Cookbook

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

Last Update – 20 May 2010 (Biopython 1.54)

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

13.4 Maximum Entropy	160
13.5 Markov Models	160
14 Graphics including GenomeDiagram	161
14.1 GenomeDiagram	161

17 Advanced	207
17.1 Parser Design	207
17.2 Substitution Matrices	207
17.2.1 SubsMat	207
17.2.2 FreqTable	210

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

Now that you've made it this far, you hopefully have a good understanding of the basics of Biopython and are ready to start using it for doing useful work. The best thing to do now is finish reading this tutorial,

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

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



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

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


T	TTC F	TCC S	TAC Y	TGC C	C
T	TTA L	TCA S	TAA Stop	TGA Stop	A
T	TTG L(s)	TCG S	TAG Stop	TGG W	G
--+-----+-----+-----+-----+--					

```
['ATT', 'ATC', 'ATA', 'ATG', 'GTG']  
>>> mi to_table.forward_table["ACG"]  
'T'
```

3.11 Comparing Seq objects

3.12 MutableSeq objects

Just like the normal Python string, the Seq

3.13 UnknownSeq objects

Biopython 1.50 introduced another basic sequence object, the UnknownSeq object. This is a subclass of the basic Seq object and its purpose is to represent a sequence where we know the length, but not the actual letters making it up. You could of course use a normal Seq object in this situation, but it wastes rather a lot of memory to hold a string of a million "N" characters when you could just store a single letter "N" and the desired length as an integer.

```
>>> from Bio.Seq import UnknownSeq
>>> unk = UnknownSeq(20)
>>> unk
UnknownSeq(20, alphabet = Alphabet(), character = '?')
>>> print unk
????????????????????
>>> len(unk)
20
```

You can of course store a sequence of unknown length in a Seq object. For example, you can create a Seq object of length 1000000 (1 million) using the UnknownSeq class. This is a subclass of the basic Seq object and its purpose is to represent a sequence where we know the length, but not the actual letters making it up. You could of course use a normal Seq object in this situation, but it wastes rather a lot of memory to hold a string of a million "N" characters when you could just store a single letter "N" and the desired length as an integer.

3.14 Working with directly strings

To close this chapter, for those you who *really* don't want to use the sequence objects (or who prefer a functional programming style to an object orientated one), there are module level functions in `Bi o. Seq` will

Chapter 4

Sequence Record objects

Chapter

annotations

location – The location of the SeqFeature


```
>>> from Bio import SeqFeature
>>> start_pos = SeqFeature.AfterPosition(5)
>>> end_pos = SeqFeature.BetweenPosition(8, 1)
>>> my_location = SeqFeature.FeatureLocation(start_pos, end_pos)
```

If you print out a FeatureLocation object, you can get a nice representation of the information:

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

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

```
>>> my_location.start
Bio.SeqFeature.AfterPosition(5)
>>> print my_location.start
>5
>>> my_location.end
Bio.SeqFeature.BetweenPosition(8, 1)
>>> print my_location.end
(8^9)
```

(f-y(ou) don't know it) Wb-n454(v)27(ar)254(9(um(t)-bTJ0aer0sta485[(lf)-454(y)3(jons)-334(27(an)254o)-454(ed7(an)28(t)-454askt

A reference also has a location object so that it can specify a particular location on the sequence that


```
dbxrefs=[ 'Project: 10638' ])  
>>> len(record)  
9609  
>>> len(record.features)  
29
```

For this example we're going to focus in on the *pim* gene, YP_pPCP05

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("ls_orchid.fasta", "fasta"):
    print seq_record.id
    print repr(seq_record.seq)
    print len(seq_record)
```

The above example is repeated from the introduction in Section [2.4](#), and will load the orchid DNA sequences in the FASTA format file [ls_orchid.fasta](#). If instead you wanted to load a GenBank file, you would use `SeqIO.parse("ls_orchid.fasta", "genbank")`. The file `ls_orchid.fasta` is a GenBank file with the following header: `>ls_orchid.fasta`

5.2 Parsing sequences from the net


```
from Bio import SeqIO
orchid_dict = SeqIO.to_dict(SeqIO.parse("ls_orchid.gbk", "genbank"))
```

```

def get_accession(record):
    """Given a SeqRecord, return the accession number as a string.

    e.g. "gi|2765613|emb|Z78488.1|PTZ78488" -> "Z78488.1"
    """
    parts = record.id.split("|")
    assert len(parts) == 5 and parts[0] == "gi" and parts[2] == "emb"
    return parts[3]

```

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

```

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

```

Finally, as desired, the new dictionary keys:

5.4 Sequence files as Dictionaries – Indexed files

As the previous couple of examples tried to illustrate, using `Bio.SeqIO.to_dict()`

Suppose you wanted to know how many records the `Bio.SeqIO.write()` function wrote to the handle? If your records were in a list you could just use `len(my_records)`, however you can't do that when your records come from a generator/iterator. Therefore as of Biopython 1.49, the `Bio.SeqIO.write()` function

```
>>> from Bio import SeqIO
>>> help(SeqIO.convert)
...
```

That would create an in memory list of reverse complement records where the sequence length was under 700 base pairs. However, we can do exactly the same with a generator expression - but with the advantage that this does not create a list of all the records in memory at once:

Chapter 6

Multiple Sequence Alignment objects

This chapter is about Multiple Sequence Alignments, by which we mean a collection of multiple sequences


```
from Bio import AlignIO
alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
print "Alignment length %i" % alignment.get_alignment_length()
```

Note the website should have an option about showing gaps as periods (dots) or dashes, we've shown dashes above. Assuming you download and save this as file "PF05371_

Al pha	AAAAAC
Beta	AAACCC
Gamma	AACAAC
Del ta	CCCCCA
Epsi lon	CCCAAC

...	5	6
Al pha	AAAACC	
Beta	ACCCCC	
Gamma	AAAACC	
Del ta	CCCCAA	
Epsi lon	CAAACC	

>XXX
ACTACCGCTAGCTCAGAAG
>Al pha
ACTACGACTAGCTCAGG
>YYY
ACTACGGCAAGCACAGG
>Al pha
--ACTACGAC--TAGCTCAGG
>ZZZ
GGACTACGACAATAGCTCAGG

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

Leaving the first index as : means take all the rows:

```
>>> print alignment[:, :6]
```

SingleLetterAlphabet() alignment with 7 rows and 6 columns

AEPNAA COATB_BPIKE/30-81

AEPNAA Q9T008_BPIKE/1-52

DGTSTA COATB_BPI22/32-83

AEFSPA COATB_BM13/24-71

AEFSPA COATB_BZJ2E/1491

AEFSPA Q9T09B_BFDE/1491

COATB_BPF1/52-78
sCOATB_BPF1/52-78watosremeasse Notic(e)-46((columns)-45(7,s)-49(8s)-45(aand)-45(9s)-45(whi h)1ca)27hsa


```
>>> edited.sort()
>>> print edited
SingleLetterAlphabet() alignment with 7 rows and 49 columns
DGTSTAATEAMNSLKTQATDLI DQTWPVVTSTVAVAGLAI RLFKKFSSKA COATB_BPI 22/32-83
FAADDAAKAAFDSLTAQATEMSGYAWALVVLVVGATVGI KLFKKFVSRA COATB_BPI F1/22-73
AEPNAAATEAMDSLKTQAI DLI SQTWPVVTTVVAGLVI RLFKKFSSKA COATB_BPI KE/30-81
AEGDDPAKAAFDSLQASATEYI GYAWAMVVVI VGATI GI KLFKKFTSKA COATB_BPM13/24-72
AEGDDPAKAAFDSLQASATEYI GYAWAMVVVI VGATI GI KLFKKFASKA COATB_BPZJ2/1-49
AEPNAAATEAMDSLKTQAI DLI SQTWPVVTTVVAGLVI KLFKKFVSRA Q9TQ08_BPI KE/1-52
AEGDDPAKAAFDSLQASATEYI GYAWAMVVVI VGATI GI KLFKKFTSKA Q9TQ09_BPF1/1-49
```

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


6.4.2 MUSCLE

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



```
>>> from Bio.Emboss.Applications import NeedleCommandline
>>> cline = NeedleCommandline(asequence="alpha.faa", bsequence="beta.faa",
...                             gapopen=10, gapextend=0.5, outfile="needle.txt")
>>> print cline
needle -outfile=needle.txt -asequence=alpha.faa -bsequence=beta.faa -gapopen=10 -gapextend=0.5
```

```
100: from Bio import AlignIO>>> align = AlignIO.read("needle.txt", "emboss")>>> print alignSingleLetterAlphabet() alignment from the 21 ind just like in the MUSCLE file. SPALKTNAKGLVGLVATGEYQ
```


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)
```

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

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

Now that we've got the BLAST results back into a handle again, we are ready to do something with

breaking the Biopython parsers. Our HTML BLAST parser has been removed, but the plain text BLAST parser is still available (see [Section 7.5](#)

```

>>> from Bio.Blast import NCBI XML
>>> blast_records = NCBI XML. parse(result_handle)
>>> blast_record = blast_records.next()
# ... do something with blast_record
>>> blast_record = blast_records.next()
# ... do something with blast_record
>>> blast_record = blast_records.next()
# ... do something with blast_record
>>> blast_record = blast_records.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
# No further records

```

Or, you can use a for-loop:

```

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

```

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

```
...     for hsp in alignment.hsps:
```

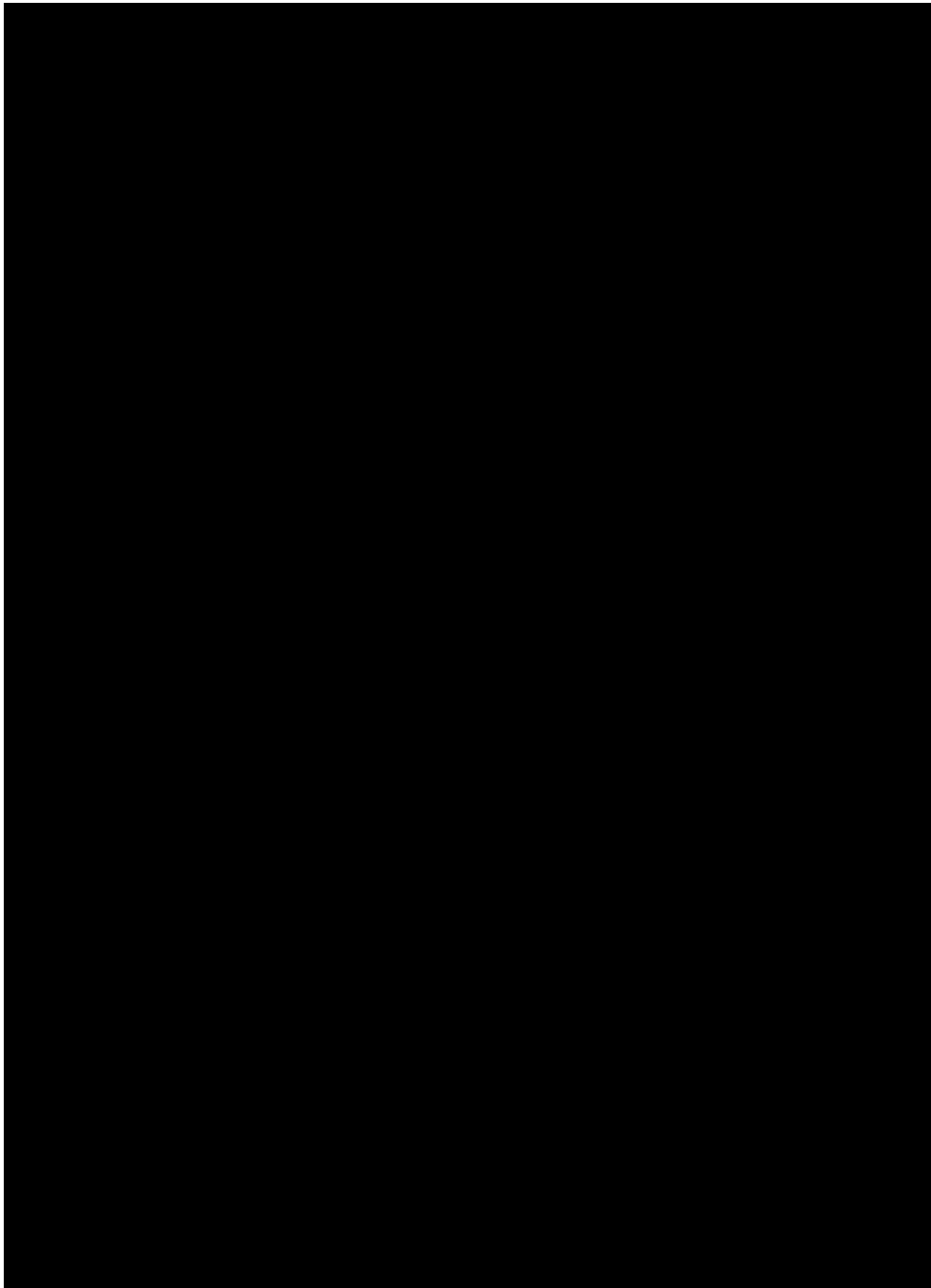



Figure 7.2: Class diagram for the PSIBlast Record class.

Well, now that we've got a handle (which we'll call `result_handle`), we are ready to parse it. This can be done with the following code:

```
>>> from Bio.Blast import NCBIStandalone
>>> blast_parser = NCBIStandalone.BlastParser()
```


Chapter 8

Accessing NCBI's Entrez databases

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

Genome (g) 042 (ts) 042manca1 (lyo) 042eEnheoeioe(o) 042ynan(e) 305(CB) 1ir

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

```
>>> print result
<?xml version="1.0"?>
```

>>>. 0610805FdEvalges 525 (Edign (tim.9626Tifs 2710handle70a) 3Base8378Ed (pown 061082500) a50Me) 28T4TcdBF/0307T


```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"      # Always tell NCBI wh-you25(wh-areEntrez)]TJ0-21.9551Td[

>>> record.email = Eread(handle)Entrez
>>> record["IdList"]Entrez
```

```

>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"      # Always tell NCBI who you are
>>> id_list = ["19304878", "18606172", "16403221", "16377612", "14871861", "14630660"]
>>> print Entrez.epost("pubmed", id=",".join(id_list)).read()
<?xml version="1.0"?>
<!DOCTYPE ePostResult PUBLIC "-//NLM/DTD ePostResult, 11 May 2002//EN"
  "http://www.ncbi.nlm.nih.gov/entrez/query/DTD/ePost_020511.dtd">
<ePostResult>
<QueryKey>1</QueryKey>
<WebEnv>NCID_01_206841095_130.14.22.101_9001_1242061629</WebEnv>
</ePostResult>

```

The returned XML includes two important strings, QueryKey and WebEnv which together define your history

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"    # Always tell NCBI who you are
```

1 attttttacg aacctgtgga aatttttggNtatgacatgtggaaa

11actgtgga(atttcg)-5tgtggagattc atgctgtgga(a5cttcgtttttggNtaatgaa5ct)]TJO. 2304-11. 95520. 037321t

11

```

filename = "gi_186972394.gbk"
if not os.path.isfile(filename):
    print "Downloading..."
    net_handle = Entrez.efetch(db="nucleotide", id="186972394", rettype="gb")
    out_handle = open(filename, "w")
    out_handle.write(net_handle.read())
    out_handle.close()
    net_handle.close()
    print "Saved"

print "Parsing..."
record = SeqIO.read(filename, "genbank")
print record

```

To get the output in XML format, which you can parse using the `Bio.Entrez.read()` function, use `retmode="xml"`:

```

>>> from Bio import Entrez
>>> handle = Entrez.efetch(db="nucleotide", id="186972394", retmode="xml")
>>> record = Entrez.read(handle)
>>> handle.close()
>>> record[0]["GBSeq_definition"]
'Selenipedium aequinoctiale maturase K (matK) gene, partial cds; chloroplast'
>>> record[0]["GBSeq_source"]
'chloroplast Selenipedium aequinoctiale'

```

So, that dealt with `seqe0187178T(s)-1.mFTrh ee pausingtst sot thegh`
`$ial,45cm018data34(S178T(s)e)]TJ/10.7845cm0g2n`


```
>>> for row in record["eGQueryResult"]: print row["DbName"], row["Count"]
...
pubmed 6
pmc 62
journals 0
...
```

See the [EGQuery help page](#) for more information.

8.9 ESpell: Obtaining spelling suggestions

ESpell retrieves spelling suggestions. In this example, we use `Bio.Entrez.espell()` to obtain the correct spelling of Biopython:

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


8.13 Examples

8.13.1 PubMed and Medline

If you are in the medical field or interested in human issues (and many times eH-i=-318(or)-34Td[(If)-318(y)28(ou)eyonot!

We can get the lineage directly from this record:


```

>>> 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))
>>> pmc_ids = [link["Id"] for link in results[0]["LinkSetDb"][0]["Link"]]
>>> pmc_ids
['2744707', '2705363', '2682512', ..., '1190160']

```

Great - eleven articles. But why hasn't the Biopython application note been found (PubMed ID 19304878)? Well, as you might have guessed from the variable names, there are not actually PubMed

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.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 2 2 atoms from

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

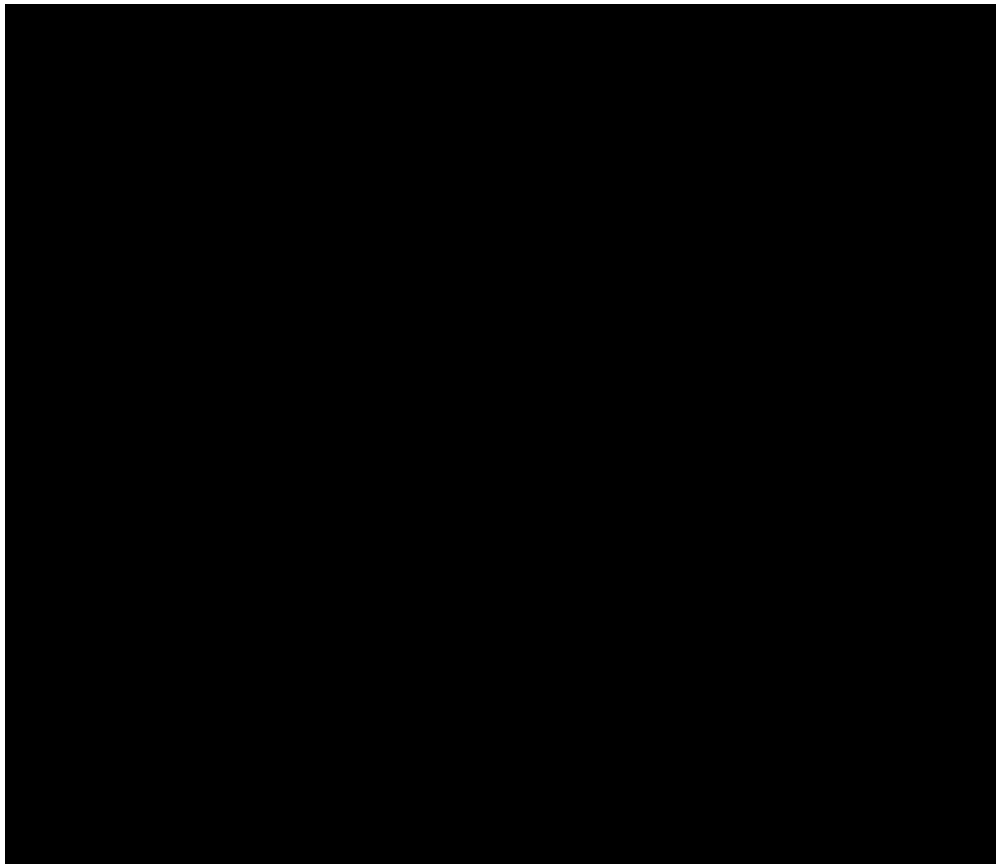


Figure 10.1: UML diagram of the SMCRA data structure used to represent a macromolecular structure.


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

This corresponds to:

- The Structure with id "1abc"

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

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

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

10.2 Disorder

10.2.1 General approach

D0Geer should be dealt with from two points of view: the atom and the residue points of view. In general, we have tried to encapsulate all the complexity that arises from disorder. If you just want to loop over all C atoms, you do not care that some residues have a D0Geered side chain. On the other hand it should also be possible to represent disorder completely in the data structure. Therefore, disordered atoms or residues are stored in special objects that behave as if there is no disorder. This is done by only representing a subset of the disordered atoms or residues. Which subset is picked (e.g. which of the two disordered OG side chain atom positions of a Ser residue is used) can be specified by the user.

10.2.2 Disordered atoms

D0Geered atoms are represented by ordinary Atom objects, but all Atom objects that represent the same physical atom are stored in a D0GeeredAtom object. Each Atom object in a D0Getom object can be uniquely indexed using its altloc specifier. The D0Getom object forwards all uncaught method calls to the selected Atom object, by default the one that represents the atom with the highest occupancy. The user can of course change the selected Atom object, making use of its altloc specifier. In this way atom disorder is represented correctly without much additional complexity. In other words, if you are not interested in atom disorder, you will not be bothered by it.

Each D0Geered atom has a characteristic altloc identifier. You can specify that a DisorderedAtom object should behave like the Atom object associated with a specific altloc identifier:

```
atom.d0Ge # select altloc A atom
```

```
print atom.get_altloc()  
"A"
```

```
atom.d0Ge # select altloc B atom  
print atom.get_altloc()  
"B"
```

10.2.3 Disordered residues

10.2.3.1 Common case

The most common case is a residue that contains one or more disordered atoms. This is eventually solved by using D0Getom objects to represent the disordered atoms, and storing the D0GeeredAtom object in

a Residue object just like ordinary Atom objects. The D0GeeredAtom will behave exactly like an ordinary

atom (in fact the atom with the highest occupancy) by forwarding all uncaught method calls to one of the Atom objects (the selected Atom object). This is done by a polymorphic method, i.e. when two or more polymorphic methods are present in the class hierarchy, the method to be called is determined by the type of the object.

be stored in a Residue object in the common case. The disordered residue is represented by the

Residue object, but the atoms are stored in DisorderedAtom objects. The DisorderedAtom objects are stored in the DisorderedAtom object.

The DisorderedAtom object is a subclass of the Atom object. The DisorderedAtom object is a subclass of the Atom object.

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

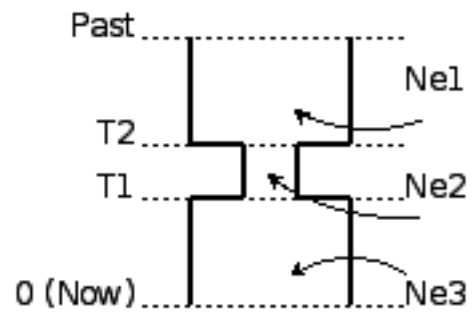


Figure 11.1: A bottleneck

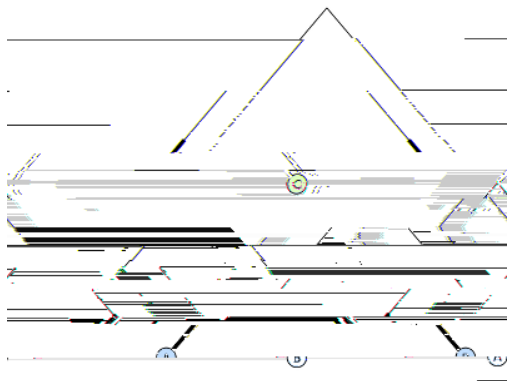
11.2.1.2 Chromosome structure

We strongly recommend reading SIMCOAL2 documentation to understand the full potential available in

12.2 Viewing and exporting trees

The simplest way to get an overview of a Tree object is to print it:

```
>>> tree = Phyl o.read("example.xml", "phyl oxml")
>>> print tree
Phylogeny(rooted='True', description='phyl oXML allows to use either a "branch_length"
attribute...', name='example from Prof. Joe Felsenstein's book "Inferring Phyl...')
  Clade()
    Clade(branch_length='0.06')
```



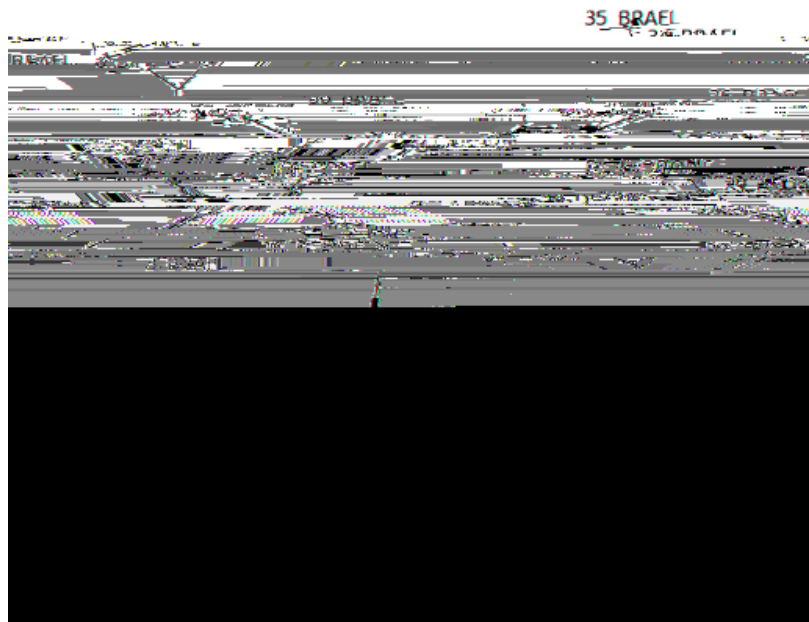


Figure 12.2: A larger tree, using neato for layout.

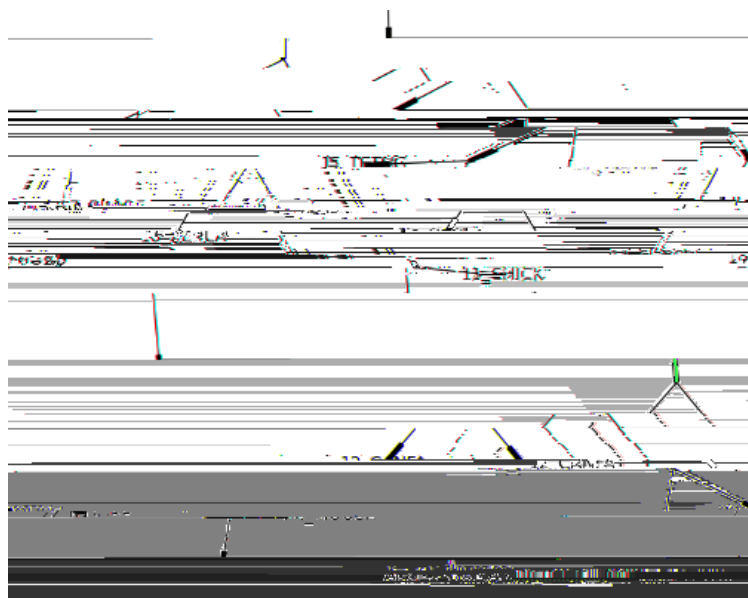


Figure 12.3: A zoomed-in portion of the same tree.

- None matches None
- If a string is given, the value is treated as a regular expression (which must match the whole string in the corresponding element attribute, not just a prefix). A given string without special regex characters will match string attributes exactly, so if you don't use regexes, don't worry about it.

12.3.2 Information methods

These methods provide information about the whole tree (or any clade).

`common_ancestor` Find the most recent common ancestor of all the given targets. (This will be a Clade

prune Prunes a terminal clade from the tree. If taxon is from a bifurcation, the connecting node will

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

```

Iteration: 2 Log-likelihood function: -5.76877209868

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

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 which the model is recalculated from the training data after removing the gene to be predicted:

```
>>> for i in range(len(ys)):
    model = LogisticRegression.train(xs[:i]+xs[i+1:], ys[:i]+ys[i+1:])
    print "True:", ys[i], "Predicted:", LogisticRegression.classify(model, xs[i])
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: 1
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 1
True: 0 Predicted: 0
True: 0 Predicted: 0
```

The leave-one-out analysis shows that the prediction of the logistic regression model is incorrect for only two of the gene pairs, which corresponds to a prediction accuracy of 88%.

In Biopython, the k -nearest neighbors method is available in `Bi o. kNN`. To illustrate the use of the k -nearest neighbor method in Biopython, we will use the same operon data set as in section 13.1.

13.2.2 Initializing a k -nearest neighbors model

Using the data in Table 13.1, we initialize a k -nearest neighbor model as follows:

```

...
>>> 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 will calculate the total weight assigned to the classes OP and NOP. For the default weighting scheme, this reduces to the number of neighbors in each category. For *yxcE*, *yxcD*, we find

```

>>> x = [6, -173.143442352]
[6, -173.14yxcE, -11.90P: 2352]

```


Chapter 14

Graphics including GenomeDiagram

The Bio. Graphics

14.1.3 A top down example

14.1.4 A bottom up example

```
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,
```

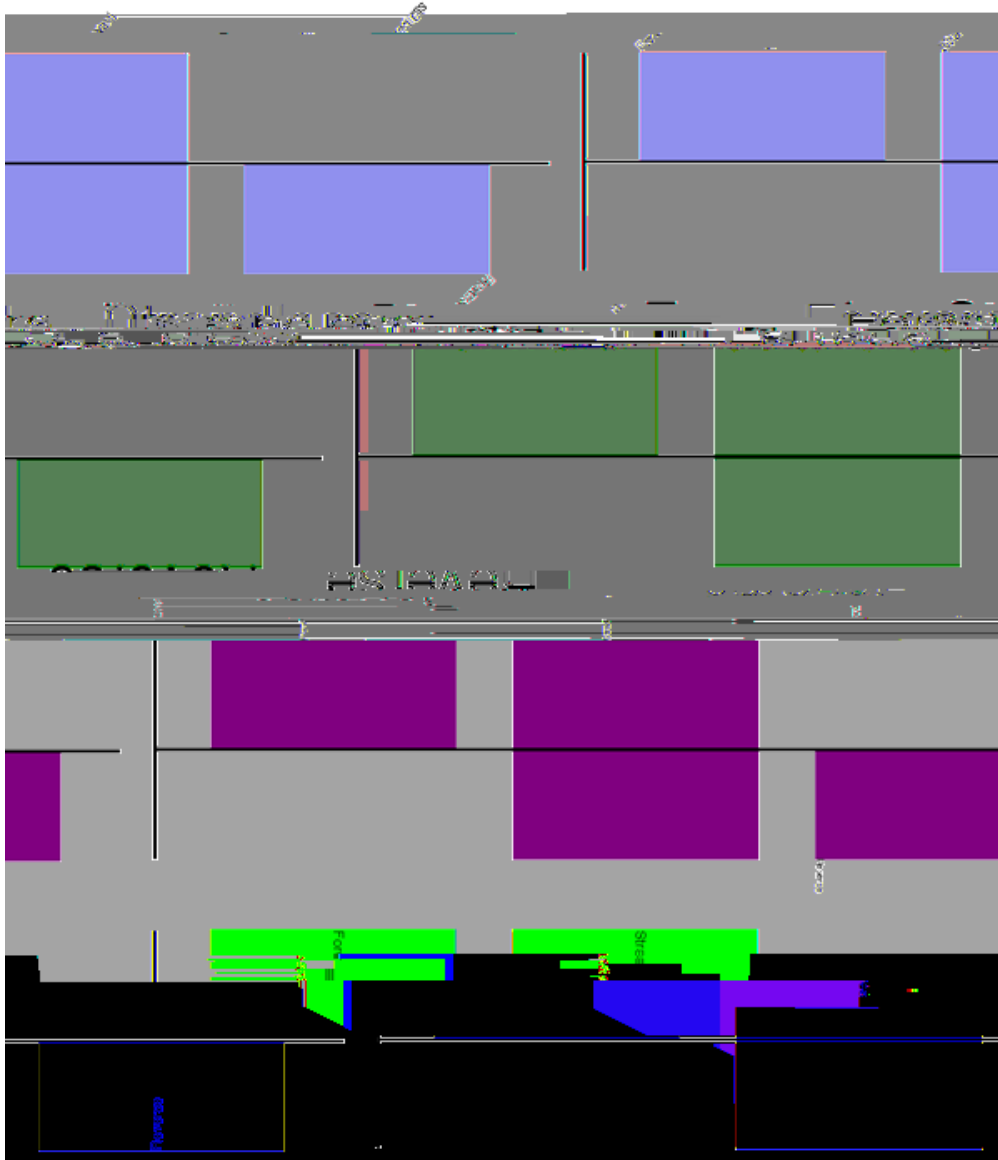


Figure 14.3: Simple GenomeDiagram showing label options. The top plot in pale green shows the default label settings (see Section 14.1.5)

14.1.7 Feature sigils

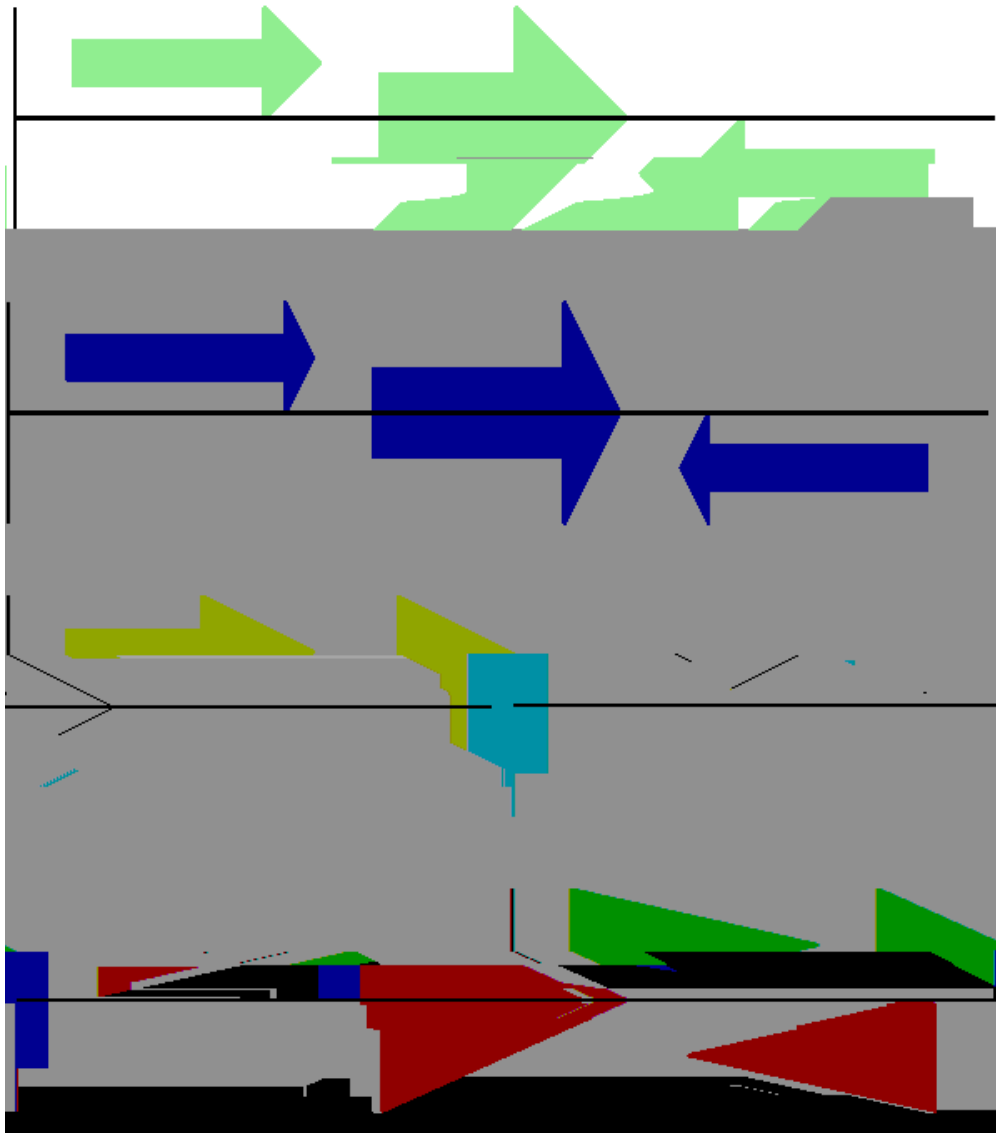


Figure 14.5: Simple GenomeDiagram showing arrow head options (see Section [14.1.7](#))

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

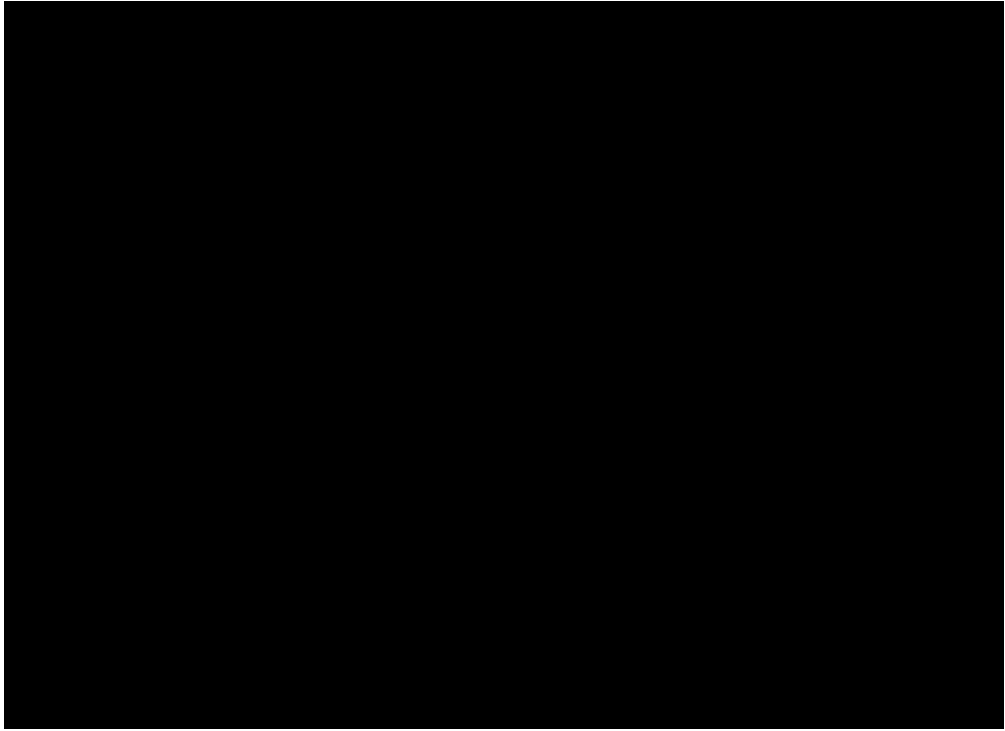


Figure 4 Midragus
sites (see Section 9)


```

#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 14.7. 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 colored ones for the regions of interest.

Chapter 15

Cookbook – Cool things to do with it

15.1.2 Translating a FASTA file of CDS entries

Suppose you've got an input file of CDS entries for some organism, and you want to generate a new FASTA file containing their protein sequences. i.e. Take each nucleotide sequence from the original file, and translate

it. Back in Section 3.9 we saw how to use the Seq object's `translate()` method to translate a single sequence.

```

from Bio import SeqIO
#Get the lengths and ids, and sort on length
len_and_ids = sorted((len(rec), rec.id) for rec in \
                      SeqIO.parse("ls_orchid.fasta", "fasta"))
ids = reversed([id for (length, id) in len_and_ids])
del len_and_ids #free this memory
record_index = SeqIO.index("ls_orchid.fasta", "fasta")
records = (record_index[id] for id in ids)
SeqIO.write(records, "sorted.fasta", "fasta")

```

First we scan through the file once using

from Bio import SeqIO

Because we are using a FASTQ input file in this example, the SeqRecord objects have per-letter-

15.1.8 Converting FASTA and QUAL files into FASTQ files

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


```
fq_dict.keys()[:4]
['SRR014849.80961', 'SRR014849.80960', 'SRR014849.80963', 'SRR014849.288943']
>>> fq_dict["SRR014849.171880"].seq
Seq('AAAATCATCTCTGGCGCCCCGTTGGAACCGAAAGGGTTGAATTCAAACCCTTT...CAG', SingleLetterAlphabet())
```

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

The example in Section [15.1.3](#) show how you can use the `Bio.SeqIO.indexIra0-o.Seq7AACCC.9626Tffuen u(sho)-38`

15.1.11 Identifying open reading frames

94 orchid sequences

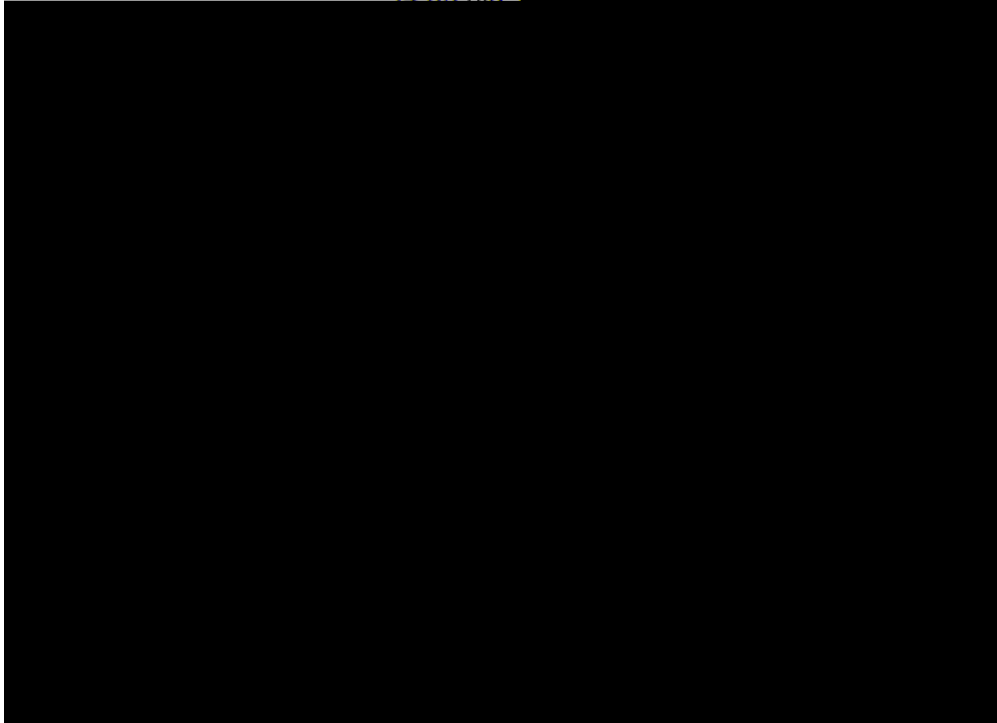


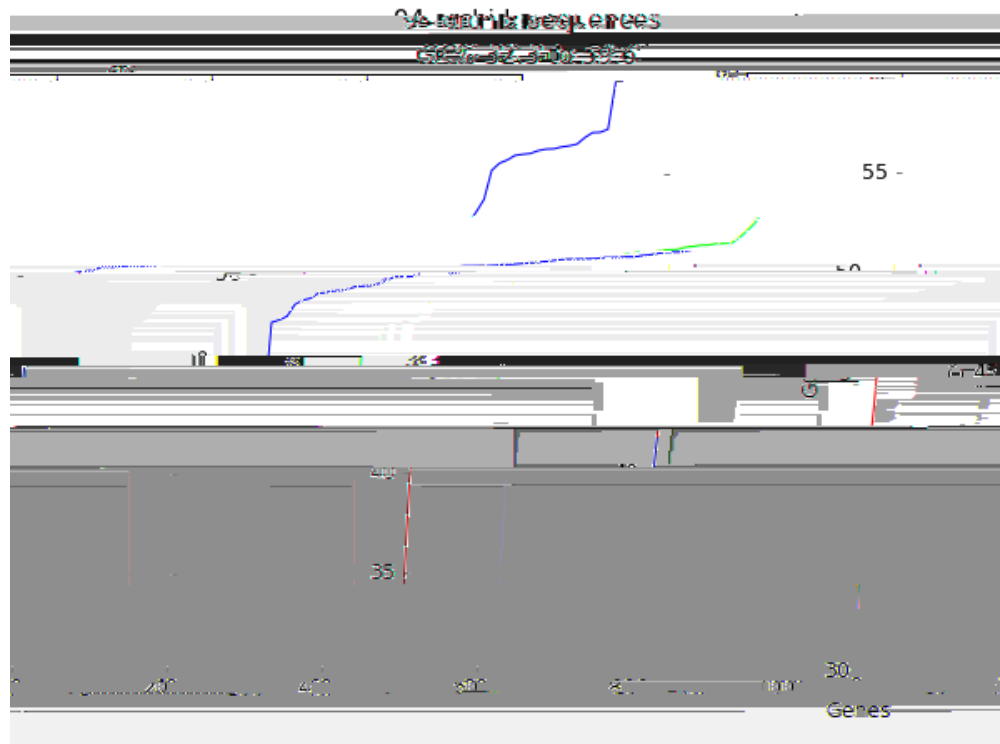
Figure 15.1: Histogram of orchid sequence lengths.

Tip: Rather than using

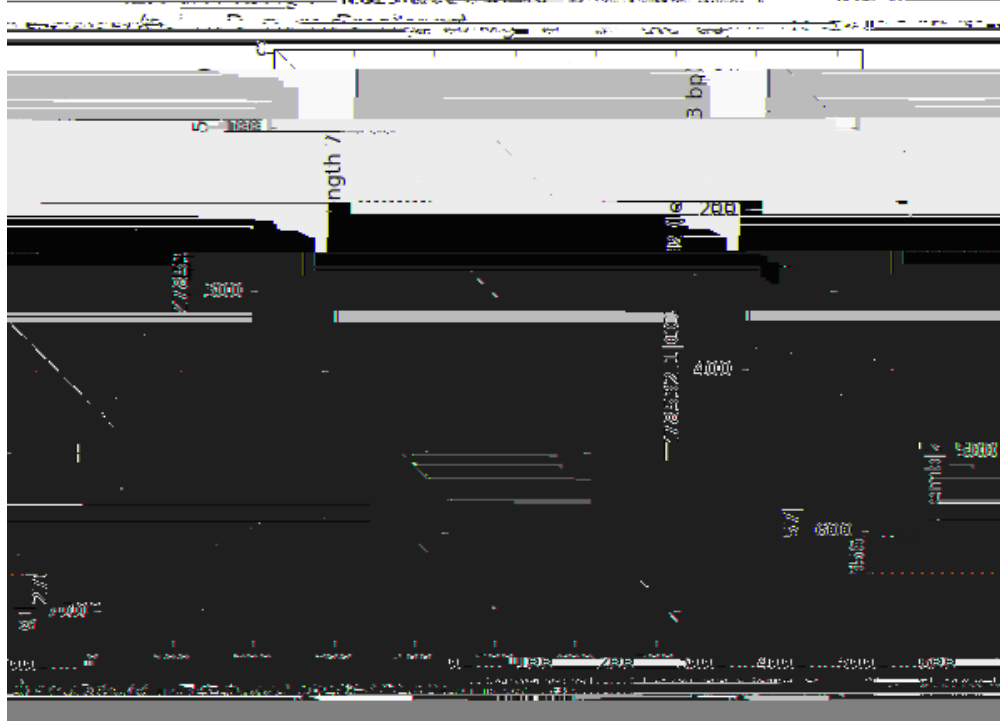
```

pylab.title("%i orchid sequences\nGC%% %0.1f to %0.1f" \
            % (len(gc_values), min(gc_values), max(gc_values)))
pylab.xlabel("Genes")
pylab.ylabel("GC%")
pylab.show()

```

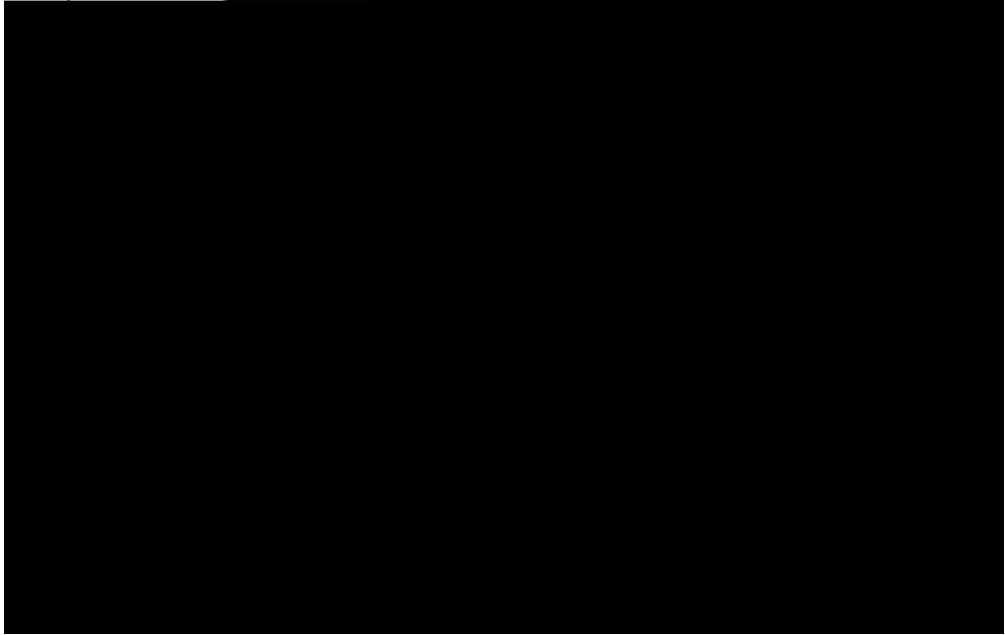


Dot plot using window size 7





6700



from the read lengths, this data was from an Illumina Genome Analyzer and was probably originally in one of the two Solexa/Illumina FASTQ variant file formats instead.

This example uses the `pylab.savefig(...)` function instead of `pylab.show(...)`

2. Get a position specific score matrix for the alignment – see section [15.3.3](#)
3. Calculate the information content for the alignment – see section

15.3.4 Information Content

A potentially useful measure of evolutionary conservation is the information content of a sequence.

A useful introduction to information theory targetted towards molecular biologists can be found at <http://www.lecb.nci.fcrf.gov/~toms/paper/primer/>

15.5 BioSQL – storing sequences in a relational database

[BioSQL](#) is a joint effort between the [OBF](#) projects (BioPerl, BioJava etc) to support a shared database schema for storing sequence data. In theory, you could load a GenBank file into the database with BioPerl, then using Biopython extract this from the database as a record object with features - and get more or less

Chapter 16

- 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.

-


```
print "2 + 3 =", Bi ospam.addi ti on(2, 3)
print "9 - 1 =", Bi ospam.addi ti on(9, -1)
print "2 * 3 =", Bi ospam.mul ti pli ca ti on(2, 3)
print "9 * (- 1) =", Bi ospam.mul ti pli ca ti on(9, -1)
```

We generate the corresponding output with `python run_tests.py -g test_Bi ospam.py`, and check the output (test_Biospay)]TJ/F89.9626Tf99.376590Td[:y)]TETG1001-67.0187596.39127cm0g0G0g0G1001-67.0187596.39127cmBT

```
- 1 1
* 3 3
* (- 1) 9 91
```

with(h)eth(n)3n12prnt-andh(e)28khe)312(h)n(e)p
hps(e)n(ou)072match28(h)e)372li(n)1(e)-1tetgeatped
(ou)072match28(h)e)372li(n)1(e)-1tetgeatped
We n28(ain28(tn)290(aln)1ln)290((th)1(e)-912mop)-87dri(n)290(Bip)28(ye)1tthepo(n)1(e)-1ttseaidrs(imlee)290((h)1n


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

    def test_addition2(self):
        """A second addition test"""
```


Chapter 17

Advanced

17.1 Parser Design


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


Chapter 19

Appendix: Useful stuff about Python

