



# Contents

## 1 Introduction

4.3.1	SeqFeature objects	38
4.3.2	Positions and locations	39
4.3.3	Sequence described by a feature or location	42
4.4	References	43
4.5	The format method	43
4.6	Slicing a SeqRecord	43
4.7	Adding SeqRecord objects	46
4.8	Reverse-complementing SeqRecord objects	48
<b>5</b>	<b>Sequence Input/Output</b>	<b>49</b>
5.1	Parsing or Reading Sequences	49



<b>10 Swiss-Prot and ExPASy</b>	<b>145</b>
10.1 Parsing Swiss-Prot files	145
10.1.1 Parsing Swiss-Prot records	145
10.1.2 Parsing the Swiss-Prot keyword and category list	147
10.2 Parsing Prosite records	148
10.3 Parsing Prosite documentation records	149
10.4 Parsing Enzyme records	149
10.5 Accessing the ExPASy server	151
10.5.1 Retrieving a Swiss-Prot record	151
10.5.2 Searching Swiss-Prot	152
10.5.3 Retrieving Prosite and Prosite documentation records	152
10.6 Scanning the Prosite database	153
<b>11 Bio.PopGen: Population genetics</b>	<b>155</b>
11.1 GenePop	155
11.2 Coalescent simulation	157
11.2.1 Creating scenarios	157
11.2.2 Running Fastsimcoal2	159
11.3 Other applications	160
11.3.1 FDist: Detecting selection and molecular adaptation	160
11.4 Future Developments	163

13.8 Comparing motifs . . . . .	202
13.9 <i>De novo</i> motif finding . . . . .	203
13.9.1 MEME . . . . .	203
13.9.2 AlignAce . . . . .	204
13.10 Useful links . . . . .	205
<b>14 Cluster analysis</b> . . . . .	<b>206</b>
14.1 Distance functions . . . . .	207
14.2 Calculating cluster properties . . . . .	210
14.3 Partitioning algorithms . . . . .	212
14.4 Hierarchical clustering . . . . .	215
14.5 Self-Organizing Maps . . . . .	219
14.6 Principal Component Analysis . . . . .	221
14.7 Handling Cluster/TreeView-type files . . . . .	222
14.8 Example calculation . . . . .	227
14.9 Auxiliary functions . . . . .	227
<b>15 Supervised learning methods</b> . . . . .	<b>228</b>
15.1 The Logistic Regression Model . . . . .	228
15.1.1 Background and Purpose . . . . .	228
15.1.2 Training the logistic regression model . . . . .	229a[0(. [(206)]TJ 1
14.6 Principal Component Analysis . . . . .	203 .
14.9 Auxiliary functions . . . . .	227
14.4 Hierarchical clustering . . . . .	58(k9.9626 Tf 39.987 05.(206)-500(.)-500(.rg 1 0 0 RG /F8 941)-500(.)-500(.)-500(. .
14.9 Auxiliary functions . . . . .	rg 1 0 0 RG /F8 943 . . . . .







# 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 scientific 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 specific

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](#) file.

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

## Chapter 2

### Quick Start { What can you do with Biopython?

followed by what you would type in:

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq('AGTACACTGGT')
>>> my_seq
Seq('AGTACACTGGT')
>>> my_seq

>>> my_seq.alphabet
```

edited histidine DNA protein database 83(, >) 3964 at protein database 1010

jean tierson (from) 468 (th(t) 469 (Python) 469 (string) 468 (ind) 469 (th(t) 468 (method) -28 dtd) 469 (itq)] TJ-216. 1010

```
>>> my_seq
Seq('AGTACACTGGT')
>>> my_seq.complement
Seq('TCATGTGACCA')
>>> my_seq.reverse_complement
```



## 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', Al phabet())
>>> my_seq.al phabet
Al phabet()
```

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.al phabet
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.al phabet
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_seq):
>>> stabet[(atc(my_seq[index]))]TJO-11.955Td5051

>>>stabet(my_[0]051>> l ett51
>>>stabet(my_[2]051>>rdex, l ett51
>>>stabet(my_[-1]051>> l ett51
```

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 _seqsort

```

```

>>> from Bio.Alphabet import
>>> protein_seq.alphabet
>>> dna_seq.alphabet
>>> 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' ", IUPAUunambiguousDNAaa)
>>> dna_seq + dna_seq
>>>>> dna_seq.alphabet, 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 different 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 the 0-11.956 Txr3(0-1(allai-333(Py4(as)-956 Tx6(So)--333(on)-3334(as)-3do?)rote44525(g)-334(tld)-334525(on)-3ab)-2

```
>>> 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 import Seq
>>> 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(l i ne, q-369(whi l e)-362(theq)]T/FF349.9626Tf174.41160Td[i dq)]T/F849.9626Tf14.062  
>>> recorddescription
```

```
>>> recordletter_annotations
```

```
>>> recorddbxrefsq
```

```
i 4547i 454gets-333(aburesqq)]TJ00he. g. 141mpl ete>
```







```
>>> my_location.sTt
```

```
>>>my_location.sTt)
```

```
>>> my_locationend
```

```
>>>my_locationend)
```

```

>>> for feature in record.features:
...     if my_snp in feature:
...         print("%s %s" % (feature.type, feature.qualifiers.get('db_xref')))
...
source ['taxon: 229193']
gene ['GeneID: 2767712']
CDS ['GI: 45478716', 'GeneID: 2767712']

```

Note that gene and CDS features from GenBank or EMBL files defined with joins are the union of the exons { they do not cover any introns.

### 4.3.3 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: [l:45478716'ef, [':rID:2767712']51
  Key: geef, Value: pim']51

Key: Value: NP_91.971.1']51
Key: Value: 11']51
Key: Value: MGGGMI SKLFCLALI FLSSSGLAEKNTYTAKDI LQNLELNTFGNSLSH...']51
```



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



Also note that in an example like this, you should probably change the record identifiers since the NCBI references refer to the *original* unmodified 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 sequences 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.gb", "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

Description: 18S rDNA (1.8 kb) from *Isotria medeolae* (L.) Nutt. (Mammillariaceae). The sequence was determined by the Sanger method. The sequence was deposited in the GenBank database on 10/10/2000.

```
print(first_record.annotations)
```

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

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

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

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

```
from Bio import SeqIO
```

## 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). This is basically a helper function to build a normal Python dictionary

#### 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("Is_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 JUeWn6DPhgZ9nAyowsgtoD9TTo

Z78532.1 MN/s0q9zDoCvEEc+k/I FwCNF2pY

...

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

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

### 5.4.3 Sequence files as Dictionaries { Database indexed files

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

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

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

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index("Is_orchid.gb", "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 file format:

[illegible]

```
>>> 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))
```





Epsi l on	CCCAAC
...	
5	6
Al pha	AAAACC
Beta	ACCCCC
Gamma	AAAACC
Del ta	CCCCAA
Epsi l on	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
>Al pha
--ACTACGAC--TAGCTCAGG
>ZZZ
GGACTACGACAATAGCTCAGG
```

## 6.2 Writing Alignments

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

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

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



KA  
KA  
KA  
KA  
RA

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

```
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()



DGTSTATSYATEAMNSLKTOATDLIDQTWPVVTSVAVAGLAIRL...SKA COATB\_BPI 22/32-83

```
>>> 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_cli = NeedleCommand(r"C:\EMBOSS\525(ne.exe", line)
TJO-11.955T...
```

## Chapter 7

# BLAST

Hey, everybody loves BLAST right? I mean, geez, how can get it get any easier to do comparisons between

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 the format expected by the parser, described in section [7.3](#) below.

The argument `expect` sets the expectation or e-value threshold.

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

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



You can do the BLAST search yourself on the NCBI site through your web browser, and then save

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...

||||||| | ||||||||| || ||| || || |||-11.955Td31.955T(||)-525(||)-525(|||-25(|||acatattcctc...)

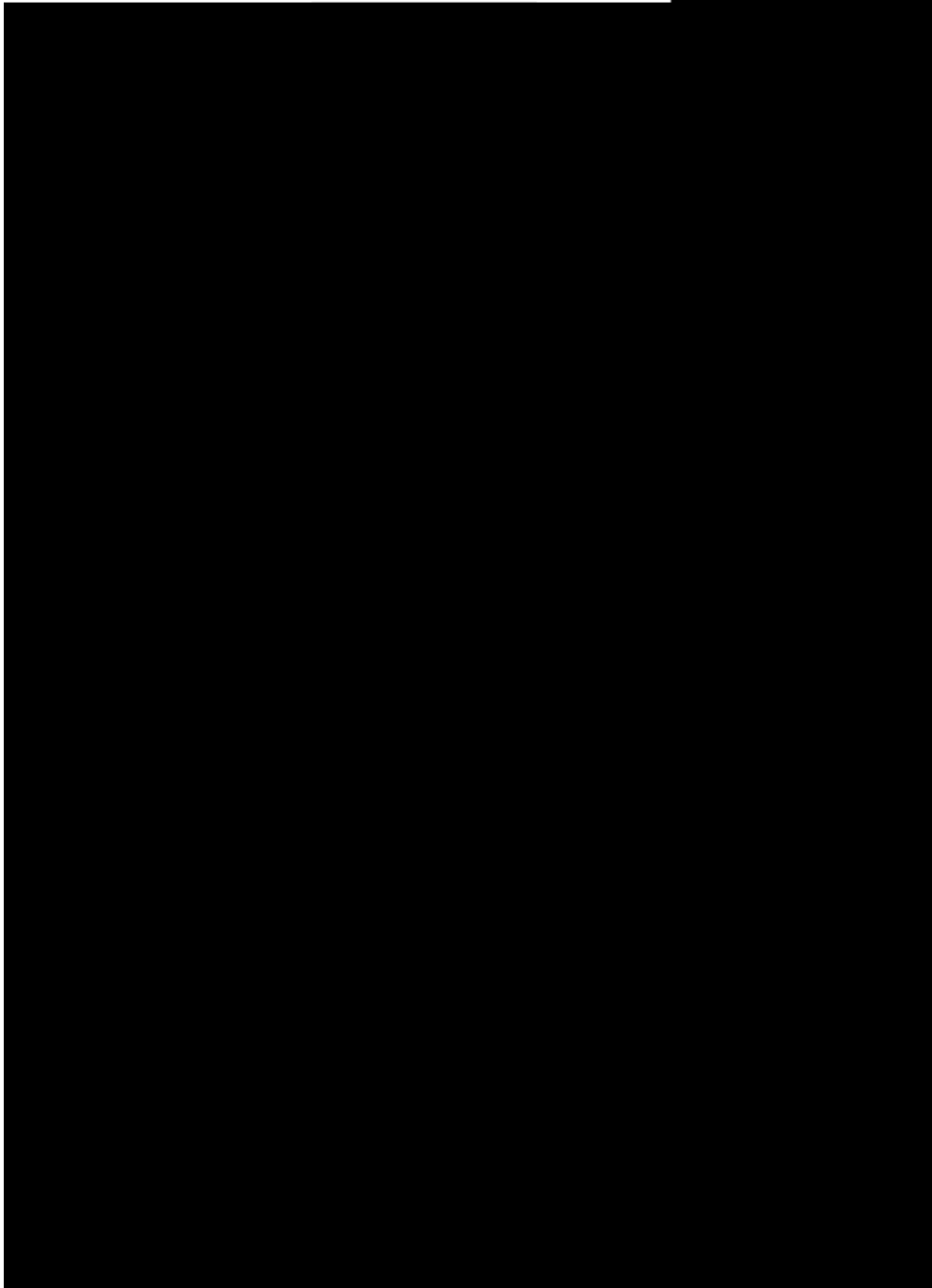
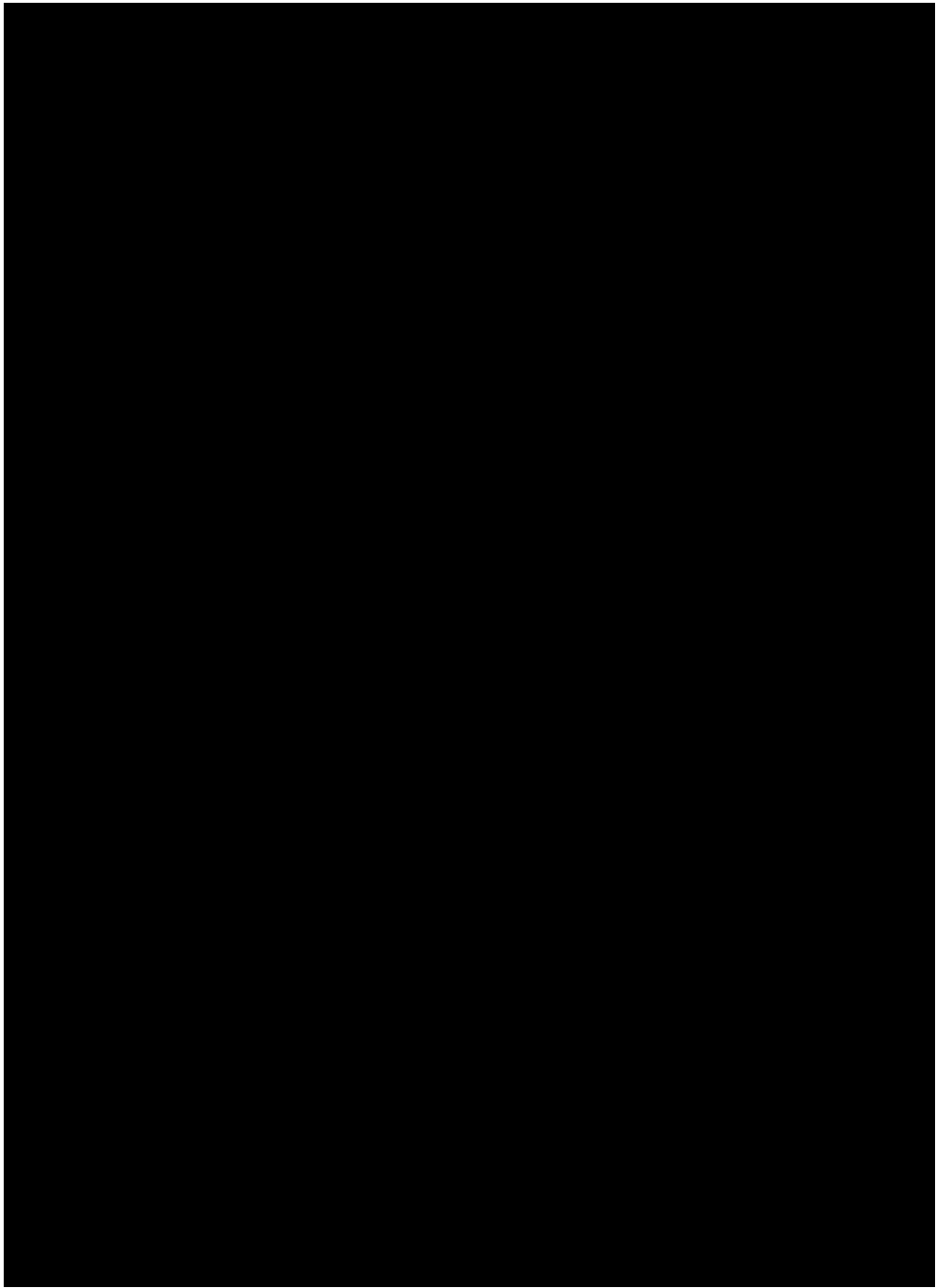


Figure 7.1: Class diagram for the Blast Record class representing all of the info in a BLASTThe



```
...     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.subject[0:75] + '...')
```

If you also read the section [7.3](#)

### 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 at these and deal with them as you see fit.









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

```
>>> blat_gresul t = SearchIO.read('my_blat.psl', 'blat-psl')
>>> print(blat_gresul t)
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)

Hi t range: [0: 61] (1)

Fragments: 1 (61 columns)

Query - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTAGAGGG

|||||

Hi t - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTAGAGGG

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

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's Basic I/O Format Support](#).

## 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 field 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=htses
>>:tre.em(ai:)-125=:thr@example.com"pai fai wh-125Bi oy3(y)5Bi oareEn(trez)]TJ0-11.955Td[(>>h,)-ccesm(ai:)-
```

<DbName>uni gene</DbName>  
<DbName>uni sts</DbName>







EDKFLHLNYVSDLLI PHPI HLEI LVQI LQCRI KDVPSLHLLRLLFHEYHNLNSLI TSK  
KFI YAFSKRKKRFLWLLYNSYVVECEYLFQFLRKQSSYL RSTSSGVFLERTHLYVKI E  
HLLVCCNSFQRI LCFLKDPFMHYVRYQGKAI LASKGTLI LMKKWKFHLVNFWSYFH  
FWSQPYRI HI KQLSNYSFSFLGYFSSVLENHLVVRNQMLENSFI I NLLTKKFDTI APV  
I SLI GSLSKAQFCTVLGHPI SKPI WTD FSDSDI LDRFCRI CRNLCRYHSGSSKKQVLY  
RI KYI LRLSCARTLARKHKSTVRTFMRRLGSGLLEEFFMEEE"

ORIGIN

1 attttttacg aacctgtgga aatttttgggt tatgacaata aatctagttt agtacttgtg  
61 aaacgtttta ttactcgaat gtatcaacag aattttttga tttcttcggt taatgattct  
121 aaccaaaaag gattttgggg gcacaagcat tttttttctt ctcatttttc ttctcaaag  
181 gtatcagaag gttttggagt cattctggaa attccattct cgtcgcaatt agtatcttct  
241 ctgaagaaa aaaaaatacc aaaatatcag aatttatc45(aaaaaataMattttc)-52aaaat(attctg)]TJ0-11. 955Td301a  
61 atatctcgg

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 specified only one PubMed ID to search for, record contains only one item. This item is a dictionary containing information about our search term, as well as all the related items that were found:

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

The "LinkSetDb"

## 9.8 EGQuery: Global Query - counts for search terms

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

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





### 9.12.1 Parsing Medline records

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

### 9.12.2 Parsing GEO records

GEO (



PROT SIM      ORG=9986; PROTI =126722851; PROTI D=NP\_001075655. 1; PCT=76. 90; ALN=288

...

PROT SIM      ORG=9598; PROTI =114619004; PROTI D=XP\_519631. 2; PCT=98. 28; ALN=288

UpUNT          388

EQUENCEMACC=BC0267185. 1; TI Dg455013076; EQTYPE=mRNA.

EQUENCEMACC=NMP\_0\_0051. 2; TI Dg11629-5606; EQTYPE=mRNA.

EQUENCEMACC=D190025. 1; TI Dg2194166; EQTYPE=mRNA.

EQUENCEMACC=D190005. 1; TI Dg2194126; EQTYPE=mRNA.

EQUENCEMACC=BC0158785. 1; TI Dg146184206; EQTYPE=mRNA.

EQUENCEMACC=CR4079631. 1; TI Dg4711\_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("S0", "?"))
...     print("")
```

The output for this looks like:

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

```

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

```

So, we expect to find 814 Entrez Nucleotide records (this is the number I obtained in 2008; it is likely to increase in the future). If you find some ridiculously high number of hits, you may want to reconsider if you really want to download all of them, which is our next step:

```

814 from Bio import Entrez
815 Entrez.email = "james@alum.mit.edu"
816 handle = Entrez.efetch(db="nucleotide", id="123456789", rettype="text", retmode="text")
817 record = handle.read()
818 handle.close()
819 print(record)

```



```
>>> 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)

```





```
fetch_handle = Entrez.efetch(db="pubmed",
                             rettype="medline", retmode="text",
                             retstart=start, retmax=batch_size,
                             webenv=search_results["WebEnv"],
                             query_key=search_results["QueryKey"])

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



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

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

We can now print out some information about this record:

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

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

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

This prints

2Fe-2S.

Protein which contains at least one 2Fe-2S iron-sulfur cluster: 2 iron atoms  
complexed to 2 inorganic 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("mysingleprosite record.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    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 (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 all 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"]
'3.5111.955T55Td>>> record["L-525(with)955T56Td[.'') as handle:
>>> record[["C3(lipase)40.95are hydrolyze

```

```
>>> ecnumbers = [record["ID"] for record in records]
```

## 10.5 Accessing the ExPASy server

Swiss-Prot, Prosite, and Prosite documentation records can be downloaded from the ExPASy web server at <http://www.expasy.org>. Six kinds of queries are available from ExPASy:

**get\_prodoc\_entry** To download a Prosite documentation record in HTML format

**get\_prosite\_entry** To download a Prosite record in HTML format

**get\_entry**



### 10.5.2 Searching Swiss-Prot

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



6

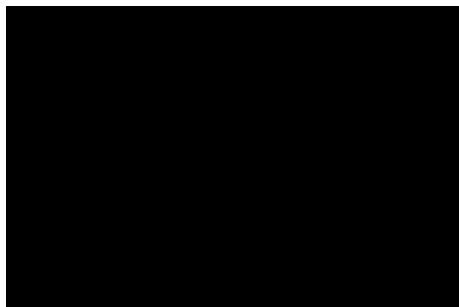
```
>>> result[0]
{'signature_ac': u'PS50948', 'level': u'0', 'stop': 98, 'sequence_ac': u'USERSEQ1', 'start': 16, 'score': 1.0}
>>> result[1]
{'start': 37, 'stop': 39, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00005'}
>>> result[2]
{'start': 45, 'stop': 48, 'sequence_ac': u'USERSEQ1', 'signature_ac': 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 specified 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 should) be replaced 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

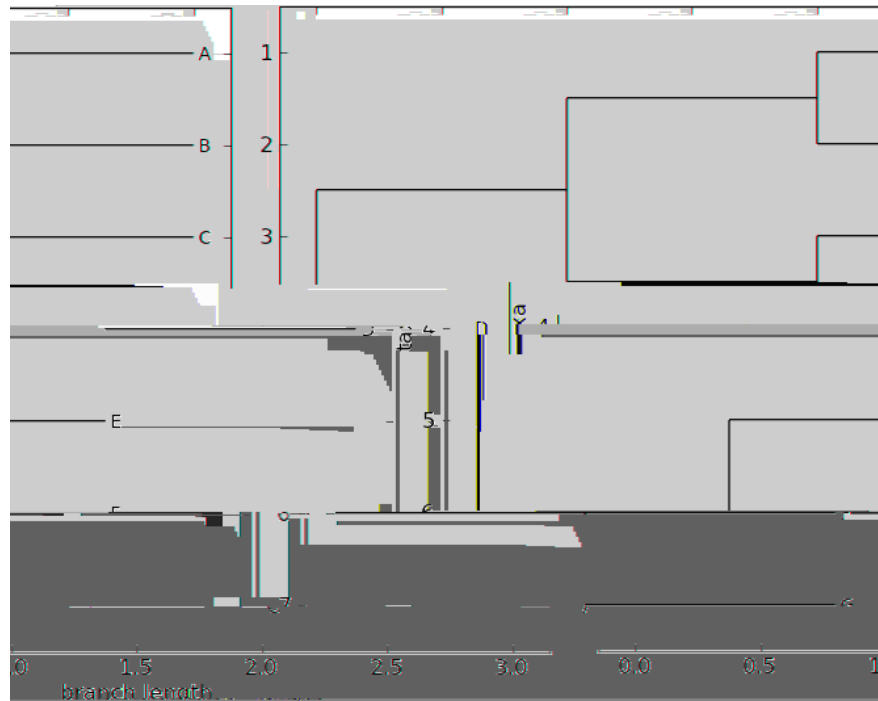


Figure 12.1: A rooted tree drawn with Phyl o. draw.

Note that the file 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):

```
>>> Phyl o. draw(tree)
```

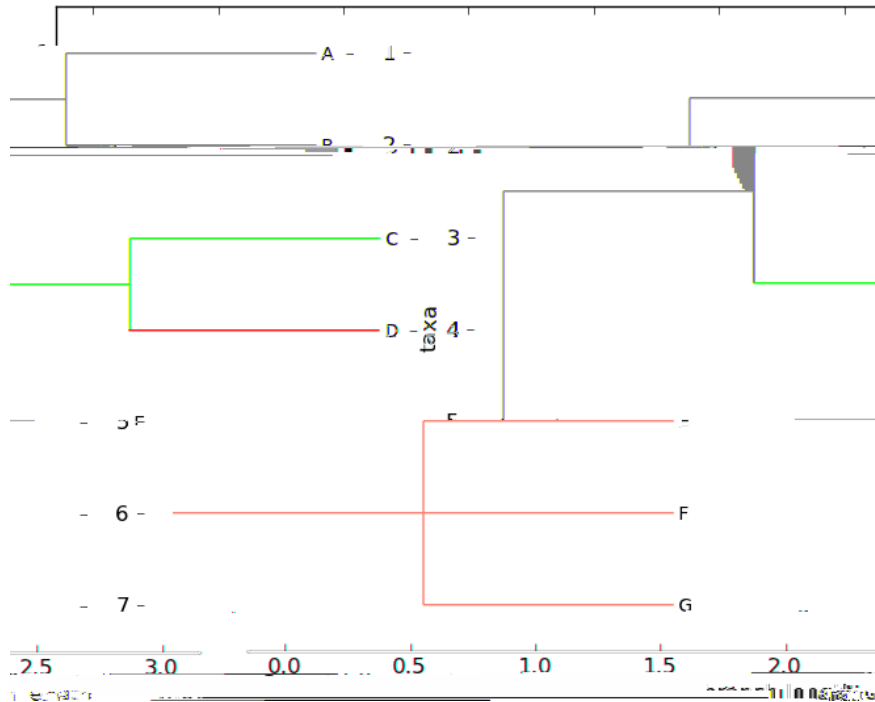


Figure 12.2: A colored tree drawn with Phyl o. draw.

14lthename(a)29 olorh | (a)29 hegre4(w)2thestanzard(a)29 output,ee45357oessethethatw7(ould(a)29 b)-c oesf30(orm9 w







Figure Areotredetrure1(ure)-33plottedrewithrethe



Figure 12.7: A larger tree, using `neato`





#### 12.4.2 Information methods



prune

## 12.6 PAML integration

Biopython 1.58 brought support for PAML ([link](#))

**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 = motifs.create(instances)
```

```
>>> 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 and writing motifs. There are not any really well established standards for storing motifs, but there are a couple of formats that are more used than others.

**13.2.1 JASPAR**

One of the most popular motif databases is







```
>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
--	---	---	---	---	---	---	---	---	---	---	----



Collection CORE



\*\*\*\*\*







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





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
```

```
>>> 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
```



## 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

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

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



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

where the following arguments are defined:

data (required)

Array containing the data for the items.

mask (default: None)

Array of integers showing which data are missing. If `mask[i,j]==0`, then `data[i,j]` is missing. If `mask==None`, then all data are present.

clusterid (default: None)

Vector of `data[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 different (random) initial condition. If `initial_id` is given, the value of `npass` is ignored, as the clustering algorithm behaves deterministically in that case.

`initial_id` (default: None)

Specifies the initial clustering to be used for the EM algorithm. If `initial_id==None`, then a different random initial clustering is used for each of the `npass` runs of the EM algorithm. If `initial_id` is not None

Medoids clustering is deterministic. If `initial_id` is not None, the clustering is deterministic.



In pairwise average-linkage clustering, the distance between two nodes is defined as the average over all pairwise distances between the items of the two nodes.

In pairwise centroid-linkage clustering, the distance between two nodes is defined as the distance



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

$$\alpha = \alpha_{\text{init}} \left(1 - \frac{i}{n}\right);$$

$\alpha_{\text{init}}$  is the initial value of  $\alpha$  as specified by the user,  $i$  is the number of the current iteration step, and  $n$  is the total number of iteration steps to be performed. While changes are made rapidly in the beginning of the





`gweight`

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

`gorder`



transpose (default: 0)

Determines if the centroids of the rows of data are to be calculated (transpose==0

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

transpose

## 14.8 Example calculation

## Chapter 15

The logistic regression model gives us appropriate values for the parameters  $\beta_0$ ,  $\beta_1$ ,  $\beta_2$  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*, *ycxH* and *yxiB*, *yxiA*:

Table 15.2: Adjacent gene pairs of unknown operon status.

Gene pair		Intergene distance $x_1$	Gene expression score $x_2$
<i>yxcE</i>	<i>ycxH</i>	6	-173.143442352
<i>yxiB</i>	<i>yxiA</i>	309	-271.005880394

The logistic regression model classifies *yxcE*, *ycxH* as belonging to the same operon (class OP), while *yxiB*, *yxiA* are predicted to belong to different operons:

```
>>> print("yxcE, ycxH: ", LogisticRegressionClassifier(model, [6, -173.143442352]))
yxcE, ycxH: 1
>>> print("yxiB, yxiA: ", LogisticRegressionClassifier(model, [309, -271.005880394]))
yxiB, yxiA: 0
```

*yxcE*, 0





```

...
>>> 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 22E16 Tf 2g98 0 T 2g98277(h)-333(category)83(.)-444(F)83`

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 corresponds to a prediction accuracy of 76%.

### 15.3 Naïve Bayes

This section will describe the Bio.NaiveBayes module.

### 15.4 Maximum Entropy

This section will describe the Bio.MaximumEntropy module.

### 15.5 Markov Models

This section will describe the Bio.MarkovModel and/or Bio.HMM.MarkovModel

## Chapter 16

# Graphics including GenomeDiagram

The Bio.Graphics module depends on the third party Python library [ReportLab](#). Although focused on



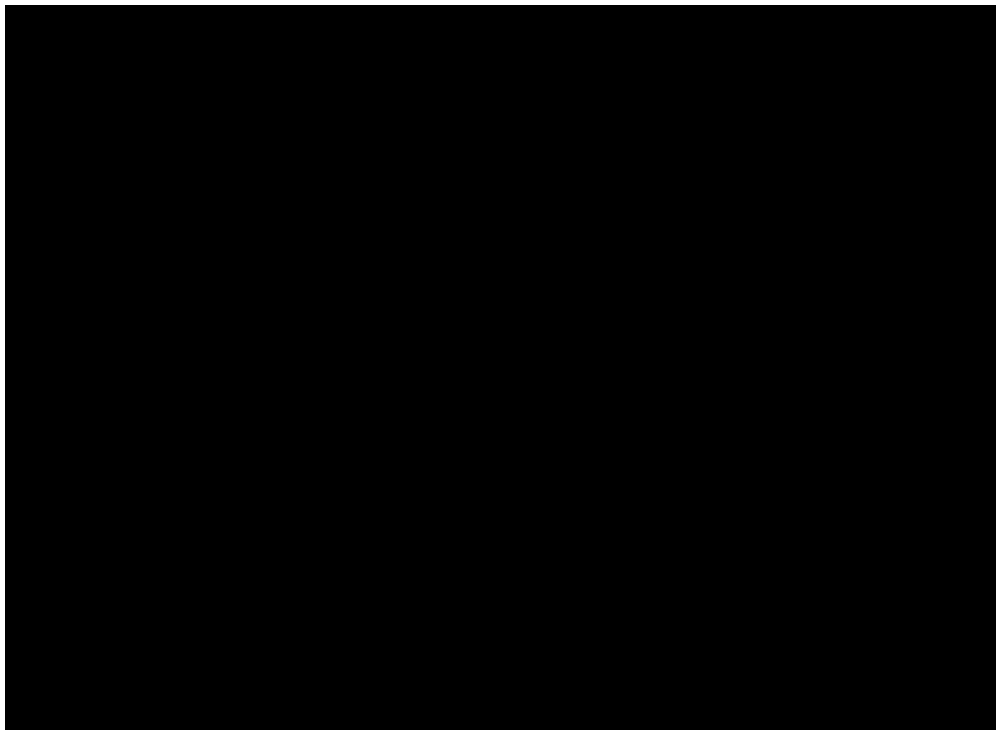


Figure 16.1: Simple linear diagram for *Yersinia pestis* biovar *Microtus* plasmid pPCP1.



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 Mi crotus 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). So if you have a SeqFeature object, you can build a diagram like this:

```
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)
```

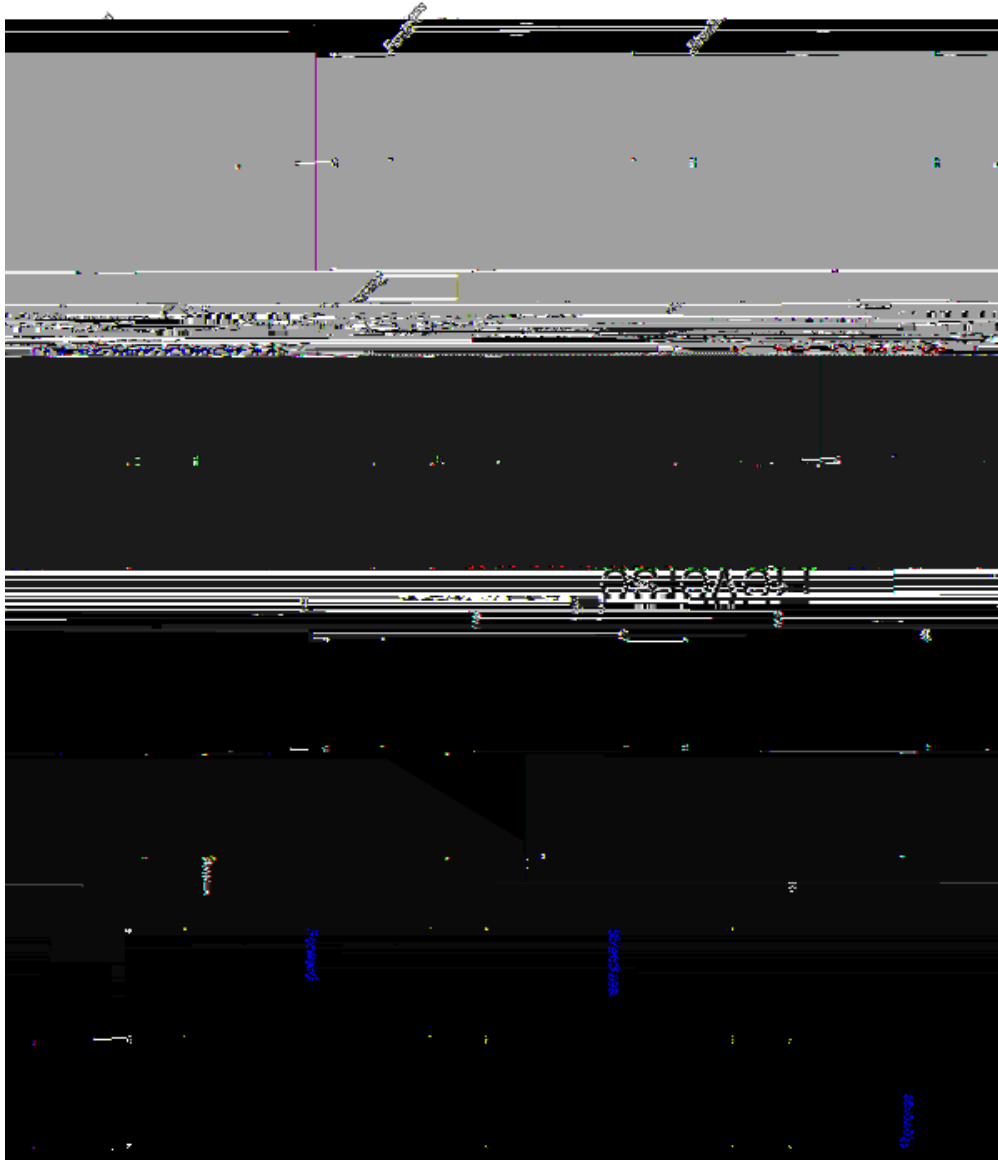


Figure 16.3: Simple GenomeDiagram showing label options. The top plot in pale green shows the default label settings (see Section 16.1.5) while the rest show variations in the label size, position and orientation (see Section 16.1.6).

### 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 different sigils (see Section 16.1.7)





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



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

gd_digram.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

You can download these using Entrez if you like, see Section 9.6 for more details. For the third record we've worked out where the phage is integrated into the genome, and slice the record to extract it (with the features preserved, see Section 4.6), and must also reverse complement to match the orientation of the first two phage (again preserving the features, see Section 4.8):

```
from Bio import SeqIO

A_rec = SeqIO.read("NC_002703.gbk", "gb")
B_rec = SeqIO.read("AF323668.gbk", "gb")
C_rec = SeqIO.read("NC_003212.gbk", "gb")[2587879:2625807].reverse_complement(name=True)
```

The figure we are imitating used different colors for different gene functions. One way to do this is to

$i += 1$

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

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

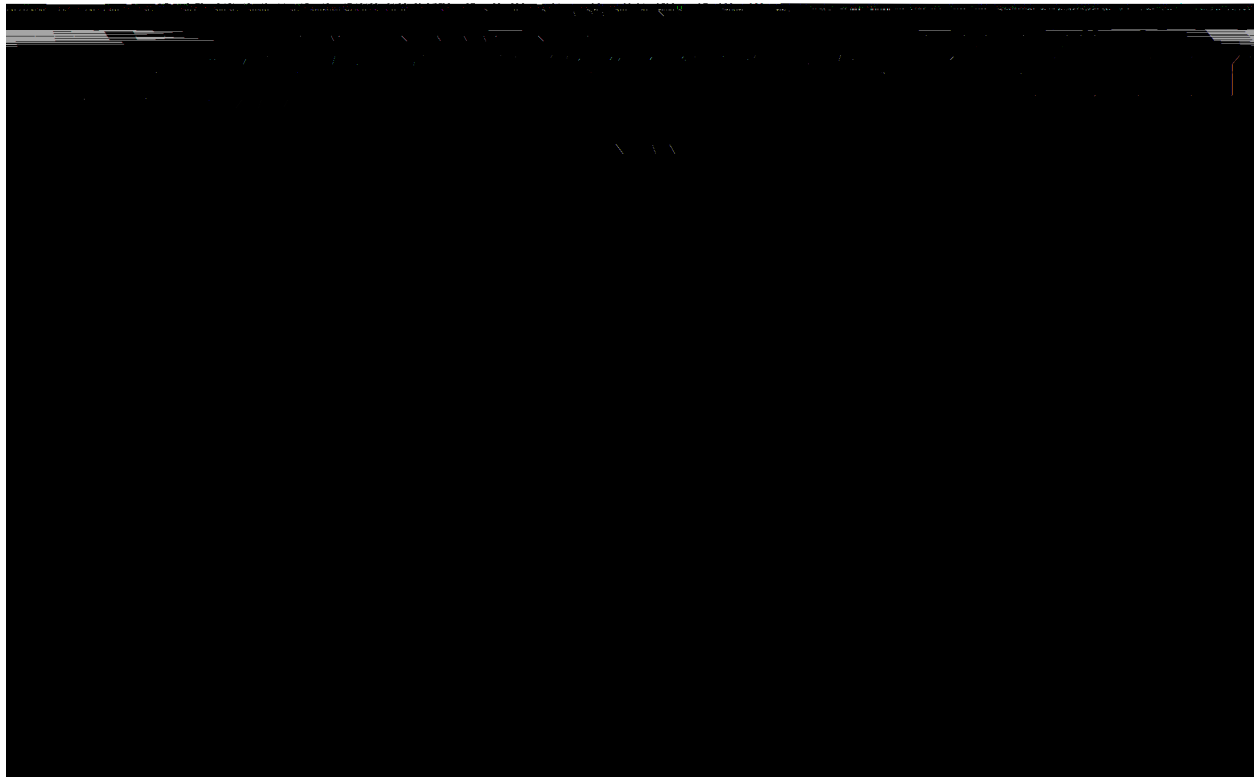


Figure 16.10: Linear diagram with three tracks for *Lactococcus* phage Tuc2009 (NC\_002703), bacteriophage bIL285 (AF323668), and prophage 5 from *Listeria innocua* Clip11262 (NC\_003212) plus basic cross-links shaded by percentage identity (see Section 16.1.11).

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 there(P

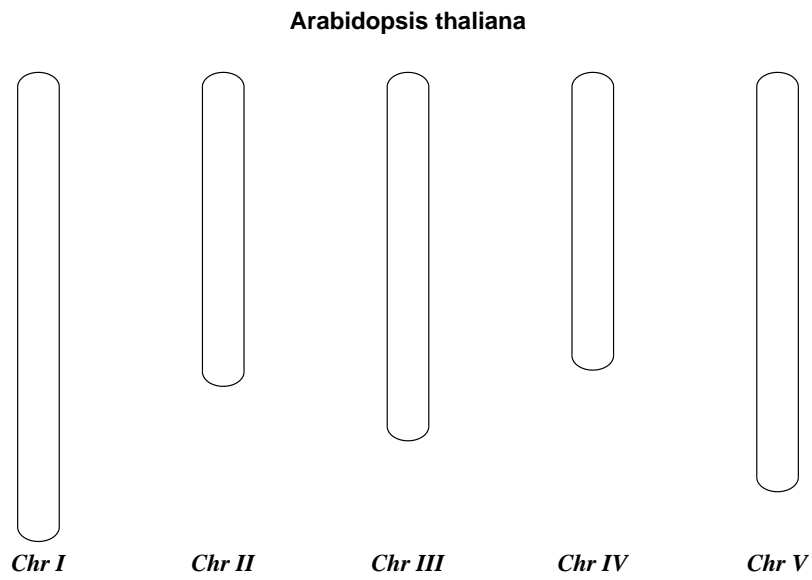


Figure 16.12: Simple chromosome diagram for *Arabidopsis thaliana*.





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



```
>>> 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'
```

```
/find/compound/300-310/mol_weight -> KEGG.find("compound", "300-310", "mol_weight")  
/get/hsa:10458+ece:Z5100/aaseq -> 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 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")
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 file once using

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 using 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")
```





### 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:

```

NQIQGVCSPDSGEFMVTFETVMEIKILHK...GVA - length 355, strand 1, 41:1109
WDVKTVTGVLHHPFHLTFSLCPEGATQSGR...VKR - length 111, strand -1, 491:827
KSGELRQTTPPASSTLHLRLILQRSGVMMEI...NPE - length 285, strand 1, 1030:1888
RALTGLSAPGIRSQTSCDRLRELRYVPVSL...PLQ - length 119, strand -1, 2830:3190
RRKEHVSKRRRPQKRPRRRRFFHRLRPPDE...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 file (as visualised in Section 16.1.9).

If however all you want to find are the locations of the open reading frames, then it is a waste of time to translate every possible codon, including doing the reverse complement to search the reverse strand too. All you need to do is search for the possible stop codons (and their reverse complements). Using regular







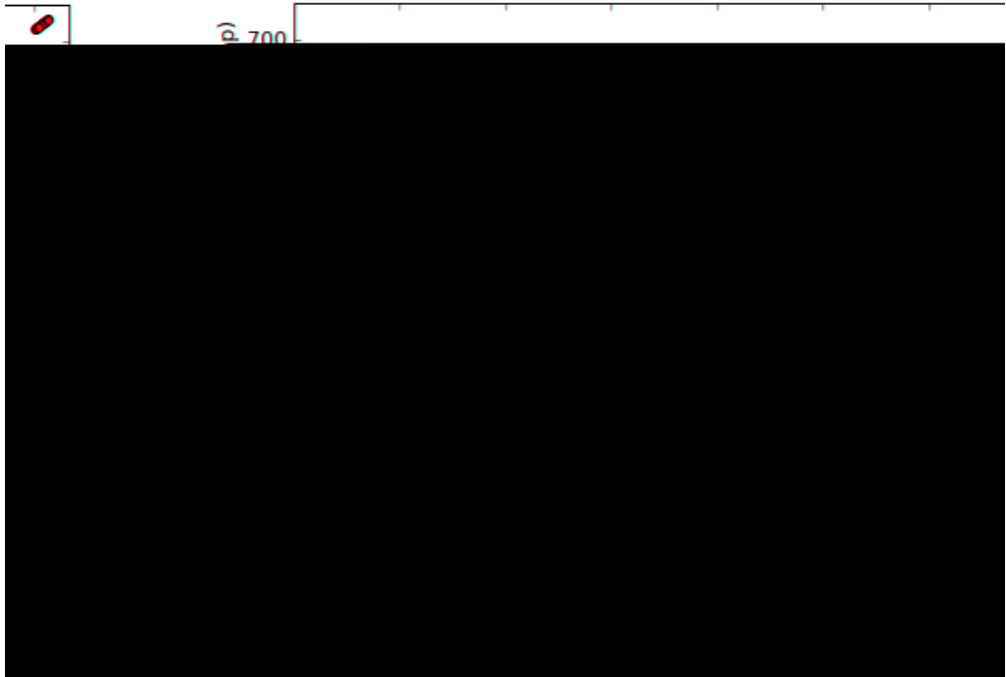
Figure 18.3: Nucleotide dot plot of two orchid sequence lengths (using pylab's imshow function).

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)

```





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









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

## Chapter 19

# The Biopython testing framework





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`

```
import unittest
from Bio import Biopam
```

```
class BiopamTestAddition(unittest.TestCase):
```

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

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

```
class BiopamTestDivision(unittest.TestCase):
```

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

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

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

ytests:-514454u 4anoic33841eetea28(y)433341estsyyouwoin tsc131(et.TJ 0 g 0 G /F14 9.9626 Tf -9.963 -195926 Td [( )]  
(y)eo33his8(y)43scripthd n(y)eo3yoiyyym4tidyglivryco



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 `anr6FreqMat4(anr6instance36 Td [(i.)]TJ -34.205 Tf3.9484 -16.9936 Td [(i`

- i. Full matrix size:  $N \times N$
- 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`: expected frequencies table. Used if provided, if not, generated from the `acc_rep_mat`.
- (c) `logbase`: base of logarithm for the log-odds matrix. Default base 10.
- (d) `round_digit`: number after decimal digit to which result should be rounded. Default zero.

### 20.2.2 FreqTable

FreqTable.FreqTableOptions

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 stuff 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 benefits over plain text information:

1. They provide a standard way to deal with information stored in different ways. The text information can be in a file, 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



# 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 differentiation". *Proceedings of the National Academy of Science USA* **96** (6): 2907{2912 (1999). [doi:10.1073/pnas.96.6.2907](https://doi.org/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](https://doi.org/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>