

# Biopython Tutorial and Cookbook



#### 4.3.2 Locations

<b>8</b>	<b>Accessing NCBI's Entrez databases</b>	<b>85</b>
8.1	Entrez Guidelines . . . . .	86
8.2	EInfo: Obtaining information about the Entrez databases . . . . .	86
8.3	ESearch: Searching the Entrez databases . . . . .	88



#### 14.1.7 Identifying open reading frames











## 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 programming language. However the task of parsing these files can be frustrated by the fact that the formats can change quite regularly, and that formats may contain small subtleties which can break even the most well designed parsers.

We are now going to briefly introduce the Bio.SeqIO module – you can find out more in Chapter 5. We'll start with an online search for our friends, the lady slipper orchids. To keep this introduction simple, we're



The code in these modules basically makes it easy to write Python code that interact with the CGI scripts on these pages, so that you can get results in an easy to deal with format. In some cases, the results can be tightly integrated with the Biopython parsers to make it even easier to extract information.

## 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())
```

For some biological uses, you may actually want an overlapping count (i.e. 3 in this t4(3)-315i use1(oux(appm1(oup(ape50)



Here is an example of adding a generic nucleotide sequence to an unambiguous IUPAC DNA sequence, resulting in an ambiguous nucleotide sequence:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_nucleotide
>>> from Bio.Alphabet import IUPAC
>>> nuc_seq = Seq("GATCGATGC", generic_nucleotide)
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> nuc_seq
Seq('GATCGATGC', NucleotideAlphabet())
>>> dna_seq
Seq('ACGT', IUPACUnambiguousDNA())
>>> nuc_seq + dna_seq
Seq('GATCGATGCACGT', NucleotideAlphabet())
```

### 3.6 Nucleotide sequences and (reverse) complements







the moleinhatihat



C	CTT L	CCT P	CAT H	CGT R	T
C	CTC L	CCC P	CAC H	CGC R	C
C	CTA L	CCA P	CAA Q	CGA R	A
C	CTG L	CCG P	CAG Q	CGG R	G
A	ATT I(s)	ACT T	AAT N	AGT S	T
A	ATC I(s)	ACC T	AAC N	AGC S	C
A	ATA M(s)	ACA T	AAA K	AGA Stop	A
A	ATG M(s)	ACG T	AAG K	AGG Stop	G
G	GTT V	GCT A	GAT D	GGT G	T
G	GTC V	GCC A	GAC D	GGC G	C
G	GTA V	GCA A	GAA E	GGA G	A
G	GTG V(s)	GCG A	GAG E	GGG G	G

You may find these following properties useful – for example if you are trying to do your own gene finding:

```
>>> mi_to_table.stop_codons
['TAA', 'TAG', 'AGA', 'AGG']
>>> mi_to_table.start_codons
['ATT', 'ATC', 'ATA', 'ATG', 'GTG']
>>> mi_to_table.forward_table["ACG"]
'T'
```

If you actually want to do this, you can be more explicit by using the Python `id` function,

```
>>> id(seq1) == id(seq2)
```

```
False
```

```
>>> id(seq1) == id(seq1)
```

```
True
```

```
>>> mutable_seq
MutableSeq('GCCATGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
>>> mutable_seq.reverse()
>>> mutable_seq
```

```
>>> unk_dna.complement()
UnknownSeq(20, alphabet = IUPACAmbiguousDNA(), character = 'N')
>>> unk_dna.reverse_complement()
UnknownSeq(20, alphabet = IUPACAmbiguousDNA(), character = 'N')
>>> unk_dna.transcribe()
UnknownSeq(20, alphabet = IUPACAmbiguousRNA(), character = 'N')
>>> unk_protein = unk_dna.translate()
>>> unk_protein
UnknownSeq(6, alphabet = ProteinAlphabet(), character = 'X')
>>> print unk_protein
XXXXXX
>>> len(unk_protein)
6
```

You may be able to find a use for the UnknownSeq

## Chapter 4

# Sequence Record objects

Chapter

annotations

Working with per-letter-annotations is similar, `letter_annotations` is a dictionary like attribute which will let you assign any Python sequence (i.e. a string, list or tuple) which has the same length as the sequence:

```
>>> simple_seq_r.letter_annotations["phred_quality"] = [40, 40, 38, 30]
>>> print simple_seq_r.letter_annotations
{'phred_quality': [40, 40, 38, 30]}
>>> print simple_seq_r.letter_annotations["phred_quality"]
[40, 40, 38, 30]
```

The `dbxrefs` and `features`







**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

```
>>> print sub_record.features[0]
type: gene
location: [42:480]
ref: None:None
strand: 1
qualifiers:
  Key: db_xref, Value: ['GeneID: 2767712']
  Key: gene, Value: ['pim']
  Key: locus_tag, Value: ['YP_pPCP05']

>>> print sub_record.features[1]
D/DS gene
```

```
strand: 1
qualifiers:
```

See Sections 14.1.4 and 14.1.5 for some FASTQ example where the per-letter annotations (the read quality scores) are also sliced.



## Chapter 5

# Sequence Input/Output

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



```

first_record = record_iterator.next()
print first_record.id
print first_record.description

second_record = record_iterator.next()
print second_record.id
print second_record.description

handle.close()

```

Note that if you try and use `.next()` and there are no more results, you'll either get back the special Python object `None` or a `StopIteration` exception.

One special case to consider is when your sequence files have multiple records, but you only want the first one. In this situation the following code is very concise:

```

from Bio import SeqIO
first_record = SeqIO.parse(open("Is_orchid.gb"), "genbank").next()

```

A word of warning here – using the `.next()` method like this will silently ignore any additional records in the file. If your files have *one and only one* record, like some of the online examples later in this chapter, or a GenBank file for a single chromosome, then use the new `Bio.SeqIO.read()` function instead. This will

Z78439.1

```
Seq(' CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC', IUPACAmbiguousDNA())
```

592

The first record

Z78533.1

```
Seq(' CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', IUPACAmbiguousDNA())
```

740

You can of course still use a for loop with a list of SeqRecord objects. Using a list is much more flexible

In general, the annotation values are strings, or lists of strings. One special case is any references in the file get stored as reference objects.

Suppose you wanted to extract a list of the species from the [ls\\_orchid.gbk](#) GenBank file. The information we want, *Cypripedium irapeanum*, is held in the annotations dictionary under 'source' and 'organism', which

```
from Bio import SeqIO
handle = open("Is_orchid.fasta")
all_species = []
```

```
from Bio import Entrez
from Bio import SeqIO
handle = Entrez.efetch(db="nucleotide", rettype="gb", id="6273291")
seq_record = SeqIO.read(handle, "gb") #using "gb" as an alias for "genbank"
handle.close()
print "%s with %i features" % (seq_record.id, len(seq_record.features))
```

Assuming your network connection is OK, you should get back:



This time the keys are:

Now, recall the



```
from __future__ import with_statement #Needed on Python 2.5
from Bio import SeqIO
```

```
with in_handle = open("Is_orchid.gb") :
    with out_handle = open("my_example.fasta", "w") :
```

Now list comprehensions have a nice trick up their sleeves, you can add a conditional statement:

```
records = [make_rc_record(rec) for rec in SeqIO.parse(in_handle, "fasta") if len(rec)<700]
```



## Chapter 6

# Sequence Alignment Input/Output, and Alignment Tools

### 6.1.1 Single Alignments

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

```
# ST0Bk0LM#
```

```
1
```

```
1
```

```
C#
```

```
C#
```

```
1
```

```
-HHHHHHHHHHHHHHHH--HHHHHHHH--HHHHHHHHHHHHHHHHHHHHHHHH--e
```

```
---S-T...CHCHHHHCCCCTCCCTTCHHHHHHHHHHHHHHHHHHHHHCTT--e
```





```
AEGDDP--AKAAFDLSQASATEYI GYAWAMVVVI VGATI GI KLFKKFASKA
>Q9T0Q9_BPF1/1-49
AEGDDP--AKAAFDLSQASATEYI GYAWAMVVVI VGATI GI KLFKKFTSKA
>COATB_BPF1/22-73
FAADDATSQAKAAFDLSAQATEMSGYAWALVVLVVGATVGI KLFKKFVSRA
```

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\_seed.faa" then you can load it with almost

Al pha	AAACAA
Beta	AAACCC
Gamma	ACCCAA
Del ta	CCCACC
Epsi l on	CCCAAA
5	6
Al pha	AAAAAC
Beta	AAACCC
Gamma	AACAAC
Del ta	CCCCCA
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 this in using Bi o. Al i gnI O





















the command line before trying it from within Python, as the Biopython wrapper is very faithful to the actual command line API:

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





Even if you have EMBOSS installed, running this command may not work – you might get a message about “command not found” (especially on Windows). This probably means that the EMBOSS tools are not



In this example, we told EMBOSS to write the output to a file, but you *can* tell it to write the output to stdout instead (useful if you don't want a temporary output file to get rid of), and also read the input from stdin (just like in the MUSCLE example in the section above).

## Chapter 7

# BLAST

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

```
>>> my_blast_db = "/home/mdehoon/Data/Genomes/Databases/bsubtilis"
# I used formatdb to create a BLAST database named bsubtilis
# (for Bacillus subtilis) consisting of the following three files:
# /home/mdehoon/Data/Genomes/Databases/bsubtilis.nhr
# /home/mdehoon/Data/Genomes/Databases/bsubtilis.nin
# /home/mdehoon/Data/Genomes/Databases/bsubtilis.nsq
```



## 7.3 Saving BLAST output

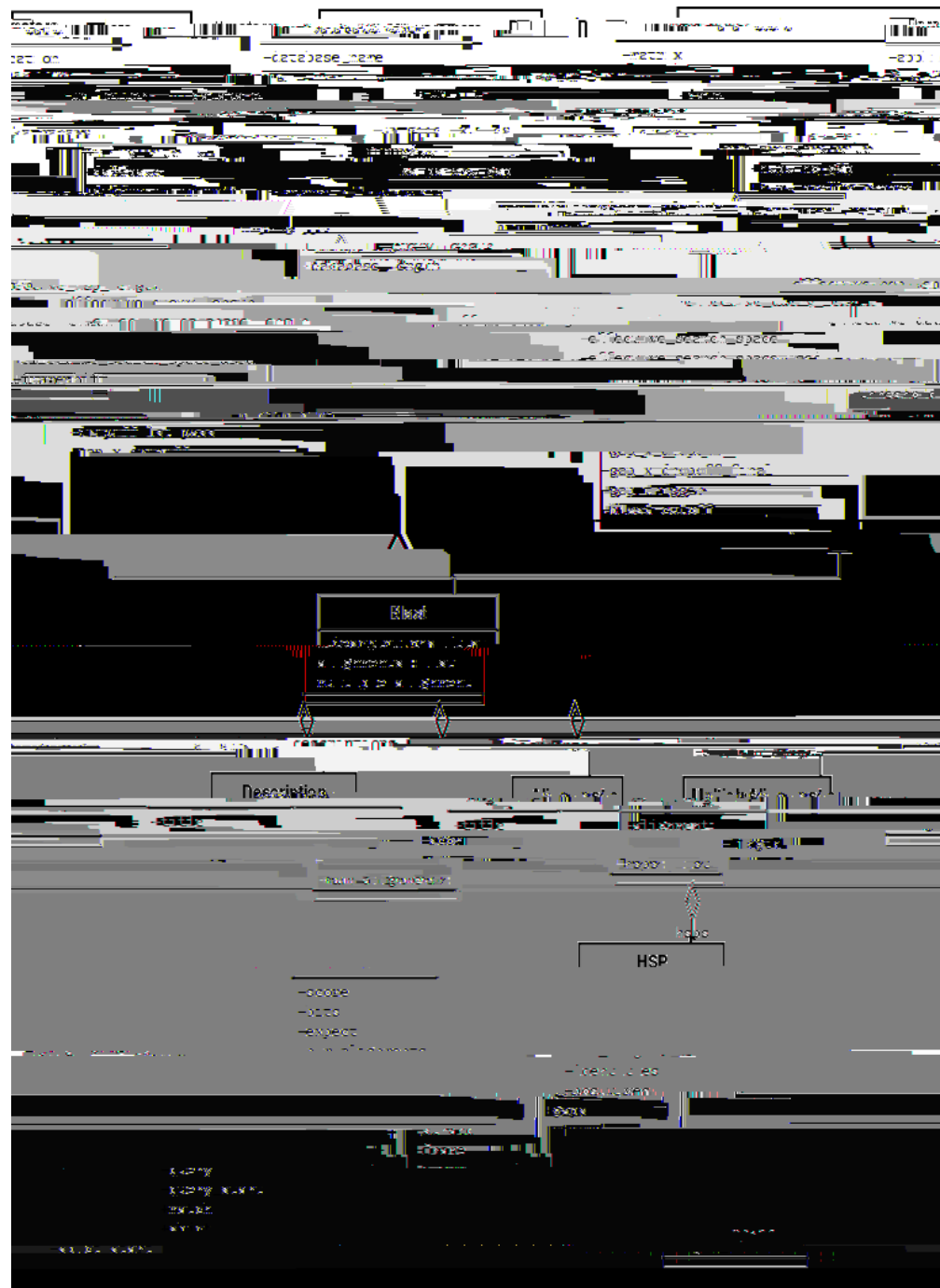
Before parsing the results, it is often useful to save them into a file so that you can use them later without having to go back and re-blasting everything. I find this especially useful when debugging my code that extracts info from the BLAST files, but it could also be useful just for making backups of things you've done.











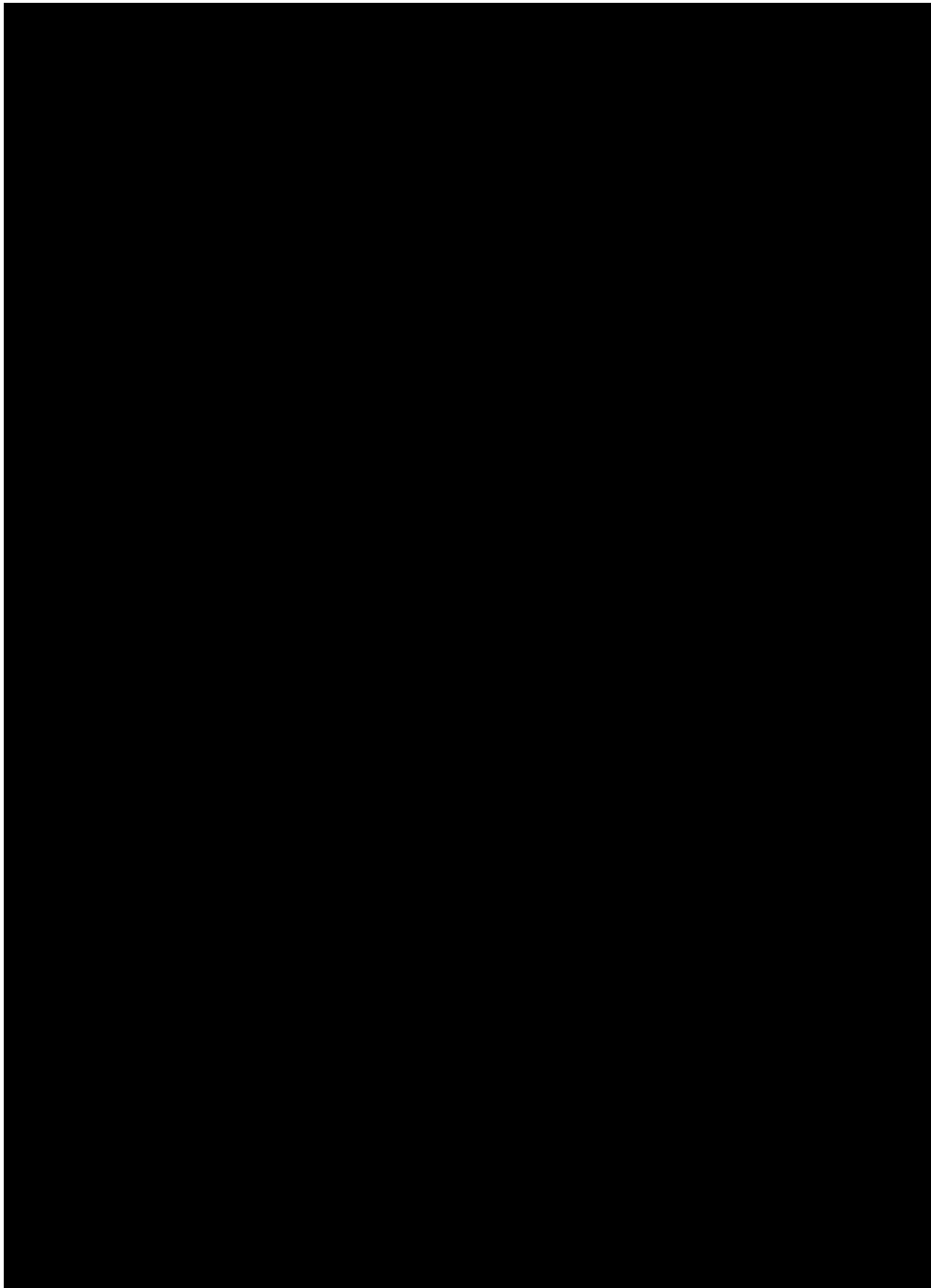


Figure 7.2: Class diagram for the PSIBlast Record class.

```

...     for hsp in alignment.hsps:
...         if hsp.expect < E_VALUE_THRESH:
...             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 [7.4](#)

nyldtodwh(tg).35(is').36(fh)1ounhindtthtgsOnh(y)28(ou).35((p)1(are').36(some)-1ethih)  
 onghsp2014NICBLS2014p2010fig19552Ddh(2)h((p)n)38(p)ndsgdspexpe4NICBLSg1rda

The iterator allows you to deal with huge blast records without any memory problems, since things are read in one at a time. I have parsed tremendously huge files without any problems using this.

### **7.6.3 Finding a bad record somewhere in a huge plain-text BLAST file**

- `item[0]` - The error message
- `item[1]` - The id of the input record that caused the error. This is really useful if you want to





```

<DbLi st>
  <DbName>pubmed</DbName>
  <DbName>protei n</DbName>
  <DbName>nucl eoti de</DbName>
  <DbName>nucore</DbName>
  <DbName>nucgss</DbName>
  <DbName>nucest</DbName>
  <DbName>structure</DbName>
  <DbName>genome</DbName>
  <DbName>books</DbName>
  <DbName>cancerchromosomes</DbName>
  <DbName>cdd</DbName>
  <DbName>gap</DbName>
  <DbName>domai ns</DbName>
  <DbName>gene</DbName>
  <DbName>genomeprj </DbName>
  <DbName>gensat</DbName>
  <DbName>geo</DbName>
  <DbName>gds</DbName>
  <DbName>homol ogene</DbName>
  <DbName>j urnal s</DbName>
  <DbName>mesh</DbName>
  <DbName>ncbi search</DbName>
  <DbName>nl mcatal og</DbName>
  <DbName>omi a</DbName>
  <DbName>omi m</DbName>
  <DbName>pmc</DbName>
  <DbName>popset</DbName>
  <DbName>probe</DbName>
  <DbName>protei ncl usters</DbName>
  <DbName>pcassay</DbName>
  <DbName>pccompound</DbName>
  <DbName>pcsubstance</DbName>
  <DbName>snp</DbName>
  <DbName>taxonomy</DbName>
  <DbName>tool ki t</DbName>
  <DbName>uni gene</DbName>
  <DbName>uni sts</DbName>
</DbLi st>
</el nfoResul t>

```

Since this is a fairly simple XML file, we could extract the information it contains simply by string searching. Using Bi o. Entrez’s parser instead, we can directly parse this XML file into a Python object:

```

>>> from Bio import Entrez
>>> handle = Entrez.ei nfo()
>>> record = Entrez.read(handle)

```

Now record is a dictionary with exactly one key:

```

>>> record.keys()
[u' DbLi st' ]

```

The values stored in this key is the list of database names shown in the XML above:





In this output, you see seven PubMed IDs (including 19304878 which is the PMID for the Biopython



SOURCE chloroplast *Selenipedium aequinoctiale*  
ORGANISM *Selenipedium aequinoctiale*  
Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;  
Spermatophyta; Magnoliophyta; Liliopsida; Asparagales; Orchidaceae;  
Cypripedioideae; *Selenipedium*.  
REFERENCE 1 (bases 1 to 1302)  
AUTHORS Neubig, K. M., Whitten, W. M., Carlswald, B. S., Blanco, M. A.,  
Endara, C. L., Williams, N. H. and Moore, M. JYS (Endara, C. L., Lopez, L., gFFa-52Phylogenetic)-52utyta

601 tcaacatctt ctggagtctt tcttgagcga acacatttat atgtaaaaat agaacatctt  
661 ctagtagtgt gttgtaattc ttttcagagg atcctatgct ttctcaagga tcctttcatg  
721 cattatgttc gatatcaagg aaaagcaatt ctggcttcaa aggggaactct tattctgatg  
781 aagaaatgga aatttcatct tgtgaatfff tggcaatctt attttcactt ttggtctcaa  
841 ccgtatagga ttcatataaa gcaattatcc aactattcct tctcttttct ggggtatfff  
901 tcaagtgtac tagaaaatca tttggtagta agaaatcaaa tgctagagaa ttcatffata  
961 ataaatcttc tgactaagaa attcgatacc atagccccag ttatttctct tattggatca  
1021 ttgtcgaaag ctcaatfff tactgtattg ggtcatccta ttagtaaacc gatctggacc  
1081 gatttctcgg attctgatat tcttgatcga ttttgccgga tatgtagaaa tctttgtcgt  
1141 tatcacagcg gatcctcaaa aaaacaggff ttgtatcgta taaaatatat acttcgactt





## 8.9 ESpell: Obtaining spelling suggestions

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



```
>>> from Bio import Medline
>>> input = open("pubmed_result1.txt")
>>> record = Medline.read(input)
```

The record now contains the Medline record as a Python dictionary:

```
>>> record["PMID"]
'12230038'
>>> record["AB"]
'Bioinformatics research is often difficult to do with commercial software.
The Open Source BioAA, ce Bi1(yopen)-52andce Bj avace olki tsce wi t]
Thaen a Medl(M)1(e)-335(r(c)-1r)ncecc
```





```
>>> record.ID
"HS.2"
>>> record.title
"N-acetyl transferase 2 (arylamine N-acetyl transferase)"
    The EXPRESS and
```

```
>>> record = Entrez.read(handle)
>>> for row in record["eGQueryResult"]:
...     if row["DbName"]=="pubmed":
...         print row["Count"]
463
```

Now we use the Bio.Entrez.efetch function to download the PubMed IDs of these 463 articles:

```
>>> handle = Entrez.esearch(db="pubmed", term="orchid", retmax=463)
>>> record = Entrez.read(handle)
>>> idlist = record["IdList"]
>>> print idlist
```

```
>>> search_author = "Wai ts T"

>>> for record in records:
...     if not "AU" in record:
...         continue
```

```
>>> print record["IdList"][:5]  
['187237168', '187372713', '187372690', '187372688', '187372686']
```

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"      # Always tell NCBI who you are
>>> handle = Entrez.egetquery(term="Opuntia AND rpl16")
>>> record = Entrez.read(handle)
```



```
>>> for record in records:
>>> ...     print "%s, length %i, with %i features" \
>>> ...         % (record.name, len(record), len(record.features))
AY851612, length 892, with 3 features
AY851611, length 881, with 3 features
AF191661, length 895, with 3 features
AF191665, length 902, with 3 features
AF191664, length 899, with 3 features
AF191663, length 899, with 3 features
AF191660, length 893, with 3 features
AF191659, length 894, with 3 features
AF191658, length 896, with 3 features
```



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

For illustrative purposes, this example downloaded the FASTA records in batches of three. Unless you are

## 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))Bio7:
>>>
```

```
>>> 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

```
>>> record.name
'PKC_PHOSPHO_SITE'
>>> record.pdoc
'PDOCC00005'
```

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
...
>>> print 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)
```





## 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

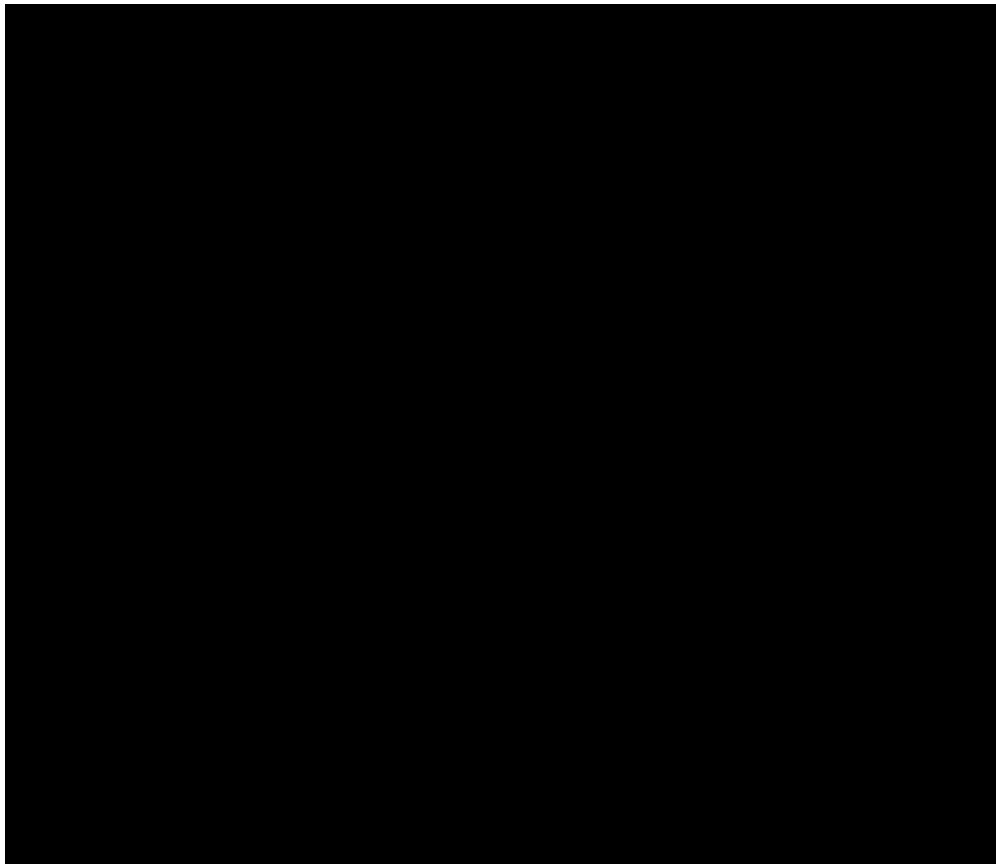


Figure 10.1: UML diagram of the SMCRA data structure used to represent a macromolecular structure.

```
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





#### 10.5.1.1 Duplicate residues

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



Chapter 11

**Bio.PopGen: Population genetics**





## 11.2 Coalescent simulation

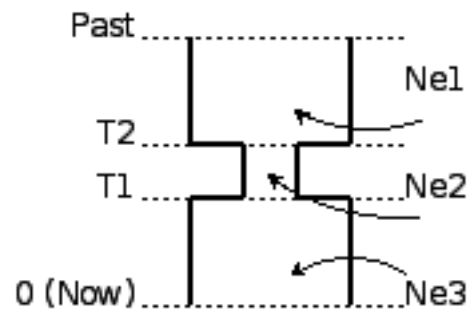


Figure 11.1: A bottleneck









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





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]]
>>> 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  
Iteration: 3 Log-likelihood function: -5.11362294338  
Iteration: 4 Log-likelihood function: -4.74870642433  
Iteration: 5 Log-likelihood function: -4.50026077146  
Iteration: 6 Log-likelihood function: -4.31127773737

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 `Bio.kNN`. To illustrate the use of the  $k$ -nearest neighbor method in Biopython, we will use the same operon data set as in section 12.1.

### 12.2.2 Initializing a $k$ -nearest neighbors model

Using the data in Table 12.1, we create and initialize a  $k$ -nearest neighbors model as follows:

```
>>> from Bio import kNN
>>> k = 3
>>> model = kNN.train(xs, ys, k)
```

where `xs` and `ys` are the same as in Section 12.1.2. Here, `k` is the number of neighbors  $k$  that will be

```

...
>>> 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 13

# Graphics including GenomeDiagram

The Bio. Graphics

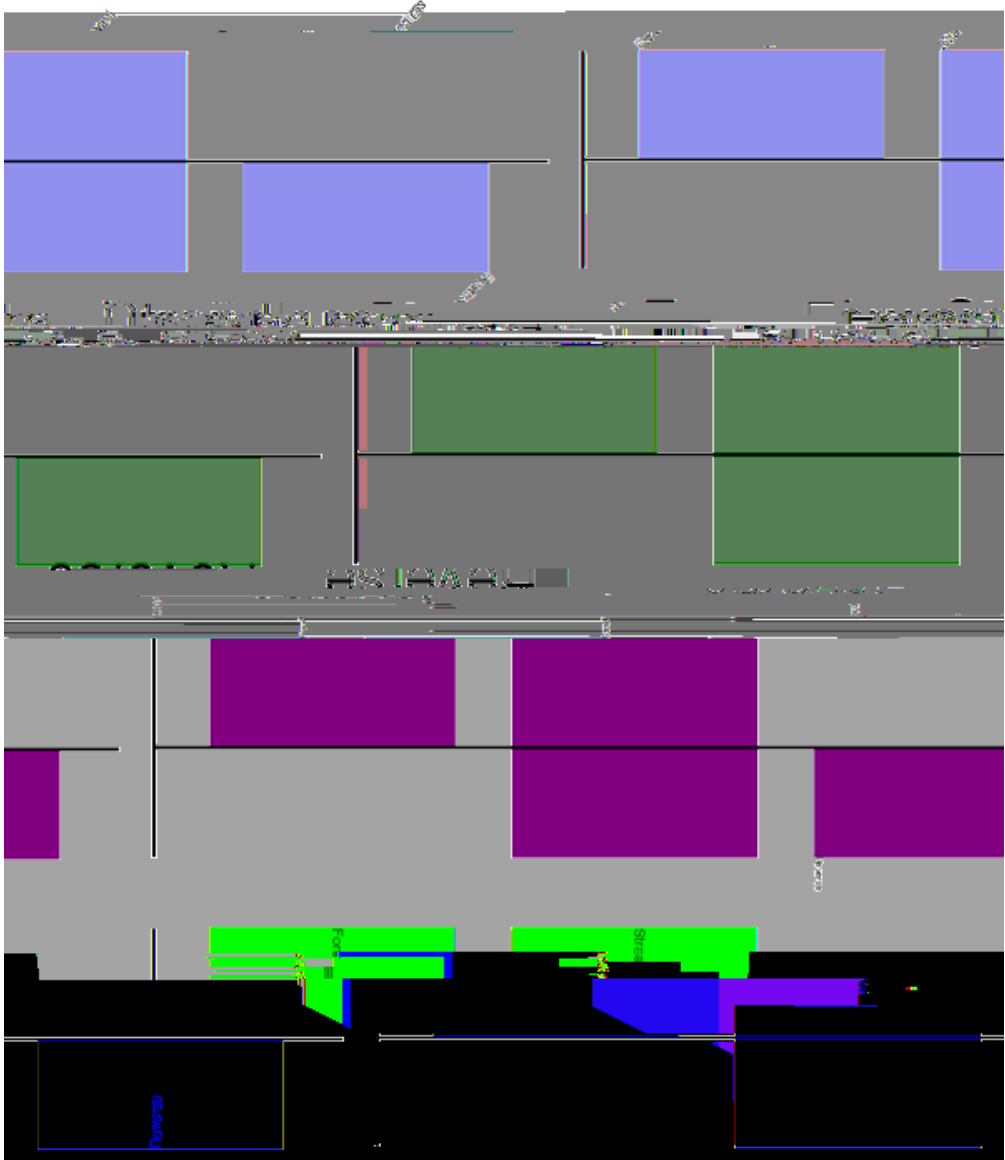




#### 13.1.4 A bottom up example

```
gds_features = gdt_features.new_set()
```

```
#Add three features to show the strand options,  
feature = SeqFeature(FeatureLocare= str=+1050)  
gds_featu.addgds_feate  
feature = SeqFeature(FeatureLocare str=None050)  
gds_featu.addgds_feate  
feature = SeqFeature(FeatureLocare= str=-1050)  
gds_featu.addgds_feate
```



### 13.1.7 Feature sigils





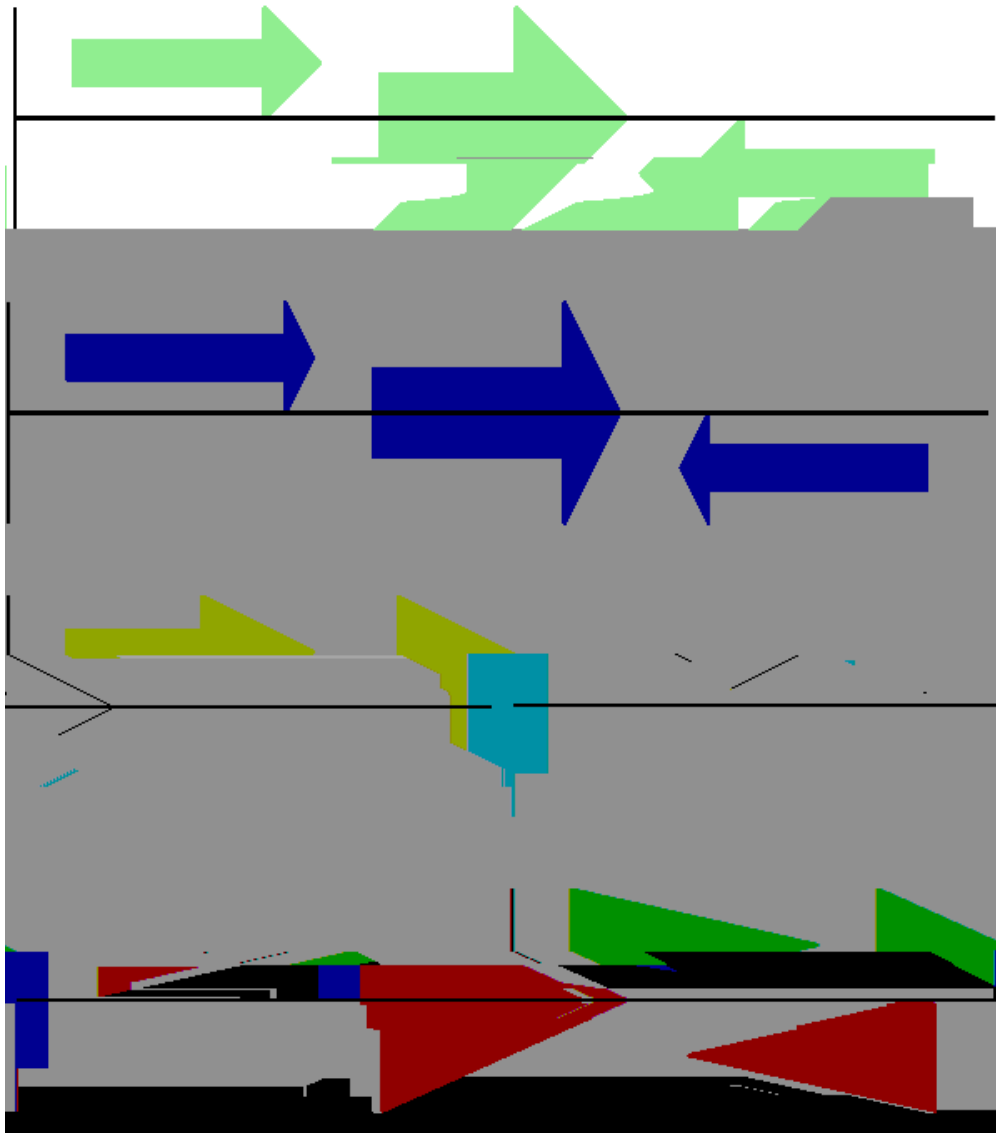


Figure 13.5: Simple GenomeDiagram showing arrow head options (see Section [13.1.7](#))







```

#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 13.7

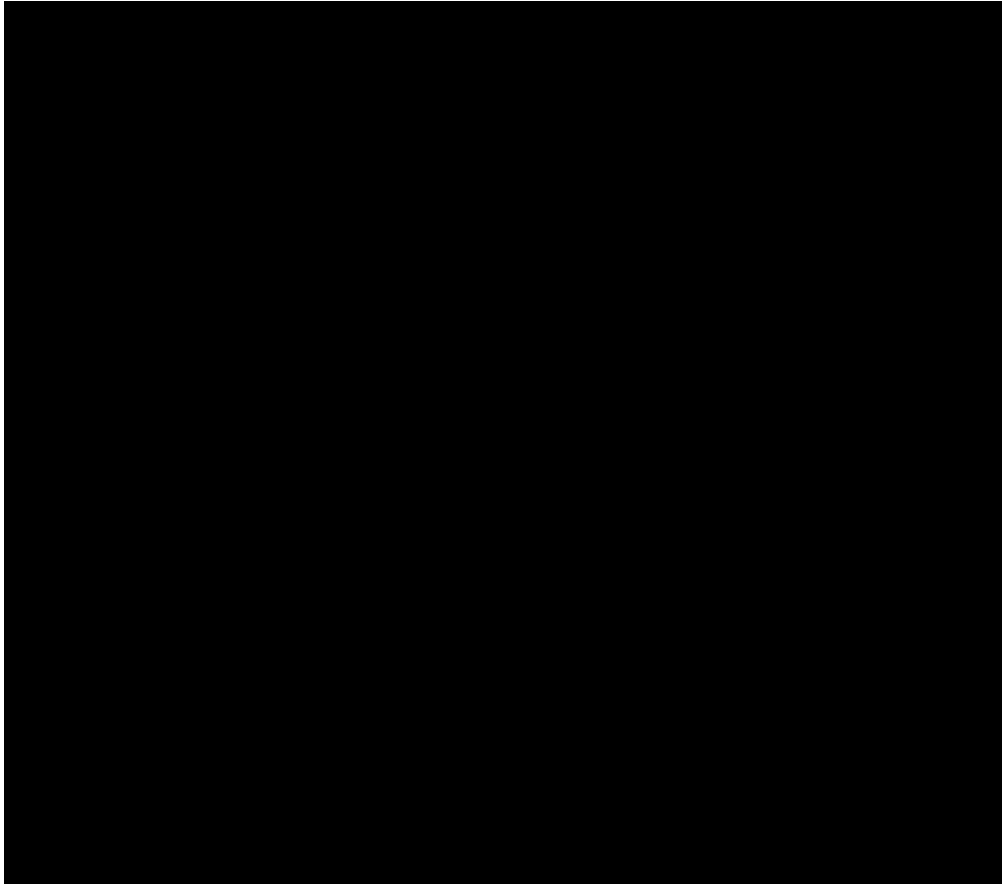


Figure 13.7: Simple chromosome diagram for *Aabidopsisamthaligr*

## Chapter 14

# Cookbook – Cool things to do with it

Biopython now has two collections of “cookbook” examples – this chapter (which has been included in this tutorial for many years and has gradually grown), and <http://biopython.org/wiki/Category:Cookbook>





### 14.1.2 Translating a FASTA file of CDS entries











And the output:

NQIQGVICSPDSGEFMVTFETVMEIKILHK...GVA - length 355, strand 1, 41:1109



```
from Bio import SeqIO
handle = open("Is_orchid.fasta")
sizes = [len(seq_record) for seq_record in SeqIO.parse(handle, "fasta")]
handle.close()

import pylab
```

First of all, we will use `Bio.SeqIO` to parse the FASTA file and compile a list of all the GC percentages. Again, you could do this with a for loop, but I prefer the list comprehension used here:

```
from Bio import SeqIO
from Bio.SeqUtils import Gc
seq_records = SeqIO.parse('fastafile.fasta', 'fasta')
gc_percentages = [Gc(record.seq) for record in seq_records]
```





```
pylab.gray()
pylab.scatter(x, y)
pylab.xlim(0, len(rec_one)-window)
pylab.ylim(0, len(rec_two)-window)
pylab.xlabel("%s (length %i bp)" % (rec_one.id, len(rec_one)))
pylab.ylabel("%s (length %i bp)" % (rec_two.id, len(rec_two)))
pylab.title("Dot plot using window size %i\n(allowing no mis-matches)" % window)
pylab.show()
```

That should pop up a new window showing the graph in Figure

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



### 14.3.3 Position Specific Score Matrices

Position specific score matrices (PSSMs) summarize the alignment information in a different way than a



```
C  0.0 7.0 0.0 0.0
A  7.0 0.0 0.0 0.0
T  0.0 0.0 0.0 7.0
T  1.0 0.0 0.0 6.0
...
```

You can access any element of the PSSM by subscripting like `your_pssm[sequence_number][residue_count_name]`.

```
expect_freq = {  
  'A' : .3,  
  'G' : .2,  
  'T' : .3,  
  'C' : .2}
```

The expected should not be passed as a raw dictionary, but instead by passed as a `UtsMat.FfreTable`}



Once you've got your log odds matrix, you can display it prettily using the function `print_mat`. Doing this on our created matrix gives:

```
>>> my_lom.print_mat()
D   6
E  -5   5
H -15 -13  10
K -31 -15 -13   6
R -13 -25 -14  -7   7
   D   E   H   K   R
```

Very nice. Now we've got our very own substitution matrix to play with!

## Chapter 15

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

We generate the corresponding output with `python run_tests.py -g test_Biospay.py`, and check the output of `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-and(h)28(h)312(h)1nput  
hps(e)n10n2n1n13n2dta  
(ou)072matchn28(h)372li(n)1(e)-1tetn2n1p  
with(e)n1teed  
We n28(ain28(tn)290(aln)1ln)290((th)1(e)-912mop)-87dri(n)290(Bip)28(ye)1tthn(n)1(e)-1ttseaidr(imlee)290((h)1n



```
result = Biospam.division(10.0, -2.0)
self.assertEqual(result, -5.0)
```

```
if __name__ == "__main__":
```











(e) Generating a log-odds matrix (LOM)

Use:

```
LOM=SubsMat._build_log_odds_mat(SFM[, logbase=10, factor=10.0, round_digits=1])
```

i. Accepts an SFM.

ii. Logbase: base of the logarithm used to generate the log-odds values.

Conversely, the residue frequencies or counts can be passed as a dictionary. Example of a count dictionary (3-letter alphabet):

{ 'A': 35, 'B': 65, 'C': 100 }

Which means that an expected data count would give a 0.5 frequency for 'C', a 0.325 probability of 'B' and a 0.175 probability of 'A' out of 200 total, sum of A, B and C)

A frequency dictionary for the same data would be:

{ 'A': 0.175, 'B': 0.325, 'C': 0.5 }

Summing uh67,T/o915.9403-366aras84(,)-ng3-366a-33367sed as aaucnc43-366as84(,3(o94(same)66an)1(nun)28(t)]Ta-









## Chapter 18

# 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

```
>>> my_info = 'A string\n with multiple lines.'
>>> print my_info
A string
  with multiple lines.
>>> import cStringIO
>>> my_info_handle = cStringIO.StringIO(my_info)
>>> first_line = my_info_handle.readline()
>>> print first_line
A string

>>> second_line = my_info_handle.readline()
>>> print second_line
  with multiple lines.
```