Biopython Tutorial and Cookbook

2. *How should I capitalize "Biopython"? Is "BioPython" OK?*
   The correct capitalization is "Biopython", not "BioPython" (even though that would have matched BioPerl, BioJava and BioRuby).

3. *How do I find out what version of Biopython I have installed?*
   Use this:

```
>>> import Bio
>>> print Bio.__version__
...
```

   If the "`import Bio`" line fails, Biopython is not installed. If the second linud(d)-e faiyatc395((evr58(ver)1(cion)-46139

14. *Why doesn't* `Bio.Blast` *work with the latest plain text NCBI blast output?*
    The NCBI keep tweaking the plain text output from the BLAST tools, and keeping our parser up to date was an ongoing struggle. We recommend you use the XML output instead, which is designed to be read by a computer program.

15. *Why doesn't* `Bio.Entrez.read()`

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(aple50)

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:

```
>>> mito_table.stop_codons
['TAA', 'TAG', 'AGA', 'AGG']
>>> mito_table.start_codons
['ATT', 'ATC', 'ATA', 'ATG', 'GTG']
>>> mito_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)
```

(I)-1(h)1(om”t)1(w)-454(vd)27(an)t254(9(um(t)-bTJ0aer0sta485[(If)-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("ls_orchid.gbk"), "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 check there are no extra records, and raise an exception if there are.

```
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("ls_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:

```
O23729
CHS3_BROFI
RecName: Full=Chalcone synthase 3; EC=2.3.1.74; AltName: Full=Naringenin-chalcone synthase 3;
Seq('MAPAMEEIRQAQRAEGPAAVLAIGTSTPPNALYQADYPDYYFRITKSEHLTELK...GAE', ProteinAlphabet())
Length 394
['Acyltransferase', 'Flavonoid biosynthesis', 'Transferase']
```

5.3.1 Specifying the dictionary keys

```
from Bio import SeqIO
from Bio.SeqUtils.CheckSum import seguid
for record in SeqIO.parse(open("ls_orchid.gbk"), "genbank") :
    print record.id, seguid(record.seq)
```

This should give:

```
Z78533.1 JUEoWn6DPhgZ9nAyowsgtoD9TTo
Z78532.1 MN/s0q9zDoCVEEc+k/IFwCNF2pY
...
Z78439Z78439Z75AbfaShya/4yyAj7IbMqgNkxdxQ
```

Now, recall the `Bio.SeqIO.to_dict()` function's `key_function` argument expects a function which turns a `SeqRecord` into a string. We can't use the `seguid()` function directly because it expects to be given a `Seq` object (or a string). However, we can use Python's `lambda`

5.4.2 Converting a file of sequences to their reverse complements

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:

```
# STOBkOLM#
1
1
```

```
          C#
          C#
```

```
   1
-HHH1HHHHHHHHHHHH--HHHHHHHHH--HHHHHHHHHHHHHHHHHHHHHH----e
```

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

```
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA
>Q9T0Q9_BPFD/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
>COATB_BPIF1/22-73
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA
```

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

```
Alpha     AAACAA
Beta      AAACCC
Gamma     ACCCAA
Delta     CCCACC
Epsilon   CCCAAA
    5     6
Alpha     AAAAAC
Beta      AAACCC
Gamma     AACAAC
Delta     CCCCCA
Epsilon   CCCAAC
...
    5     6
Alpha     AAAACC
Beta      ACCCCC
Gamma     AAAACC
Delta     CCCCAA
Epsilon   CAAACC
```

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

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

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

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

Note that this optional `seq_count` argument assumes each alignment in the file has the same number of sequences. Hypothetically you may come across stranger situations, for example a FASTA file containing several alignments each with a different number of sequences – although I would love to hear of a real world example of this. Assuming you cannot get the data in a nicer file format, there is no straight forward way to deal with this using `Bio.AlignIO`. In this case, you could consider [illegible]

```
Zeta        GTCAGCTAG
 3 13
Eta         ACTAGTACAG CTG
Theta       ACTAGTACAG CT-
Iota        -CTACTACAG GTG
```

```
from Bio import AlignIO
alignment = AlignIO.read(open("PF05371_seed.sth"), "stockholm")
name_mapping = {}
for i, record in enumerate(alignment) :
    name_mapping[i] = record.id
    record.id = "seq%i" % i
print name_mapping

handle = open("PF05371_seed.phy","w")
AlignIO.write([alignment], handle, "phylip")
handle.close()
```

Before trying to use ClustalW from within Python, you should first try running the ClustalW tool yourself by hand at the command line, to familiarise yourself the other options. You'll find the Biopython wrapper is very faithful to the actual command line API:

```
>>> return_code = subprocess.call(str(cline), shell=(sys.platform!="win32"))
...
>>> print return_code
0
```

```
filesm Enemnpdaedn-3095(on)3095scmoun sa nd
nnorkn28(ed1.)-600(Y)83(oun)3895(h)saWdnduan-3895knd
form(omm)-1(an)1(d)333((lin)1(e)334((on)-8(old)333(outn)1putn1(:e)]TETG100167.01875914.9210cm0g0G0g0G10016

  returncode = subprocess.call(str(cline),
...-162755st(outn)-525(=)-525openros.devnuall),
...-162755st(errn)-525(=)-525openros.devnuall),
...-162755sshell=(sys.platform!=”win32”))
  print returncode
0
```

```
seCl(usalWl)-895shoulndcealnterminaldhnbnat(e)-87((c)-1arn)1(e)]TJ-14.944-11.285[(be)3765oun)1tpunthen
uemdpne Imsecasethe
 Yourshouldnablne(on)333(w)-7(orkd)333(oun)1tendwr(on)333(rea(d)333(inn)333(thn)1(e)334(alignn)1me
```

```
 2 gi|6273291|gb|AF191665.1|AF191665     1        []  0.00418    0.01188       -       -
 3                                 -      1    [4, 5]  0.00083    0.00853       -       -
 4 gi|6273290|gb|AF191664.1|AF191664     3        []  0.00189    0.01042       -       -
 5 gi|6273289|gb|AF191663.1|AF191663     3        []  0.00145    0.00998       -       -
 6                                 -      0    [7, 8]  0.00014    0.00014       -       -
 7 gi|6273287|gb|AF191661.1|AF191661     6        []  0.00489    0.00503       -       -
 8 gi|6273286|gb|AF191660.1|AF191660     6        []  0.00295    0.00309       -       -
 9                                 -      0  [10, 11]  0.00125    0.00125       -       -
10 gi|6273285|gb|AF191659.1|AF191659     9        []  0.00094    0.00219       -       -
11 gi|6273284|gb|AF191658.1|AF191658     9        []  0.00018    0.00143       -       -

Root:  0
None
```

The `Bio.AlignIO` module should be able to read these alignments using `format="clustal"`.

MUSCLE can also output in GCG MSF format (using the `msf` argument), but Biopython can't currently parse that, or using HTML which would give a human readable web page (not suitable for parsing).

You can also set the other optional parameters, for example the maximum number of iterations. See the built in help for details.

You would then run MUSCLE command line string as described above for ClustalW, and parse the output using `Bio.AlignIO` to get an alignment object.

6.3.3 MUSCLE using stdout

Using a MUSCLE command line as in the examples above will write the alignment to a file. This means there will be no important information written to the standard out (stdout) or standard error (stderr) handles.

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

>HBA_HUMAN
MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHG

Next we want to use Python to run this command for us. As explained above, for full control, we recommend you use the built in Python subprocess module. For simple usage the subprocess.call() wrapper function usually su□ces:

```
>>> import subprocess
>>> return_code = subprocess.call(str(cline), shell=(sys.platform!="win32"))
Needleman-Wunsch global alignment of two sequences
>>> print return_code
0
```

```
>>> 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
```

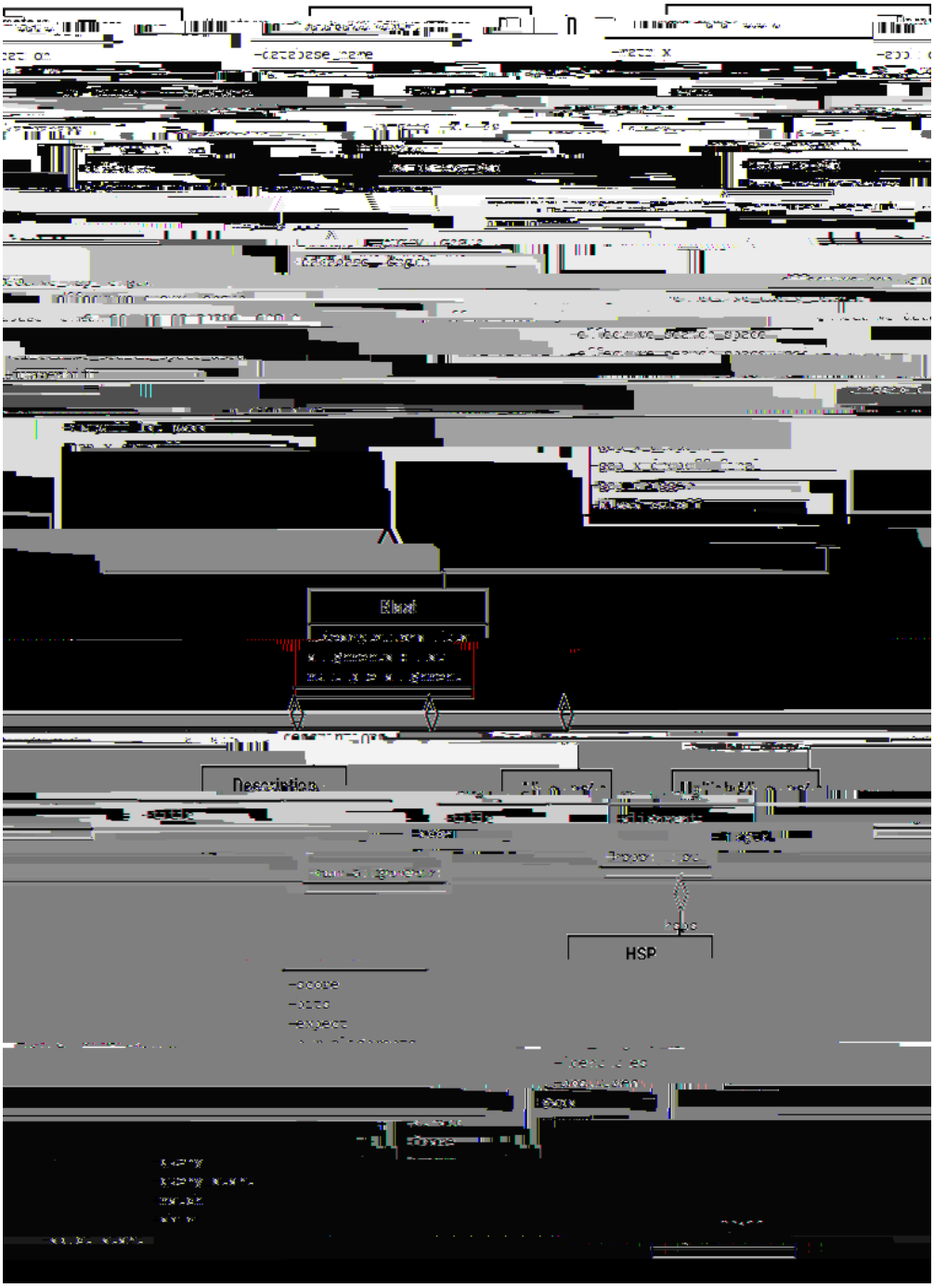
7.3 Saving BLAST output

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.

```
length: 783
e value: 0.034
tacttgttgatattggatcgaacaaactggagaaccaacatgctcacgtcacttttagtcccttacatattcctc...
||||||||| | |||||||||||  || ||||  || || ||||||||  ||||||  |  | ||||||||  |||  ||...
tacttgttggtgttggatcgaaccaattggaagacgaatatgctcacatcacttctcattccttacatcttcttc...
```

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

An important consideration for extracting information from a BLAST report is the type of objects that the information is stored in. In Biopython, the parsers return `Record` objects, either `Blast` or `PSIBlast`


Blast
Description
HSP

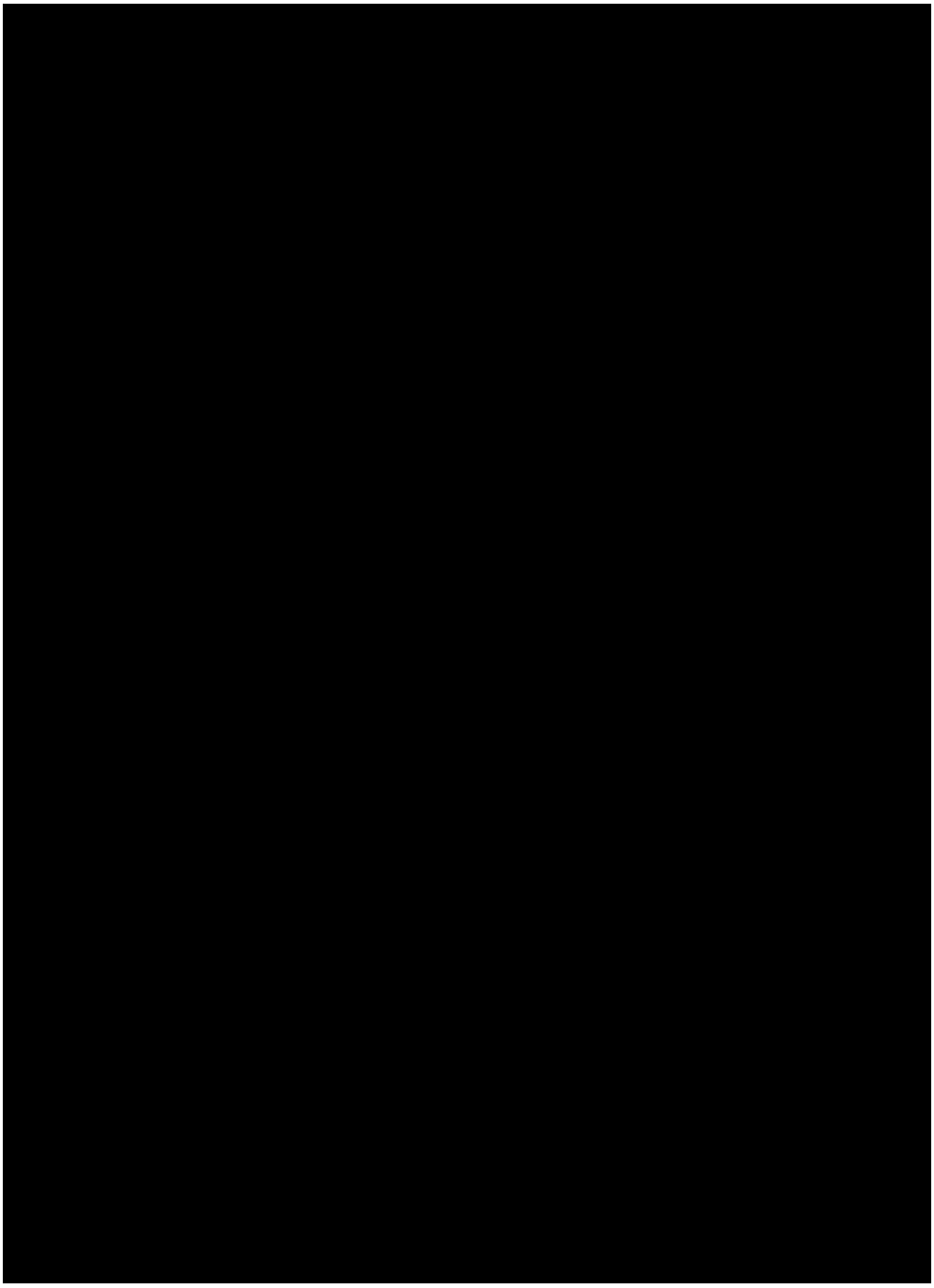


Figure 7.2: Class diagram for the PSIBlast Record class.

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

– `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 o□ending record to the specified `error_handle`

Chapter 8

Accessing NCBI's Entrez databases

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

(e)-042mancal1(lyo)-042eEnheoeoieo(o)-042ynan(e)305(CB)-1ir

```
<DbList>
        <DbName>pubmed</DbName>
        <DbName>protein</DbName>
        <DbName>nucleotide</DbName>
        <DbName>nuccore</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>domains</DbName>
        <DbName>gene</DbName>
        <DbName>genomeprj</DbName>
        <DbName>gensat</DbName>
        <DbName>geo</DbName>
        <DbName>gds</DbName>
        <DbName>homologene</DbName>
        <DbName>journals</DbName>
        <DbName>mesh</DbName>
        <DbName>ncbisearch</DbName>
        <DbName>nlmcatalog</DbName>
        <DbName>omia</DbName>
        <DbName>omim</DbName>
        <DbName>pmc</DbName>
        <DbName>popset</DbName>
        <DbName>probe</DbName>
        <DbName>proteinclusters</DbName>
        <DbName>pcassay</DbName>
        <DbName>pccompound</DbName>
        <DbName>pcsubstance</DbName>
        <DbName>snp</DbName>
        <DbName>taxonomy</DbName>
        <DbName>toolkit</DbName>
        <DbName>unigene</DbName>
        <DbName>unists</DbName>
</DbList>
</eInfoResult>
```

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

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

Now `record` is a dictionary with exactly one key:

```
>>> record.keys()
[u'DbList']
```

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

```
>>> record["DbList"]
['pubmed', 'protein', 'nucleotide', 'nuccore', 'nucgss', 'nucest',
 'structure', 'genome', 'books', 'cancerchromosomes', 'cdd', 'gap',
 'domains', 'gene', 'genomeprj', 'gensat', 'geo', 'gds', 'homologene',
 'journals', 'mesh', 'ncbisearch', 'nlmcatalog', 'omia', 'omim', 'pmc',
```

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

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

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

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

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"     # Always tell NCBI who you are
>>> handle = Entrez.espell(term="biopythooon")
>>> record = Entrez.read(handle)
>>> record["Query"]
'biopythooon'
>>> record["CorrectedQuery"]
'biopython'
```

See the ESpell help page for more information.

```
10 NAT2
11 AACP
12 SERPINA3
13 AADAC
14 AAMP
15 AANAT
16 AARS
17 AAVS1
...
```

8.11 Specialized parsers

8.11.3 Parsing UniGene records

The EXPRESS

not use the WebEnv history feature. You should use this for any non-trivial sea5ea a1(dal)-32da1(oan)2wsea1(loa(ld)-32war

Great - eleven articles. But why hasn't the Biopython application note been found (PubMed ID

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

Chapter 9

Swiss-Prot and ExPASy

9.1 Parsing Swiss-Prot files

Swiss-Prot (

```
>>> 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
'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
...
>>> print n
2073
```

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

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

```
CC   -!- Also hydrolyzes diacylglycerol.
PR   PROSITE; PDOC00110;
DR   P11151, LIPL_BOVIN ;  P11153, LIPL_CAVPO ;  P11602, LIPL_CHICK ;
DR   P55031, LIPL_FELCA ;  P06858, LIPL_HUMAN ;  P11152, LIPL_MOUSE ;
DR   O46647, LIPL_MUSVI ;  P49060, LIPL_PAPAN ;  P49923, LIPL_PIG   ;
DR   Q06000, LIPL_RAT   ;  Q29524, LIPL_SHEEP ;
//
```

In this example, the first line shows the EC (Enzyme Commission) number of lipoprotein lipase (second line). Alternative names of lipoprotein lipase are "clearing factor lipase", "diacylglycerol lipase", and

9.5 Accessing the ExPASy server

```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prosite_entry('PS00001')
>>> html = handle.read()
>>> output = open("myprositerecord.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
>>> 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

```
        ('Other1', [(1, 1),  (4, 3), (200, 200)],
    ]
]
```

11.2 Coalescent simulation

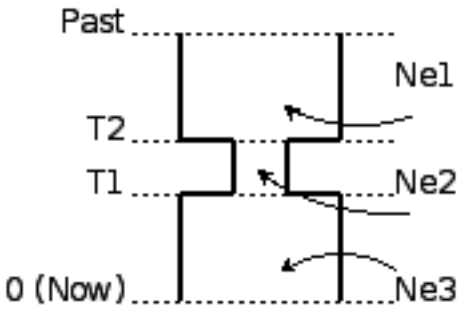


Figure 11.1: A bottleneck

11.2.1.2 Chromosome structure

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

The logistic regression model gives us appropriate values for the parameters β_0, β_1, β_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,
          0,
          0,
          0,
          0,
          0,
          0,
          0]
>>> model = LogisticRegression.train(xs, ys)
```

```
Iteration: 2 Log-likelihood function: -5.76877209868
```

0; corresponding to class OP and class NOP, respectively. For example1(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*

```
...
>>> 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,-11.9OP:2352]
```

Chapter 13

13.1.4 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.

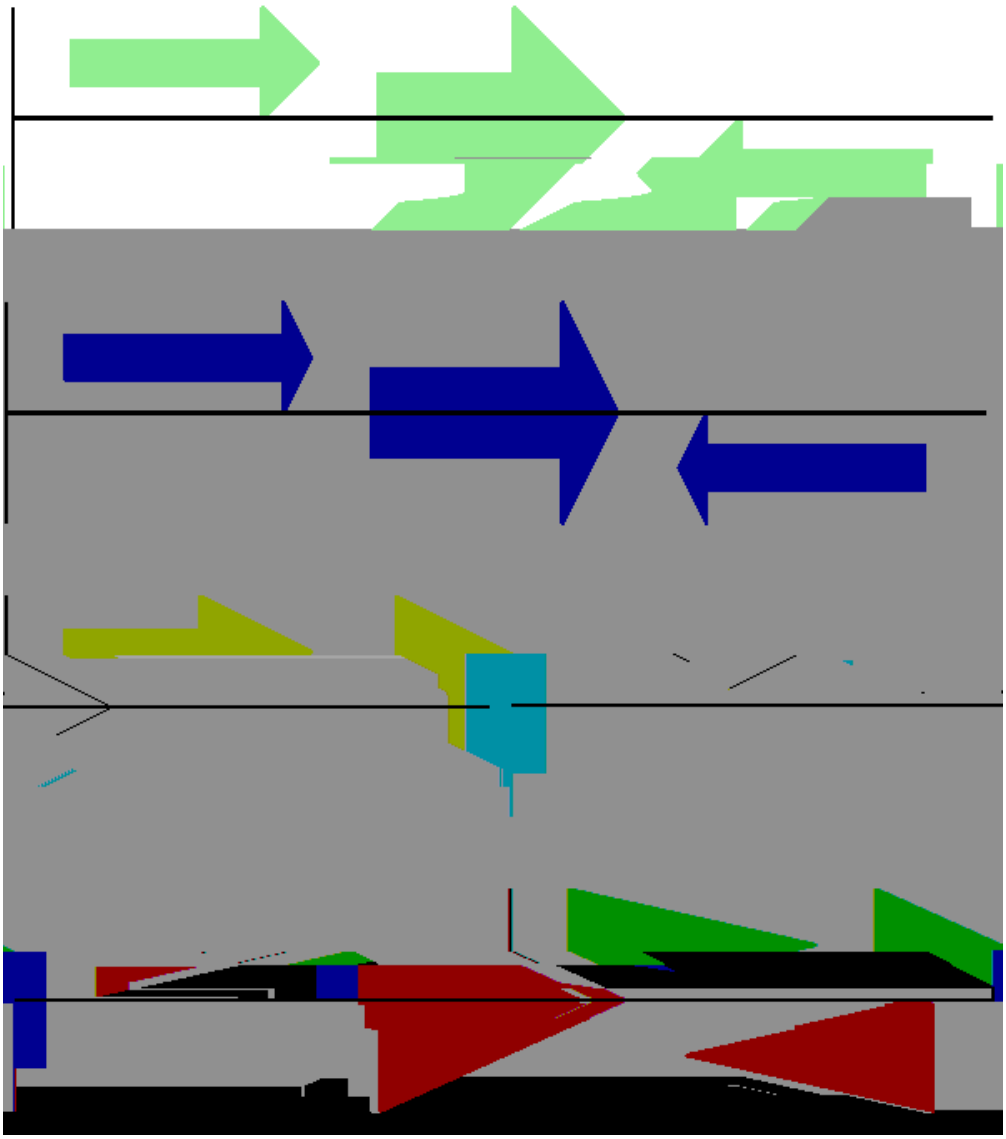
```
from reportlab.lib import colors
from reportlab.lib.units import cm
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO
record = SeqIO.read(open("NC_005816.gb"), "genbank")

#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)
```

```
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



```
    #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

14.1.2 Translating a FASTA file of CDS entries

Note that using `Bio.SeqIO.convert()` like this is *much* faster than combining `Bio.SeqIO.parse()` and `Bio.SeqIO.write()`

```
                aa_end = trans.find("*", aa_start)
                if aa_end == -1 :
                    aa_end = trans_len
                if aa_end-aa_start >= min_protein_length :
                    if strand == 1 :
                        start = frame+aa_start*3
                        end = min(seq_len,frame+aa_end*3+3)
                    else :
                        start = seq_len-frame-aa_end*3-3
                        end = seq_len-frame-aa_start*3
                    answer.append((start, end, strand,
                                   trans[aa_start:aa_end]))
                aa_start = aa_end+1
    answer.sort()
    return answer

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

And the output:

```
NQIQGVICSPDSGEFMVTFETVMEIKILHK...GVA - length 355, strand 1, 41:1109
WDVEn_prDLHHPFHLTFSLCPEGATQSGR...VKR - length 111, strand -1, 491:827
KSGELRQTPPASSTLHLRLILQRSGVMMEL...NPE - length 285, strand 1, 1030:1888
RALTGLSAPGIRSQTSCDRLRELRYVPVSL...PLQ - length 119, strand -1, 2830:3190
RRKEHVSKKRRPQKRPRRRRFFHRLRPPDE...PTR - length 128, strand 1, 3470:3857
GLNCSFFSICNWKFIDYINRLFQIIYLCKN...YYH - length 176, strand 1, 4249:4780
RGIFMSDTMVVNGSGGVPAFLFSGSTLSSY...LLK - length 361, strand -1, 4814:5900
VKKILYIKALFLCTVIKLRRFIFSVNNMKF...DLP - length 165, strand 1, 5923:6421
LSHn_pDFTDQMAQVGLCQCVNVFLDE_pr...KAA - length 107, strand -1, 5974:6298
GCLMKKSSIVATIITILSGSANAASSQLIP...YRF - length 315, strand 1, 6654:7602
IYSTSEHTGEQVMRTLDE_IASRSPESQTR...FHV - length 111, strand -1, 7788:8124
WGKLQVIGLSMWMVLFSQRFDDWLNEQEDA...ESK - length 125, strand -1, 8087:8465
TGKQNSCQMSAIWQLRQNTATKTRQNRARI...AIK - length 100, strand 1, 8741:9044
QGSGYAFPHASILSGIAMSHFYFLDLHAVK...CSD - length 114, strand -1, 9264:9609
```

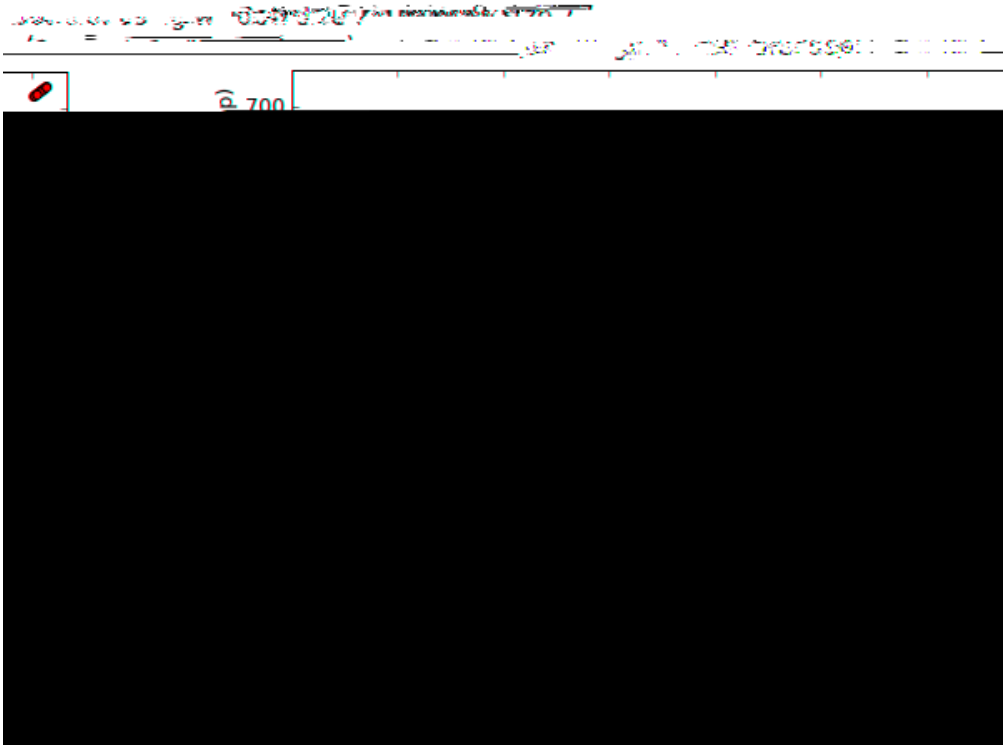
If you comment out the sort statement, then the protein sequences will be shown in the same order as before, so you can check this is doing the same thing. Here we have sorted them by location to make it easier to compare to the actual annotation in the GenBank file (as visualised in Section 13.1.8).

tutorial


94 orchid sequences

To start o, we'll need two sequences. For the sake of argument, we'll just take the first two from our

```
        except KeyError :
            section_dict[section] = [i]
#Now find any sub-sequences found in both sequences
```



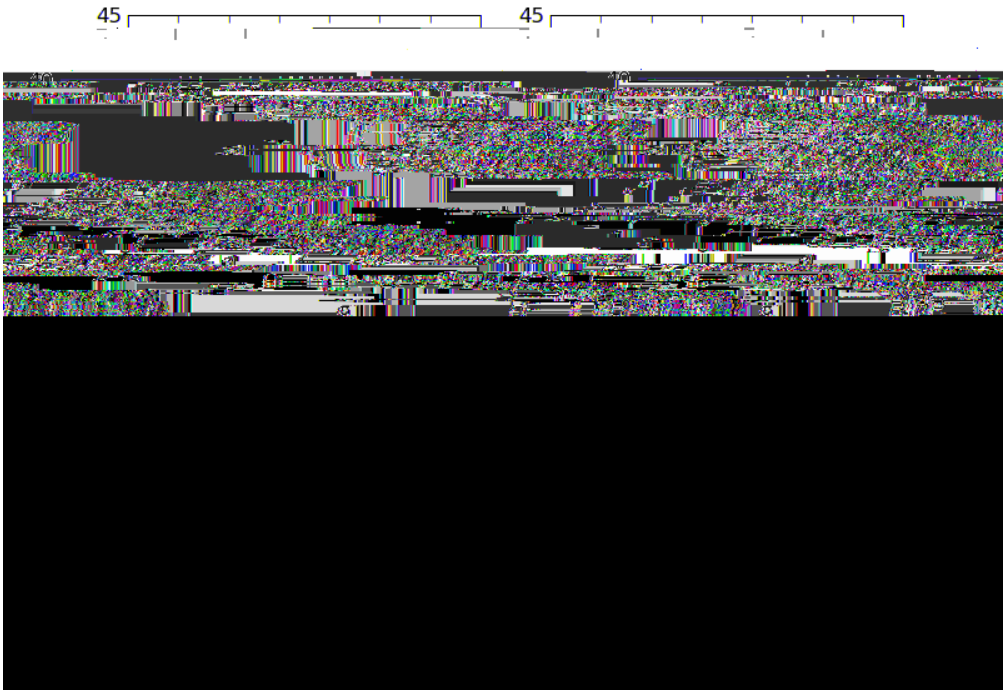


Figure 14.5: Quality plot for some paired end reads.

Chapter 6

2.

14.4 Substitution Matrices

Substitution matrices are an extremely important part of everyday bioinformatics work. They provide the scoring terms for classifying how likely two different residues are to substitute for each other. This is essential in doing sequence comparisons. The book "Biological Sequence Analysis" by Durbin et al. provides a really nice introduction to Substitution Matrices and their uses. Some famous substitution matrices are the PAM and BLOSUM series of matrices.

Biopython provides a ton of common substitution matrices, and also provides functionality for creating your own substitution matrices.

- Name
- Dates
- Type
- Parent
- Process
- Function
- Component
- Signatures
- Abstract
- Examples

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.
-

```
from Bio import Biospam

print "2 + 3 =", Biospam.addition(2, 3)
print "9 - 1 =", Biospam.addition(9, -1)
print "2 * 3 =", Biospam.multiplication(2, 3)
print "9 * (- 1) =", Biospam.multiplication(9, -1)
```

We generate the corresponding output with `python run_tests.py -g test_Biospam.py`, and check the output:

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

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


if __name__ == "__main__" :
```

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

def test_addition2(self):
    """A second addition test"""
    result = Biospam.addition(9, -1-11.9552Td[(selfaaitionelf.assert41.842850result4(9cl5(-)]TJ
```

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

Chapter 18

Appendix: Useful stu about Python

If you haven't spent a lot of time programming in Python, many questions and problems that come up in