

# Biopython Tutorial and Cookbook

Jeff Chang, Brad Chapman, Iddo Friedberg, Thomas Hamelryck, Michiel de Hoon, Peter Cock

# Contents

<b>1 Introduction</b>	<b>5</b>
1.1 What is Biopython? . . . . .	5

#### 4.4 Writing Sequence Files . . . . .

8.4	Substitution Matrices	65
8.4.1	Using common substitution matrices	65
8.4.2	Creating your own substitution matrix from an alignment	65
8.5	BioRegistry { automatically finding sequence sources	67
8.5.1	Finding resources using a configuration file	67

<b>11 Appendix: Useful stuff about Python</b>	<b>107</b>
11.1 What the heck is a handle?	107
11.1.1 Creating a handle from a string	107

# 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. The web site <http://www.biopython.org>

Interfaces to common bioinformatics programs such as:

- { Standalone Blast from NCBI
- { Clustalw alignment program.

A standard sequence class that deals with sequences, ids on sequences, and sequence features.

Tools for performing common operations on sequences, such as translation, transcription and weight calculations.

Code to perform classification of data using k Nearest Neighbors, Naive Bayes or Support Vector Machines.

Code for dealing with alignments, including a standard way to create and deal with substitution matrices.

Code making it easy to split up parallelizable tasks into separate processes.

GUI-based programs to do basic sequence manipulations, translations, BLASTing, etc.

Extensive documentation and help with using the modules, including this file, on-line wiki documentation, the web site, and the mailing list.

Integration with other languages, including the Bioperl and Biojava projects, using the BioCorba interface standard 17

## Chapter 2

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





## 2.4 Parsing sequence le formats

### 2.4.2 Simple GenBank parsing example

Now let's load the GenBank file instead - notice that the code to do this is almost identical to the snippet used above for a FASTA file - the only difference is we changed the filename and the format string:

```
from Bio import SeqIO
handle = open("Is_orchid.gb")
for seq_record in SeqIO.parse(handle, "genbank") :
    print seq_record.id
    print repr(seq_record.seq)
    print len(seq_record.seq)
handle.close()
```

This should give:

## SCOP

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.

Here we'll show a simple example of performing a remote Entrez query. More information on the other services is available in the Cookbook, which begins on page 55.

In section 2.3 of the parsing examples, we talked about using Entrez website to search the NCBI nucleotide databases for info on *Cypripedioideae*, our friends the lady slipper orchids. Now, we'll look at how to automate that process using a python script. For Entrez searching, this is more useful for displaying results than as a tool for getting sequences. The NCBI web site is mostly set up to allow remote queries so that

Snazzy! We can fetch things and display them automatically { you could use this to quickly set up searches that you want to repeat on a daily basis and check by hand, or to set up a small CGI script to do queries and locally save the results before displaying them (as a kind of lab notebook of our search results). Hopefully whatever your task, the database connectivity code will make things lots easier for you!

## 2.6 What to do next

Now that you've made it this far, you hopefully have a good understanding of the basics of Biopython and are ready to start using it 4(thin17-369(itfn1r(displaellconnef 0 -32snn550(Th)-368(r)-28(efs)-368(4hings-369(to)-368(sd)-368(2tkeatk-3356a thecokocokhyvdeot-3356aoado

## Chapter 3

# Sequence objects

```
Seq('AGTACACTGGT', Alphabet())
>>> my_seq.alphabet
Alphabet()
```

```
>>> S>> S>t
>>> Alphabet
>>> S>> show76-333(y27G9ocacts(y27G9eleme-)28(ts(y27G9of(y27G93cen)-76-n)-33ohab567G9in(y27G93cen)-76-namab
>>> my_s>> Seq('AGTACACTGGT',
>>> my_s51
Seq('AGTACACTGGT',
>>> my_seq.alphabet
Suenceset actchis51
```









```
>>> my_seq = Seq("GCCATTGTAATGGGCCGCTGAAAGGGTCCCCGA", IUPAC.unambiguous_dna)
>>> standard_translator.translate(my_seq)
Seq('AIVMGR*KGAR', IUPACProtein())
>>> mito_translator.translate(my_seq)
Seq('AIVMGRWKGAR', IUPACProtein())
```

Notice that the default translation will just go ahead and proceed blindly through a stop codon. If you are aware that `yt stostAotttfrau ttsts tsetup3o28(tunat)`-

## Chapter 4

# Sequence Input/Output

In this chapter we'll discuss in more detail the Bio.SeqIO module, which was briefly introduced in Chapter 2. This is a relatively new interface, added in Biopython 1.43, which aims to provide a simple interface for working with assorted sequence file formats in a uniform way.

The "catch" is that you have to work with SeqRecord

The above example is repeated from the introduction in Section 2.4, and will load the orchid DNA sequences in the FASTA format `ls_orchid.fasta` `load format` `le ikn`



ID: Z78533.1

Name: Z78533

Description: C. irapeanum 5.8S rRNA gene and ITS1 and ITS2 DNA.

/source=Cypripedium irapeanum

/taxonomy=['Eukaryota', 'Viridiplantae', 'Streptophyta', ..., 'Cypripedium']

/keywords=['5.8S ribosomal RNA', '5.8S rRNA gene', 'internal transcribed spacer', 'ITS1', 'ITS2']

/references=[...]

from Bio import SeqIO



Suppose you only want to download a single record? When you expect the handle to contain one and only one record, in Biopython 1.45 or later you can use the `Bio.SeqIO.read()` function:

```
from Bio import GenBank
from Bio import SeqIO
handle = GenBank.download_many(["6273291"])
seq_record = SeqIO.read(handle, "genbank")
handle.close()
```

#### 4.2.2 Parsing SwissProt sequences from the net

We can access a single SeqRecord



```
rec2 = SeqRecord(Seq("YPDYYFRI TNREHKAELKEKFQRMCDKSMI KKRYMYLTEEI LKENPSMCEYMAPSLDARQ" \
    + "DMVVVEI PKLGKEAAVKAI KEWGQ", generic_protein),
    id="gi |13919613|gb|AAK33142.1|",
    description="chalcone synthase [Fragaria vesca subsp. bracteata]")

rec3 = SeqRecord(Seq("MVTVEEFRAQCAEGPATVMAI GTATPSNCVDQSTYPDYYFRI TNSEHKVELKEKFKRMC" \
    + "EKSMI KKRYMHLTEEI LKENPNI CAYMAPSLDARQDI VVVEVPKLGKEAAQKAI KEWGQP" \
    + "KSKI THLVFCTTSGVDMPGCDYQLTKLLGLRPSVKRFMMYQQGCFAGGTVLRMAKDLAEN" \
```



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

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.seq) < 700]
```

## Chapter 5

# 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? Heck, if I was writing the code to do that it would probably take about a day and a half to complete, and the results still wouldn't be as good. But, of course, this section isn't about how cool BLAST is, since we already know that. It is about the problem with BLAST { it can be really difficult to deal with the volume of data generated by large runs, and to automate BLAST runs in general.





First, we need to get the info in the FASTA file. The easiest way to do this is to use the `Bio.SeqIO` module. In this example, we'll use `Bio.SeqIO.parse` to parse the FASTA file and store the first FASTA record in the file in a `SeqRecord` object (section [2.4.1](#) explains `Bio.SeqIO.parse` in more detail).

```
>>> from Bio import SeqIO
>>> record = SeqIO.parse("example.fasta", "fasta").next()
```



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



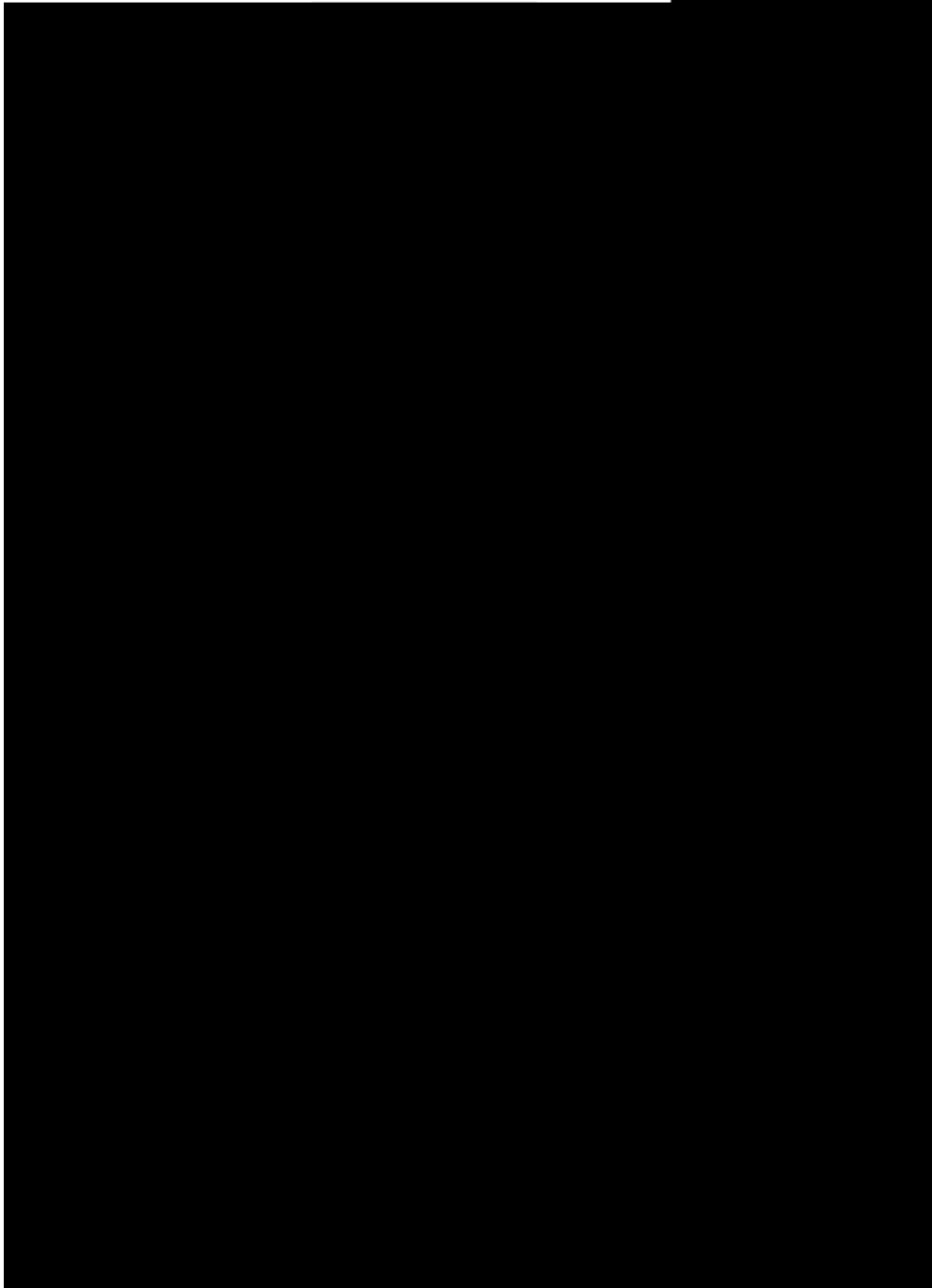


Figure 5.1: Class diagram for the Blast Record class representing all of the info in a BLAST report





Then we will assume we have a handle to a fa1(ncw)27ho lastowhicv w'illcailI]TJ/F328 9.9626 Tf382.294g 0 Td [result



```
>>> from Bio.Blast import NCBIStandalone
>>> error_handle = open(error_file, "w")
```









Again, we could use EFetch to obtain more information for each of these journal IDs.  
ESearch has many useful options | see the [ESearch help page](#) for more information.

EPost posts a list of UIs for use in subsequent search strategies; see -250(p)-28(osts)-250(a)-2EP8(h)-3olp page in a1 Tdavailb

6.Summary:ion00(Retrievingfor750(immar0(sfor750from[-3750priy:io84io84io84io84io84io84ttt44 -32.9(U94(r(6.Sum33(mevinU9

```
>>> handle = Entrez.efetch(db="nucleotide", id="57240072", rettype="genbank")
>>> print handle.read()
```

LOCUS AY851612 892 bp DNA linear PLN 10-APR-2007

DEFINITION Opuntia subulata rpl16 gene, intron; chloroplast.

ACCESSION AY851612

VERSION AY851612.1 GI:57240072

KEYWORDS .

SOURCE chloroplast Austrocylindropuntia subulata

Spermatophyta; Magnoliophyta; eudicotyledons; core eudicotyledons;

Caryophyllales; Cactaceae; Opuntioideae; Austrocylindropuntia.

REFERENCE 1 (bases 1 to 892)

TITLE Molecular Phylogenetics of the Leafy Cactus Genus Pereskia

REFERENCE 2 (bases 1 to 892)

TITLE Direct Submission

JOURNAL Genbank  
ORGANISM Austrocylindropuntia subulata  
LOCATION Botanical Garden, 1201 North Galvin  
/organism="Austrocylindropuntia subulata"  
/mol\_type="genomic DNA"

options. The available formats depend on which database you are downloading from.



```
<MenuName>PMC</MenuName>  
<Count>359</Count>  
<Status>Ok</Status>  
</ResultItem>
```

## Chapter 7

# Swiss-Prot, Prosite, Prodoc, and ExPASy

### 7.1 Bio.SwissProt: Parsing Swiss-Prot records

Swiss-Prot (<http://www.expasy.org/sprot>) is a hand-curated database of protein sequences. In Section [4.2.2](#)

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



```
>>> from Bio 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.

### 7.3 Bio.Prosite.Prodoc: Parsing Prodoc records

In the Prosite example above, the `record.pdoc` accession numbers `'PD0C00001'`, `'PD0C00004'`, `'PD0C00005'` and so on refer to Prodoc records, which contain the Prosite Documentation. The Prodoc records are available from ExPASy as individual files, and as one file `1'record.pdoc`





## Chapter 8



What we've done is create a dictionary like object `medline_dict`. To get an article we access it like `medline_dict[id_to_get]`. What this does is connect with PubMed, get the article you ask for, parse it into a record object, and return it. Very cool!

Now let's look at how to use this nice dictionary to print out some information about some ids. We just need to loop through our ids (`orchi_ids` from the previous section) and print out the information we are interested in:

```
for oid in orchi_ids[0:5]:  
    cur_record = medline_dict[oid]
```



2. `FeatureParser` { This parses the raw record in a `SeqRecord` object with all of the feature table information represented in `SeqFeatures` (see section 9.1 for more info on these objects). This is best to use if you are interested in getting things in a more standard format. If you use `Bio.SeqIO` (Chapter 4) to read a GenBank file, it will call this `FeatureParser` for you.

Depending on the type of GenBank files you are interested in, they will either contain a single record, or multiple records. Each record will start with a LOCUS line, various other header lines, a list of features, and finally the sequence data, ending with a // line.

Dealing with a GenBank file containing a single record is very easy. For example, let's use a small bacterial genome,

#### 8.2.4 Making your very own GenBank database

One very cool thing that you can do is set up your own personal GenBank database and access it like a dictionary (this can be extra cool because you can also allow access to these local databases over a network

We'll need some sequences to align, such as [opuntia.fasta](#) (also available online [here](#)) which is a small



```
consensus Seq(' TATACATNAAAGNAGGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAATGAAT
... ', IUPACAmbiguousDNA())
```

You can adjust how `dumb_consensus` works by passing optional parameters:

**the threshold** This is the threshold specifying how common a particular residue has to be at a position before it is added. The default is .7.

**the ambiguous character**

```
second_seq = alignment.get_seq_by_num(1)
my_pssm = summary_align.pos_specific_score_matrix(second_seq
```



Well, now that we have an idea what information content is being calculated in Biopython, let's look at

### 8.3.6 Translating between Alignment formats

One thing that you always end up having to do is convert between different formats. Biopython does this

```
from Bio import Clustalw
from Bio.Alphabet import IUPAC
from Bio.Align import AlignInfo

# get an alignment object from a Clustalw alignment output
c_align = Clustalw.parse_file("protein.aln", IUPAC.protein)
```

Once you've got your log odds matrix, you can display it prettily using the function `print_mat`. Doing

```
>>> print db
DBRegistry, exporting 'embl', 'embl-dbfetch-cgi', 'embl-ebi-cgi',
'embl-fast', 'embl-xembl-cgi', 'interpro-ebi-cgi',
'nucl eotide-dbfetch-cgi', 'nucl eotide-genbank-cgi', 'pdb',
'pdb-ebi-cgi', 'pdb-rcsb-cgi', 'prodoc-expasy-cgi',
'prosite-expasy-cgi', 'protein-genbank-cgi', 'swissprot',
'swissprot-expasy-cgi'
>>> db.keys()
['embl-dbfetch-cgi', 'embl-fast', 'embl', 'prosite-expasy-cgi',
'swissprot-expasy-cgi', 'nucl eotide-genbank-cgi', 'pdb-ebi-cgi',
'interpro-ebi-cgi', 'embl-ebi-cgi', 'embl-xembl-cgi',
'protein-genbank-cgi', 'pdb', 'prodoc-expasy-cgi',
'nucl eotide-dbfetch-cgi', 'swissprot', 'pdb-rcsb-cgi']
```

Now, let's say we want to retrieve a swissprot record for one of our orchid chalcone synthases. First, we get the swissprot connection, then we retrieve an record of interest:

```
>>> sp = db["swissprot"]
>>> sp
<Bio.DBRegistry.DBGroup instance at 0x82fdb2c>
record_handle = sp['023729']
>>> print record_handle.read()[:200]
ID   CHS3_BROFI      STANDARD;          PRT;   394 AA.
AC   023729;
DT   15-JUL-1999 (Rel. 38, Created)
DT   15-JUL-1999 (Rel. 38, Last sequence update)
DT   15-JUL-1999 (Rel. 38, Last annotation update)
```

This retrieval method is nice for a number of reasons. First, we didn't have to worry about where exactly swissprot records were being retrieved from { we only ask for an object that will give us any swissprot record we can get. Secondly, once we get the swissprot object, we don't need to worry about how we are getting our sequence { we just ask for it by id and don't worry about the implementation details.

The default biopython database registry object can be used similarly to retrieve sequences from EMBL, prosite, PDB, interpro, GenBank and XEMBL.

### 8.5.2.2 Registering and Grouping databases

The basic registry objects are nice in that they provide basic functionality, but if you have a more advanced

```
doc = "Query a local databases",  
failure_cases = [])
```

Now that we have specified the details for connecting to the CGI script, we are ready to register this CGI script. We just need one more detail { specifying what the script returns upon failure to find a sequence. We do this using Martel regular expressions:

```
import Martel  
my_failures = [  
    (Martel.Str("Sequence not available"), "No sequence found")]
```

Now we've got everything we need, and can register the database:

```
from Bio import register_db  
register_db(name = "nucleotide-genbank-local",  
            key = "uid",  
            source = local_cgi,  
            failure = my_failures)
```

This makes the database available as before, so if we print the keys of the database, we'll see "nucleotide-

## 8.8 Going 3D: The PDB module

Biopython also allows you to explore the extensive realm of macromolecular structure. Biopython comes with a PDBParser class that produces a Structure object. The Structure object can be used to access the

Disordered atoms and residues are represented by `DisorderedAtom` and `DisorderedResidue` classes, which are both subclasses of the `DisorderedEntityWrapper` base class. They hide the complexity associated with



#### 8.8.1.1 Structure

The Structure object is at the top of the hierarchy. Its id is a user given string. The Structure contains



```
a.get_sigatm()      # std. dev. of atomic parameters
a.get_siguij()      # std. dev. of anisotropic B factor
a.get_anisou()      # anisotropic B factor
a.get_fullname()    # atom name (with spaces, e.g. ".CA.")
```

To represent the atom coordinates, siguij, anisotropic B factor and sigatm Numpy arrays are used.

### 8.8.2 Disorder



```
for residue in chain.get_list():  
    residue_id=residue.get_id()  
    hetfield=residue_id[0]  
    if hetfield[0]=="H":  
        print residue_id
```

#### 8.8.5.1.1 Duplicate residues

#### 8.8.6 Other features





### 8.9.2 Coalescent simulation

A coalescent simulation is a backward model of population genetics with relation to time. A simulation of



Figure 8.2: A bottleneck

```
from Bio.PopGen.SimCoal.Template import generate_simcoal_from_template  
generate_simcoal_from_template('simple',  
    [(1, [('SNP', [24, 0.0005, 0.0])]),  
    [('sample_size', [30]),  
    ('pop_size', [100])])])
```

how to implement chromosome structures using the Biopython interface, not the underlying SIMCOAL2 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`):

```
[(1, [('SNP', [24, 0.0005, 0.0]]))]
```

This is actually the chromosome structure used in the above examples.

fromfromfromgenerate\_simcoal\_

2. Compute average  $F_{st}$ . This is done by `datacal` inside `FDist`.
3. Simulate "neutral" markers based on the average  $F_{st}$  and expected number of total populations. This is the core operation, done by `fdist` inside `FDist`.
4. Calculate the confidence interval, based on the desired confidence boundaries (typically 95% or 99%). This is done by `cplot` and is mainly used to plot the interval.
5. Assess each marker status against the simulation "neutral" confidence interval. Done by `pv`. This is used to detect the outlier status of each marker against the simulation.

We will now discuss each step with illustrating example code (for this example to work `FDist` binaries have to be on the executable `PATH`).

The `FDist` data format is application specific and is not used at all by other applications, as such you will probably have to convert your data for use with `FDist`. `Biopython` can help you do this. Here is an example converting from `GenePop` format to `FDist` format (along with imports that will be needed on examples further below):

**sample\_size** Average number of individuals sampled on each population.

**mut** Mutation model: 0 - Infinite alleles; 1 - Stepwise mutations

**num\_sims** Number of simulations to perform. Typically a number around 40000 will be OK, but if you get a confidence interval that looks sharp (this can be detected when plotting the confidence interval computed below) the value can be increased (a suggestion would be steps of 10000 simulations).

The confusion in wording between number of samples and sample size stems from the original application. A file named out.dat will be created with the simulated heterozygosities and  $F_{st}$ s, it will have as many lines as the number of simulations requested.

Note that fdist returns the average  $F_{st}$  that it was *capable* of simulating, for more details about this issue please read below the paragraph on approximating the desired average  $F_{st}$ .

The next (optional) step is to calculate the confidence interval:

```
cpl_interval = ctrl.run_cplot(ci=0.99)
```



# Chapter 9

## Advanced

### 9.1 The SeqRecord and SeqFeature classes

You read all about the basic Biopython sequence class in Chapter 3, which described how to do many useful things with just the sequence. However, many times sequences have important additional properties associated with them { as you will have seen with the SeqRecord object in Chapter 4. This section described



Additionally, you can also pass the id, name and description to the initialization function, but if not they

strand



If you don't want to deal with fuzzy p sAons4(p)cuand4(p)cujus54(w)27(an)28(t)-454(to)-(t)-4umtt858(ou)-454(don'ju



sequence	MAKLEI TLKRSVI GRPEDQRVTVRTLGLKKTNQTVVHEDNAAI RGMINKVSHLVSVEQ
end_sequence	
begin_sequence	
title	>gi   132679   sp   P19946   RL15_BACSU 50S RIBOSOMAL PROTEIN L15



hsps_prelim_gapped	gapped (not tblastx) and not blastn
hsps_prelim_gap_attempted	gapped (not tblastx) and not blastn
hsps_gapped	gapped (not tblastx) and not blastn
query_length	
database_length	
effective_hsp_length	
effective_query_length	
effective_database_length	
effective_search_space	
effective_search_space_used	
frameshift	blastx or tblastn or tblastx
threshold	
window_size	
dropoff_1st_pass	
gap_x_dropoff	
gap_x_dropoff_final	gapped (not tblastx) and not blastn
gap_trigger	
blast_cutoff	

### 9.3.7 Enzyme

The Enzyme.py module works with the enzyme.dat file included with the Enzyme distribution. The Enzyme Scanner produces the following events:

```
record
  identification
  description
  alternate_name
  catalytic_activity
  cofactor
  comment
  disease
  protein_reference
  databank_reference
  terminator
```

### 9.3.8 KEGG

#### 9.3.8.1 Bio.KEGG.Enzyme

The Bio.KEGG.Enzyme module works with the 'enzyme' file from the Ligand database, which can be obtained from the KEGG project. (<http://www.genome.ad.jp/kegg>).

The Bio.KEGG.Enzyme.Record contains all the information stored in a KEGG/Enzyme record. Its

tnformat6eov KEG7 r,(KEG7)istat6edeedat6eo-495(b)-t6edat6eo-333(th7 Td [(record)]TJ 0 -11.955 Td33(fro3)-535(w7ic)2ro



product  
inhibitor  
cofactor  
effector  
comment  
pathway\_db  
pathway\_id  
pathway\_desc  
organism  
gene\_id  
disease\_db  
disease\_id  
disease\_desc  
motif\_db  
motif\_id  
motif  
structure\_db  
structure\_id  
dblinks\_db  
dblinks\_id  
record\_end

#### 9.3.8.2 Bio.KEGG.Compound

### 9.3.10 Medline



sequence\_header  
sequence\_data  
terminator

input\_file\_name  
num\_int\_metabolites  
num\_reactions  
metabolite\_line  
unbalancedites

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

```
    i, data: can be either a dictionary or another SeqMat instance.
```

```
    d[(data)]L/F89.41.842F20.92101d[t.b55r]xgener.budi n1d5(Oneary)83(fr)-33(-33(b)-27follo)28(wing
```

```
    ii.data n1d1 frSeqMprovi dednary or4(ei the2G/n)-834tr fromr01 instance.
```

```
    ii.
```

('A', 'C'): 10, ('C', 'A'): 12

as order doesn't matter, user can already provide only one entry:

('A', 'C'): 22

A SeqMat instance may be initialized with either a full (rst method of counting: 10, 12) or half

- iii. factor: factor used to multiply the log-odds values. Each entry is generated by  $\log(\text{LOM}[\text{key}]) * \text{factor}$   
And rounded to the round\_digi t place after the decimal point, if required.

#### 4. Example of use

As most people would want to generate a log-odds matrix, with minimum hassle, SubsMat provides one function which does it all:

```
make_log_odds_matrix(acc_rep_mat, exp_freq_table=None, logbase=10,  
                    factor=10.0, round_digi t=0):
```

- (a) acc\_rep\_mat: user provided accepted replacements matrix
- (b) exp\_freq\_table



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)

## Chapter 10

Where to go from here { contributing  
to Biopython

Macintosh

## Chapter 11

# 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([>>>]>9aI0
>>> my_info_handle = cStri2Fst_>          >>>5(my_info_handle)-525(=)-525(cStrTd[(A)-52i2Fst_>)-50
  with mult5.23.91525(=)-525(cStrsecond_>)-5050[(>>>)>9angI0.StringI0.read>5(my_info_handle)6525(=)-525(

```