

# Biopython Tutorial and Cookbook

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

Last Update – September 2008

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	What is Biopython? . . . . .	5
1.1.1	What can I find in the Biopython package	

4.3.1	Specifying the dictionary keys . . . . .	30
4.3.2	Indexing a dictionary using the SEGUID checksum . . . . .	

<b>8</b>	<b>Swiss-Prot, Prosite, Prodoc, and ExPASy</b>	<b>77</b>
8.1	Bio.SwissProt: Parsing Swiss-Prot files . . . . .	77
8.1.1	Parsing Swiss-Prot records . . . . .	77
8.1.2	Parsing the Swiss-Prot keyword and category list . . . . .	79
8.2	Bio.Prosite: Parsing Prosite records . . . . .	80



# Chapter 1



4. *Why doesn't `str(...)` give me the full sequence of a Seq object?*





followed by what you would type 4m2B

## 2.4 Parsing sequence file formats

A large part of much bioinformatics work involves dealing with the many types of file formats designed to hold biological data. These files are loaded with interesting biological data, and a special challenge is parsing these files into a format so that you can manipulate them with some kind of software.





## Chapter 3

# Sequence objects

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

Note that using the Bio.SeqUtils.GC() function should automatically cope with mixed case sequences and the ambiguous nucleotide S which means G or C.

Also note that just like a normal python string, the Seq object is in some ways "read-only". If you need to edit your sequence, for example simulating a point mutation (look at the Seq object methods), you can use the Seq object's mutate method. For example, to simulate a point mutation at position 100, you can use the following code:



Because calling `str()` on a `Seq` object returns the full sequence as a string, you often don't actually have subject (1420753101 (xamp)1360 (u) en (u) 2048160 (31 (r) der, 3[066) -1(xamp)1060 (m1 (xaust



```
>>> coding_dna
Seq(' ATGGCCATTGTAATGGGCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> template_dna = coding_dna.reverse_complement()
>>> template_dna
Seq(' CTATCGGGCACCCCTTTCAGCGGCCCATTAACAATGGCCAT', IUPACUnambiguousDNA())
```

```
Seq(' ATGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> translate(coding_dna)
Seq(' MAI VMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```







## Chapter 4

# Sequence Input/Output

In this chapter we'll discuss in more detail the Bio.SeqIO





### 4.1.3 Getting a list of the records in a sequence file

In the previous section we talked about the fact that `Bio.SeqIO.parse()` gives you a `SeqRecord`



```
>>> print first_record.annotations["organism"]  
Cypripedium irapeanum
```

In general, 'organism' is used for the scientific name (in Latin, e.g. *Arabidopsis thaliana*), while 'source'

```
from Bio import SeqIO
all_species = [seq_record.description.split()[1] for seq_record in \
                SeqIO.parse(open("ls_orchid.fasta"), "fasta")]
print all_species
```





```

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

    e.g. "gi|2765613|emb|Z78488.1|PTZ78488" -> "Z78488.1"
    """
    parts = record.id.split(373, ad)
    partd)= nds part[0]1= "g"s nds part[2]1= |em1"

    nctsi ortomusnenor bui l di ogrtthei ctsi oary(:)]TETG1001-67.0187589.29657cm0g0G0g0G1001-67.0187589.2965
    ndles =

    ally, r asrttenvy(:)]TETG1001-67.004.421229657cm0g0G0g0G1001-67.0104.421229657cmBT/F349.9626Tf04.42

```



## 4.4 Writing Sequence Files







```

from Bio import SeqIO
record_iterator = SeqIO.parse(open("ls_orchid.gbk"), "genbank")
out_handle = open("ls_orchid.tab", "w")
for record in record_iterator :
    out_handle.write(record.format("tab"))
out_handle.close()

```

While this style of code will work for a simple sequential file format like FASTA or the simple tab separated format used in this example, it will *not* work for more complex or interlaced file formats. Thi1(le)-3T-3T-3T-35828(y1-380

## Chapter 5

# Sequence Alignment Input/Output

In this chapter we'll discuss the Bio.AlignIO module, which is very similar to the Bio.SeqIO module from the previous chapter, but deals with `SeqRecord` objects instead of `SeqRecord` objects.

bio.AlignIO module



This will give the following output:

Alignment length 52

```
AEPNAATNYATEAMDSLKTOAI DLI SQTWPVTTVVVAGLVI RLFKKFSSKA - COATB_BPI KE/30-81
AEPNAATNYATEAMDSLKTOAI DLI SQTWPVTTVVVAGLVI KLFKKFVSRA - Q9T008_BPI KE/1-52
DGTSTATSYATEAMNSLKTOATDLI DOTWPVTVSVAVAGLAI RLFKKFSSKA - COATB_BPI 22/32-83
AEGDDP---AKAAFNSLOASATEYI GYAWAMVVVI VGATI GI KLFKKFTSKA - COATB_BPM13/24-72
AEGDDP---AKAAFDSLOASATEYI GYAWAMVVVI VGATI GI KLFKKFASKA - COATB_BPZJ2/1-49
```



All that has changed in this code is the filename and the format string. You'll get the same output as before, the sequences and record identifiers are the same. However, as you should expect, if you check each SeqRecord there is no annotation nor database cross-references because these are not included in the FASTA file format.

Note that rather than using the Sanger website, you could have used Bio.AlignIO to convert the original Stockholm format file into a FASTA file yourself (see below).

With any supported file format, you can load an alignment in exactly the same way just by changing the format string. For example, use "phylip" for PHYLIP files, "nexus" for NEXUS files or "emboss" for the alignments output by the EMBOSS tools. There is a full listing on the wiki page (<http://biopython.org/wiki/AlignIO>).

### 5.1.2 Multiple Alignments

The previous section focused on reading files containing a single alignment. In general however, files can contain more than one alignment, and to read these files we must use the Bio.AlignIO.parse() function.

Suppose you have a small alignment in PHYLIP format:

	5	6
Al pha		AACAAC
Beta		AACCCC
Gamma		ACCAAC
Del ta		CCACCA
Epsi lon		CCAAAC

If you wanted to bootstrap a phylogenetic tree using the PHYLIP tools, one of the steps would be to create a set of many resampled alignments using the tool bootseq. This would givele, usecleing6cc1(is)-oge(os)-1at:

If you want to read the full text of this article, please click on the link below. The full text of this article is available for free on the BioRxiv preprint server.

structure. The most common such situation is when alignments have been saved in the FASTA file format. For example consider the following:

```
>Al pha
ACTACGACTAGCTCAG--G
>Beta
ACTACCGCTAGCTCAGAAG
>Gamma
ACTACGGCTAGCACAGAAG
>Al pha
ACTACGACTAGCTCAGG--
>Beta
ACTACCGCTAGCTCAGAAG
>Gamma
ACTACGGCTAGCACAGAAG
```



```
al phabet = Gapped(IUPAC.unambi guous_dna)

al i gn1 = Al i gnment(al phabet)
al i gn1.add_sequence("Al pha", "ACTGCTAGCTAG")
al i gn1.add_sequence("Beta", "ACT-CTAGCTAG")
al i gn1.add_sequence("Gamma", "ACTGCTAGDTAG")

al i gn2 = Al i gnment(al phabet)
al i gn2.add_sequence("Del ta", "GTCAGC-AG")
al i gn2.add_sequence("Epi sl on", "GACAGCTAG")
al i gn2.add_sequence("Zeta", "GTCAGCTAG")

al i gn3 = Al i gnment(al phabet)
al i gn3.add_sequence("Eta", "ACTAGTACAGCTG")
al i gn3.add_sequence("Theta", "ACTAGTACAGCT-")
```

```
from Bio import AlignIO
alignments = AlignIO.parse(open("PF05371_seed.sth"), "stockholm")
handle = open("PF05371_seed.aln", "w")
```

COATB\_BPM1 AEGDDP---A KAFNSLQAS ATEYIGYAWA MVVIVGATI GIKLFKKFTS  
COATB\_BPZJ AEGDDP---A KAFDSLQAS ATEYIGYAWA MVVIVGATI GIKLFKKFAS  
Q9TQQ9\_BPF AEGDDP---A KAFDSLQAS ATEYIGYAWA MVVIVGATI GIKLFKKFTS





## Chapter 6

# 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 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.

Fortunately, the Biopython folks know this only too well, so they've developed lots of tools for dealing with BLAST and making things much easier. This section details how to use these tools and do useful things with them.

Dealing with BLAST can be split up into two steps, both of which can be done from within Biopython.

```
>>> 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
```



```
>>> blast_results = result_handle.read()
```

Next, we save this string in a file:

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

After doing this, the results are in the file `my_blast.xml` and the variable `blast_results` contains the

The important point is  $t_{80a9}(t) - 34y_{intttt80aintmtmtttcint}$  The ittpi

Now you can access each BLAST record in the list with an index as usual. If your BLAST file is huge

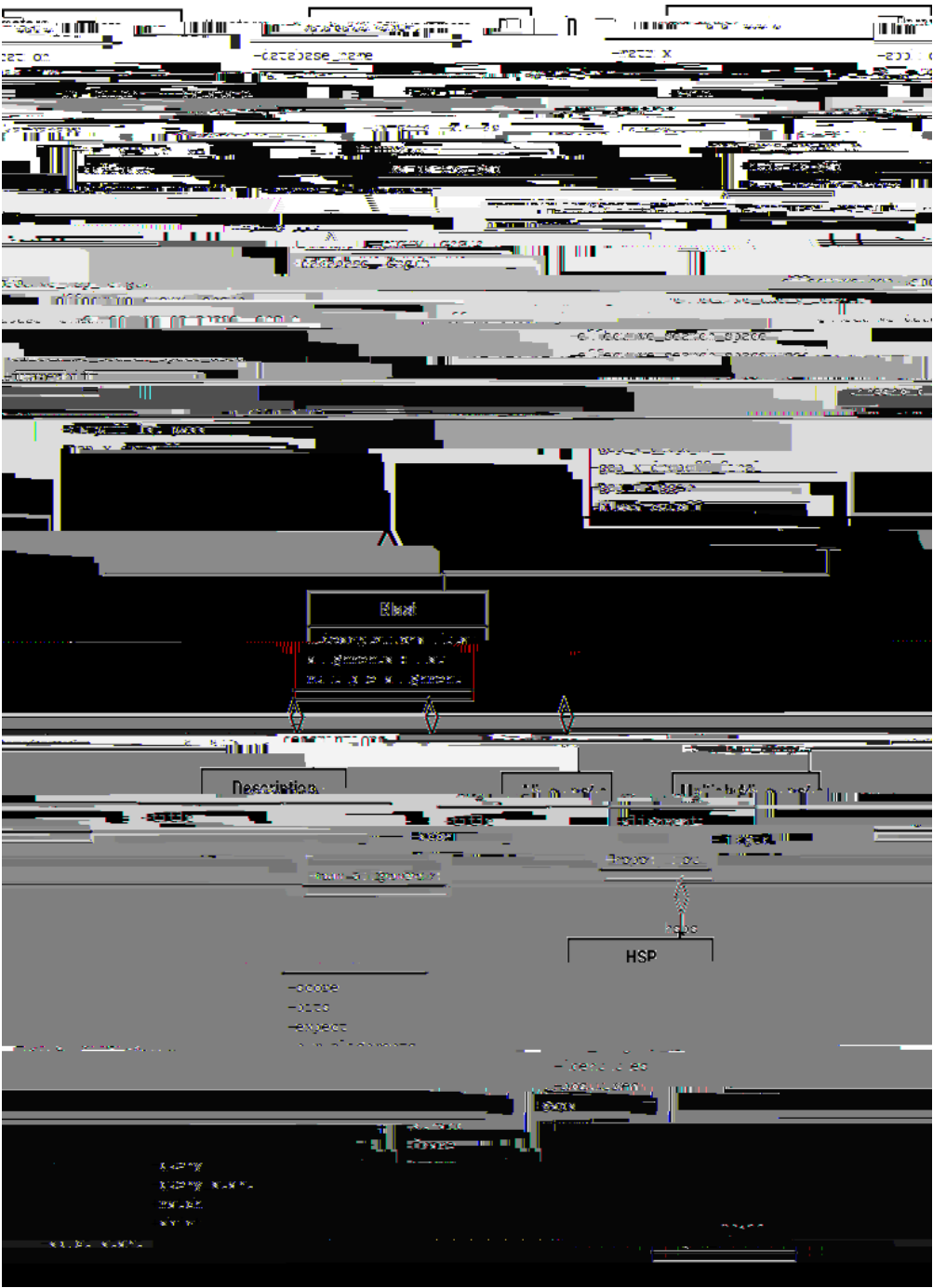


Figure 6.1: Class diagram fol6346346346389.1047-6-55-45346as55-4t047-6Re55-4cord047-6c334(d)1(iarepr346e55-4se55-4

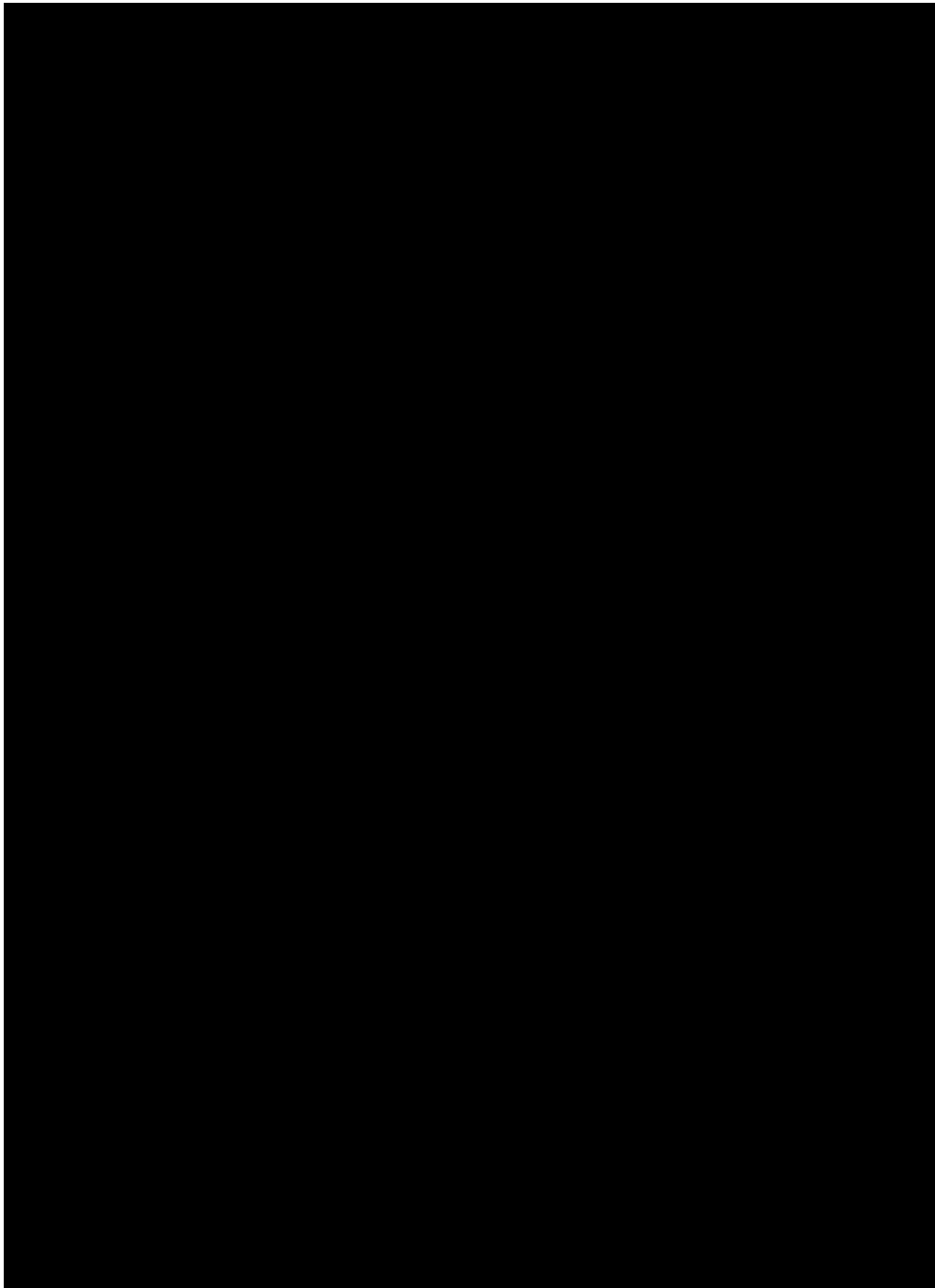


Figure 6.2: Class diagram for the PSIBlast Record class.



## 6.6 Deprecated BLAST parsers

Older versions of Biopython had parsers for BLAST output in plain text or HTML format. Over the years,

### 6.6.2 Parsing a file full of BLAST runs

Of course, local blast is cool because you can run a whole bunch of sequences against a database and get the results back without memory problems.

We can use blast to get an idea of what our blast reports in BlastRecord objects:

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

similar to the regular BlastParser







```

    <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>ncbi search</DbName>
    <DbName>nlmcatalog</DbName>
    <DbName>omica</DbName>
    <DbName>omic</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>
</InfoResult>

```

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:

```
>>> record["DbInfo"]["Description"]
'PubMed bibliographic record'
>>> record["DbInfo"]["Count"]
'17989604'
>>> record["DbInfo"]["LastUpdate"]
'2008/05/24 06:45'
```

Try `record["DbInfo"].keys()` for other information stored in this record.

## 7.3 ESearch: Searching the Entrez databases

To search any of these databases, we use `Bio.Entrez.esearch()`



## 7.5 ESummary: Retrieving summaries from primary IDs

ESummary retrieves document summaries from a list of primary IDs (see the [ESummary help page](#) for more information). In Biopython, ESummary is available as `Bio.Entrez.esummary()`. Using the search result

```

        /mol_type="genomic DNA"
        /specimen_voucher="FLAS: Blanco 2475"
        /db_xref="taxon: 256374"
gene    <1..>1302
        /gene="matK"
CDS     <1..>1302
        /gene="matK"
        /codon_start=1
        /transl_table=11
        /product="maturase K"
        /protein_id="ACC99456.1"
        /db_xref="GI: 186972395"
        /translation="I FYEPVEI FGYDNKSSLVLVKRLI TRMYQQNFLI SSVNDSNQKG
FWGHKHFFSSHFSQMVSEFGVI LEI PFSSQLVSSLEEKI PKYQNLRSI HSI FPFL
EDKFLHLNYVSDLLI PHPI HLEI LVQI LQCRI KDVPSLHLLRLLFHEYHNLNSLI TSK
KFI YAFSKRKKRFLWLLYNSVYVECEYLFQFLRKQSSYL RSTSSGVFLERTHLYVKI E
HLLVCCNSFQRI LCFLKDPFMHYVRYQGKAI LASKGTLI LMKKWKFHLVNFWSYFH
FWSQPYRI HI KQLSNYSFSLGYFSSVLENHLVVRNQMLENSFI I NLLTKKFDTI APV
I SLI GSLSKAQFCTVLGHPI SKPI WTD FSDSDI LDRFCRI CRNLCRYHSGSSKKQVLY
RI KYI LRLSCARTLARKHKSTVRTFMRRLGSGLLEEFFMEEE"

```

#### ORIGIN

```

1 attttttacg aacctgtgga aatttttggg tatgacaata aatctagttt agtacttgtg
61 aaacgtttta ttactcgaat gtatcaacag aattttttga tttcttcggt taatgattct
121 aaccaaaaag gattttgggg gcacaagcat tttttttctt ctcatttttc ttctcaaagt
181 gtatcagaag gttttggagt cattctggaa attccattct cgtcgcaatt agtatcttct
241 cttgaagaaa aaaaaatacc aaaatatcag aattttacgat ctattcattc aatatttccc
301 tttttagaag acaaattttt acatttgaat tatgtgtcag atctactaat accccatccc

```

```
>>> handle = Entrez.efetch(db="nucleotide", id="186972394", rettype="genbank")
>>> record = SeqIO.read(handle, "genbank")
>>> print record
ID: EU490707.1
Name: EU490707
Description: Selenipedium aequinoctiale maturase K (matK) gene, partial cds; chloroplast.
Number of features: 3
...
Seq('ATTTTTTACGAACCTGTGGAAATTTTGGTTATGACAATAATCTAGTTTAGTA...GAA', IUPACAmbiguousDNA())
```

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

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

If you want to download multiple records, you can use `Bio.Entrez.search()`, and then download the records with `Bio.Entrez.efetch()`.

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

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

## 7.10 Specialized parsers

The





Keep in mind that `records` is an iterator, so you can iterate through the records only once. If you want to save the records, you can convert them to a list:

```
>>> records = list(records)
```

```
>>> for row in record["eGQueryResl;E:
>>> reco>>
```





If you want to look at the raw GenBank files, you can read from this handle and print out the result:



```
>>> gi_list = search_results["IdList"]
>>> count = int(search_results["Count"])
>>> assert count == len(gi_list)
```

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

query_key=search_results["QueryKey"])
```

## Chapter 8

# Swiss-Prot, Prosite, Prodoc, and ExPASy

### 8.1 Bio.SwissProt: Parsing Swiss-Prot files

Swiss-Prot (<http://www.expasy.org/sprot>) is a hand-curated database of protein sequences. Biopython can parse the “plain text” Swiss-Prot file format, which is still used for the UniProt Knowledgebase which combined Swiss-Prot, TrEMBL and PIR-PSD. We do not (yet) support the UniProtKB XML file format.

#### 8.1.1 Parsing Swiss-Prot records

In Section 4.2.2, we described how to extract the sequence of a Swiss-Prot record as a SeqRecord object. Alternatively, you can store the Swiss-Prot record in a Bio.SwissProt.SProt.Record object, which in fact stores the complete information contained in the Swiss-Prot record. In this Section, we describe how to extract Bio.SwissProt.SProt.Record



Or, using a for loop over the record iterator:

```
>>> from Bio import SwissProt
>>> descriptions = []
>>> handle = open("uniprot_sprot.dat")
>>> for record in SwissProt.parse(handle) :
...     descriptions.append(record.description)
...
>>> len(descriptions)
290484
```

Because this is such a large input file, either way takes about seven minutes on my new desktop computer (using the uncompressed uniprot\_sprot.datunipre,nasng



The entries in this file can be parsed by the parse function in the Bio.SwissProt.KeyWList module.  
Each entry is then stored as a

```
>>> record.accessi on
'PS00005'
>>> record.name
'PKC_PHOSPHO_SITE'
>>> record.pdoc
'PDOC00005'
```

60(ifrec60(m)-32(de)1(c61(fyante)60(ene1(c61(h)33(P)1(P61(fe)-32(ds)5Eda60(i(434(TEo)JTEe)21(195)oute)nd)r).S0000528914.34629.962

**sprot\_search\_full** To search for a Swiss-Prot record

**sprot\_search\_de** To search for a Swiss-Prot record

To access this web server from a Python script, we use the Bio.ExPASy module.

```
>>> from Bio import ExPASy
```

```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prodoc_entry(' PDOC00001' )
>>> html = handle.read()
>>> output = open("myprodocrecord.html", "w")
>>> output.write(html)
>>> output.close()
```

For these functions, an invalid accession number returns an error message in HTML format.

## Chapter 9

# Cookbook – Cool things to do with it

### 9.1 Dealing with alignments



### 9.1.2 Calculating summary information



#### 9.1.4 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 `SubsMat.FreqTable` object (see section 10.4.2 for more information about `FreqTables`). The `FreqTable` object provides a standard for associating the dictionary with an `Alphabet`, similar to how the Biopython `Seq` class works.

To create a `FreqTable` object, from the frequency dictionary you just need to do:

```
from Bio.Alphabet import IUPAC
from Bio.SubsMat import FreqTable

e_freq_table = FreqTable.FreqTable(expect_freq, FreqTable.FREQ,
                                    IUPAC.unambiguous_dna)
```

Now that we've got that, calculating the relative information content for our region of the alignment is as simple as:

```
info_content = summary_align.information_content(5, 30,
                                                  e_freq_table = e_freq_table,
                                                  chars_to_ignore = ['N'])
```

Now, `info_content` will contain the relative information content over the region in relation to the expected frequencies.

The value return is calculated using base 2 as the logarithm base in the formula above. You can modify this by passing the parameter

### 9.2.1 Using common substitution matrices

- `exp_freq_table` – You can pass a table of expected frequencies for each alphabet. If supplied, this



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 disorder and behave eith

#### 9.4.1.1 Structure

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



The second field in the Residue id is the sequence identifier, an integer describing the position of the residue in the chain.



Since these residues belong to a different residue type (e.g. let's say Ser 60 and Cys 60) they should not

```

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

```

Print out the coordinates of all CA atoms in a structure with B th Bucture (with 31 atoms) (335-3352-150-91012)

```

hetfield_list=chain.get_list()
return hetfield_list

```

#### 9.4.5.1.1 Duplicate residues

#### 9.4.6 Other features

There are also some tools to analyze a crystal structure. Tools exist to superimpose two coordinate sets (SVDSuperimposer), to extract polypeptides from a structure (Polypeptide), to perform neighbor lookup (NeighborSearch) and to write out PDB files (PDBIO). The neighbor lookup is done using a KD tree module written in C++. It is very fast and also includes a fast method to find all point pairs within a certain distance of each other.

A Polypeptide object is simply a UserList of Residue objects. You can construct a list of Polypeptide objects from a Structure object as follows:

```
model_nr=1
polypeptide_list=build_peptides(structure, model_nr)
```

```
for polypeptide in polypeptide_list:
    print polypeptide
```

The Polypeptide objects are always created from a single Model (in this case model 1).

### 9.5 Bio.PopGen: Population genetics

```

        ('Ind1', [(1, 2), (3, 3), (200, 201)],
        ('Ind2', [(2, None), (3, 3), (None, None)],
    ],
    [
        ('Other1', [(1, 1), (4, 3), (200, 200)],
    ]
]

```

So we have two populations, the first with two individuals, the second with only one. The first individual of the first population is called Ind1, allelic information for each of the 3 loci follows. Please note that for any locus, information might be missing (see as an example, Ind2 above).

A few utility functions to manipulate GenePop records are made available, here is an example:

```
from Bio.PopGen import GenePop
```

```
#Imagine that you have loaded rec, as per the code snippet above...
```

```
rec.remove_population(pos)
```

```
#Removes a population from a record, pos is the population position in
# rec.populations, remember that it starts on position 0.
# rec is altered.
```

```
rec.remove_locus_by_position(pos)
```

```
#Removes a locus by its position, pos is the locus position in
# rec.loci_list, remember that it starts on position 0.
# rec is altered.
```

```
rec.remove_locus_by_name(name)
```

```
#Removes a locus by its name, name is the locus name as in
# rec.loci_list. If the name doesn't exist the function fails
# silently.
# rec is altered.
```

```
rec_loci = rec.split_in_loci()
```

```
#Splits a record in loci, that is, for each loci, it creates a new
# record, with a single loci and all populations.
```

### 9.5.2 Coalescent simulation



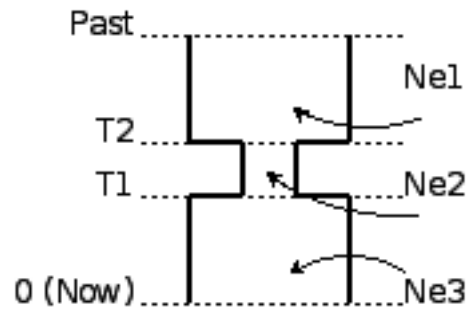


Figure 9.2: A bottleneck

```
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])])
```

Executing this code snippet will generate a file on the current directory called simple\_100\_300.par this







**sample.size** Average number of individuals sampled on each population.

**mut** MTflationide 0nsp-8(wisge)-334mnlafTfons.

```
sim_fst = ctrl.run_fdist_force_fst(npops = 15, nsamples = fd_rec.num_pops,  
    fst = fst, sample_size = samp_size, mut = 0, num_sims = 40000,  
    limit = 0.05)
```



```
>>> from Bio.Seq import Seq
>>> simple_seq = Seq("GATC")
>>> from Bio.SeqRecord import SeqRecord
>>> simple_seq_r = SeqRecord(simple_seq)
```

Additionally, you can also pass the id, name and description to the initialization function, but if not they will be set as strings indicating they are unknown, and can be modified subsequently:



**ref\_db** – This works along with ref to provide a cross sequence reference. If there is a reference, ref\_db will be set as None if the reference is in the same database, and will be set to the name of the database otherwise.

**strand**

I just mention this because sometimes I get confused between the two.





## 10.4 Substitution Matrices

### 10.4.1 SubsMat

This module provides a class and a few routines for generating substitution matrices, similar to BLOSUM or PAM matrices.

(e)



(a) `read_count(f)`: read a count file from stream `f`. Then convert to frequencies



## Chapter 11

# Where to go from here – contributing to Biopython

### 11.1 Maintaining a distribution for a platform

We try to release Biopython to make it as easy to install as possible for users. Thus, we try to provide the

**Macintosh** – We would love to find someone who wants to maintain a Macintosh distribution, and make



