**Course: Advanced Bioinformatics**

**Module title: Protein Structure**

**Module no. : 205**

This module depicts the structure of proteins.

**Swiss-Prot and ExPASy**

**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.

**Parsing Swiss-Prot records**

You can store the Swiss-Prot record in a Bio.SwissProt.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.Record objects from a Swiss-Prot file.

To parse a Swiss-Prot record, we first get a handle to a Swiss-Prot record. There are several ways to do so, depending on where and how the Swiss-Prot record is stored:

* Open a Swiss-Prot file locally:  
  >>> handle = open("myswissprotfile.dat")
* Open a gzipped Swiss-Prot file:
* >>> import gzip
* >>> handle = gzip.open("myswissprotfile.dat.gz")
* Open a Swiss-Prot file over the internet:
* >>> import urllib
* >>> handle = urllib.urlopen("http://www.somelocation.org/data/someswissprotfile.dat")
* Open a Swiss-Prot file over the internet from the ExPASy database:
* >>> from Bio import ExPASy
* >>> handle = ExPASy.get\_sprot\_raw(myaccessionnumber)

The key point is that for the parser, it doesn’t matter how the handle was created, as long as it points to data in the Swiss-Prot format.

We can use Bio.SeqIO to get file format agnostic SeqRecord objects. Alternatively, we can use Bio.SwissProt get Bio.SwissProt.Record objects, which are a much closer match to the underlying file format.

To read one Swiss-Prot record from the handle, we use the function read():

>>> from Bio import SwissProt

>>> record = SwissProt.read(handle)

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

We can now print out some information about this record:

>>> print(record.description)

'RecName: Full=Chalcone synthase 3; EC=2.3.1.74; AltName: Full=Naringenin-chalcone synthase 3;'

>>> for ref in record.references:

... print("authors:", ref.authors)

... print("title:", ref.title)

...

authors: Liew C.F., Lim S.H., Loh C.S., Goh C.J.;

title: "Molecular cloning and sequence analysis of chalcone synthase cDNAs of

Bromheadia finlaysoniana.";

>>> print(record.organism\_classification)

['Eukaryota', 'Viridiplantae', 'Streptophyta', 'Embryophyta', ..., 'Bromheadia']

To parse a file that contains more than one Swiss-Prot record, we use the parse function instead. This function allows us to iterate over the records in the file.

For example, let’s parse the full Swiss-Prot database and collect all the descriptions. You can download this from the [ExPAYs FTP site](ftp://ftp.expasy.org/databases/uniprot/current_release/knowledgebase/complete/uniprot_sprot.dat.gz) as a single gzipped-file uniprot\_sprot.dat.gz (about 300MB). This is a compressed file containing a single file, uniprot\_sprot.dat (over 1.5GB).

As described at the start of this section, you can use the Python library gzip to open and uncompress a .gz file, like this:

>>> import gzip

>>> handle = gzip.open("uniprot\_sprot.dat.gz")

However, uncompressing a large file takes time, and each time you open the file for reading in this way, it has to be decompressed on the fly. So, if you can spare the disk space you’ll save time in the long run if you first decompress the file to disk, to get the uniprot\_sprot.dat file inside. Then you can open the file for reading as usual:

>>> handle = open("uniprot\_sprot.dat")

As of June 2009, the full Swiss-Prot database downloaded from ExPASy contained 468851 Swiss-Prot records. One concise way to build up a list of the record descriptions is with a list comprehension:

>>> from Bio import SwissProt

>>> handle = open("uniprot\_sprot.dat")

>>> descriptions = [record.description for record in SwissProt.parse(handle)]

>>> len(descriptions)

468851

>>> descriptions[:5]

['RecName: Full=Protein MGF 100-1R;',

'RecName: Full=Protein MGF 100-1R;',

'RecName: Full=Protein MGF 100-1R;',

'RecName: Full=Protein MGF 100-1R;',

'RecName: Full=Protein MGF 100-2L;']

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)

468851

Because this is such a large input file, either way takes about eleven minutes on my new desktop computer (using the uncompressed uniprot\_sprot.dat file as input).

It is equally easy to extract any kind of information you’d like from Swiss-Prot records. To see the members of a Swiss-Prot record, use

>>> dir(record)

['\_\_doc\_\_', '\_\_init\_\_', '\_\_module\_\_', 'accessions', 'annotation\_update',

'comments', 'created', 'cross\_references', 'data\_class', 'description',

'entry\_name', 'features', 'gene\_name', 'host\_organism', 'keywords',

'molecule\_type', 'organelle', 'organism', 'organism\_classification',

'references', 'seqinfo', 'sequence', 'sequence\_length',

'sequence\_update', 'taxonomy\_id']

**Parsing the Swiss-Prot keyword and category list**

Swiss-Prot also distributes a file keywlist.txt, which lists the keywords and categories used in Swiss-Prot. The file contains entries in the following form:

ID 2Fe-2S.

AC KW-0001

DE Protein which contains at least one 2Fe-2S iron-sulfur cluster: 2 iron

DE atoms complexed to 2 inorganic sulfides and 4 sulfur atoms of

DE cysteines from the protein.

SY Fe2S2; [2Fe-2S] cluster; [Fe2S2] cluster; Fe2/S2 (inorganic) cluster;

SY Di-mu-sulfido-diiron; 2 iron, 2 sulfur cluster binding.

GO GO:0051537; 2 iron, 2 sulfur cluster binding

HI Ligand: Iron; Iron-sulfur; 2Fe-2S.

HI Ligand: Metal-binding; 2Fe-2S.

CA Ligand.

//

ID 3D-structure.

AC KW-0002

DE Protein, or part of a protein, whose three-dimensional structure has

DE been resolved experimentally (for example by X-ray crystallography or

DE NMR spectroscopy) and whose coordinates are available in the PDB

DE database. Can also be used for theoretical models.

HI Technical term: 3D-structure.

CA Technical term.

//

ID 3Fe-4S.

...

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 Bio.SwissProt.KeyWList.Record, which is a Python dictionary.

>>> from Bio.SwissProt import KeyWList

>>> handle = open("keywlist.txt")

>>> records = KeyWList.parse(handle)

>>> for record in records:

... print(record['ID'])

... print(record['DE'])

This prints

2Fe-2S.

Protein which contains at least one 2Fe-2S iron-sulfur cluster: 2 iron atoms

complexed to 2 inorganic sulfides and 4 sulfur atoms of cysteines from the

protein.

...

**Parsing Prosite records**

Prosite is a database containing protein domains, protein families, functional sites, as well as the patterns and profiles to recognize them. Prosite was developed in parallel with Swiss-Prot. In Biopython, a Prosite record is represented by the Bio.ExPASy.Prosite.Record class, whose members correspond to the different fields in a Prosite record.

In general, a Prosite file can contain more than one Prosite records. For example, the full set of Prosite records, which can be downloaded as a single file (prosite.dat) from the [ExPASy FTP site](ftp://ftp.expasy.org/databases/prosite/prosite.dat), contains 2073 records (version 20.24 released on 4 December 2007). To parse such a file, we again make use of an iterator:

>>> from Bio.ExPASy import Prosite

>>> handle = open("myprositefile.dat")

>>> records = Prosite.parse(handle)

We can now take the records one at a time and print out some information. For example, using the file containing the complete Prosite database, we’d find

>>> from Bio.ExPASy import Prosite

>>> handle = open("prosite.dat")

>>> records = Prosite.parse(handle)

>>> record = next(records)

>>> record.accession

'PS00001'

>>> record.name

'ASN\_GLYCOSYLATION'

>>> record.pdoc

'PDOC00001'

>>> record = next(records)

>>> record.accession

'PS00004'

>>> record.name

'CAMP\_PHOSPHO\_SITE'

>>> record.pdoc

'PDOC00004'

>>> record = next(records)

>>> record.accession

'PS00005'

>>> record.name

'PKC\_PHOSPHO\_SITE'

>>> record.pdoc

'PDOC00005'

and so on. If you’re interested in how many Prosite records there are, you could use

>>> from Bio.ExPASy import Prosite

>>> handle = open("prosite.dat")

>>> records = Prosite.parse(handle)

>>> n = 0

>>> for record in records: n+=1

...

>>> n

2073

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

>>> from Bio.ExPASy import Prosite

>>> handle = open("mysingleprositerecord.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.

**Parsing Prosite documentation records**

In the Prosite example above, the record.pdoc accession numbers 'PDOC00001', 'PDOC00004', 'PDOC00005' and so on refer to Prosite documentation. The Prosite documentation records are available from ExPASy as individual files, and as one file (prosite.doc) containing all Prosite documentation records.

We use the parser in Bio.ExPASy.Prodoc to parse Prosite documentation records. For example, to create a list of all accession numbers of Prosite documentation record, you can use

>>> from Bio.ExPASy import Prodoc

>>> handle = open("prosite.doc")

>>> records = Prodoc.parse(handle)

>>> accessions = [record.accession for record in records]

Again a read() function is provided to read exactly one Prosite documentation record from the handle.

**Parsing Enzyme records**

ExPASy’s Enzyme database is a repository of information on enzyme nomenclature. A typical Enzyme record looks as follows:

ID 3.1.1.34

DE Lipoprotein lipase.

AN Clearing factor lipase.

AN Diacylglycerol lipase.

AN Diglyceride lipase.

CA Triacylglycerol + H(2)O = diacylglycerol + a carboxylate.

CC -!- Hydrolyzes triacylglycerols in chylomicrons and very low-density

CC lipoproteins (VLDL).

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 "diglyceride lipase" (lines 3 through 5). The line starting with "CA" shows the catalytic activity of this enzyme. Comment lines start with "CC". The "PR" line shows references to the Prosite Documentation records, and the "DR" lines show references to Swiss-Prot records. Not of these entries are necessarily present in an Enzyme record.

In Biopython, an Enzyme record is represented by the Bio.ExPASy.Enzyme.Record class. This record derives from a Python dictionary and has keys corresponding to the two-letter codes used in Enzyme files. To read an Enzyme file containing one Enzyme record, use the read function inBio.ExPASy.Enzyme:

>>> from Bio.ExPASy import Enzyme

>>> with open("lipoprotein.txt") as handle:

... record = Enzyme.read(handle)

...

>>> record["ID"]

'3.1.1.34'

>>> record["DE"]

'Lipoprotein lipase.'

>>> record["AN"]

['Clearing factor lipase.', 'Diacylglycerol lipase.', 'Diglyceride lipase.']

>>> record["CA"]

'Triacylglycerol + H(2)O = diacylglycerol + a carboxylate.'

>>> record["PR"]

['PDOC00110']

>>> record["CC"]

['Hydrolyzes triacylglycerols in chylomicrons and very low-density lipoproteins

(VLDL).', 'Also hydrolyzes diacylglycerol.']

>>> record["DR"]

[['P11151', 'LIPL\_BOVIN'], ['P11153', 'LIPL\_CAVPO'], ['P11602', 'LIPL\_CHICK'],

['P55031', 'LIPL\_FELCA'], ['P06858', 'LIPL\_HUMAN'], ['P11152', 'LIPL\_MOUSE'],

['O46647', 'LIPL\_MUSVI'], ['P49060', 'LIPL\_PAPAN'], ['P49923', 'LIPL\_PIG'],

['Q06000', 'LIPL\_RAT'], ['Q29524', 'LIPL\_SHEEP']]

The read function raises a ValueError if no Enzyme record is found, and also if more than one Enzyme record is found.

The full set of Enzyme records can be downloaded as a single file (enzyme.dat) from the [ExPASy FTP site](ftp://ftp.expasy.org/databases/enzyme/enzyme.dat), containing 4877 records (release of 3 March 2009). To parse such a file containing multiple Enzyme records, use the parse function in Bio.ExPASy.Enzyme to obtain an iterator:

>>> from Bio.ExPASy import Enzyme

>>> handle = open("enzyme.dat")

>>> records = Enzyme.parse(handle)

We can now iterate over the records one at a time. For example, we can make a list of all EC numbers for which an Enzyme record is available:

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

**Accessing the ExPASy server**

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

**get\_prodoc\_entry**

To download a Prosite documentation record in HTML format

**get\_prosite\_entry**

To download a Prosite record in HTML format

**get\_prosite\_raw**

To download a Prosite or Prosite documentation record in raw format

**get\_sprot\_raw**

To download a Swiss-Prot record in raw format

**sprot\_search\_ful**

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.

**Retrieving a Swiss-Prot record**

Let’s say we are looking at chalcone synthases for Orchids. Chalcone synthase is involved in flavanoid biosynthesis in plants, and flavanoids make lots of cool things like pigment colors and UV protectants.

If you do a search on Swiss-Prot, you can find three orchid proteins for Chalcone Synthase, id numbers O23729, O23730, O23731. Now, let’s write a script which grabs these, and parses out some interesting information.

First, we grab the records, using the get\_sprot\_raw() function of Bio.ExPASy. This function is very nice since you can feed it an id and get back a handle to a raw text record (no html to mess with!). We can the use Bio.SwissProt.read to pull out the Swiss-Prot record, or Bio.SeqIO.read to get a SeqRecord. The following code accomplishes what I just wrote:

>>> from Bio import ExPASy

>>> from Bio import SwissProt

>>> accessions = ["O23729", "O23730", "O23731"]

>>> records = []

>>> for accession in accessions:

... handle = ExPASy.get\_sprot\_raw(accession)

... record = SwissProt.read(handle)

... records.append(record)

If the accession number you provided to ExPASy.get\_sprot\_raw does not exist, then SwissProt.read(handle) will raise a ValueError. You can catch ValueException exceptions to detect invalid accession numbers:

>>> for accession in accessions:

... handle = ExPASy.get\_sprot\_raw(accession)

... try:

... record = SwissProt.read(handle)

... except ValueException:

... print("WARNING: Accession %s not found" % accession)

... records.append(record)

**Searching Swiss-Prot**

Now, you may remark that I knew the records’ accession numbers beforehand. Indeed, get\_sprot\_raw() needs either the entry name or an accession number. When you don’t have them handy, you can use one of the sprot\_search\_de() or sprot\_search\_ful() functions.

sprot\_search\_de() searches in the ID, DE, GN, OS and OG lines; sprot\_search\_ful() searches in (nearly) all the fields. They are detailed on <http://www.expasy.org/cgi-bin/sprot-search-de> and <http://www.expasy.org/cgi-bin/sprot-search-ful> respectively. Note that they don’t search in TrEMBL by default (argument trembl). Note also that they return html pages; however, accession numbers are quite easily extractable:

>>> from Bio import ExPASy

>>> import re

>>> handle = ExPASy.sprot\_search\_de("Orchid Chalcone Synthase")

>>> # or:

>>> # handle = ExPASy.sprot\_search\_ful("Orchid and {Chalcone Synthase}")

>>> html\_results = handle.read()

>>> if "Number of sequences found" in html\_results:

... ids = re.findall(r'HREF="/uniprot/(\w+)"', html\_results)

... else:

... ids = re.findall(r'href="/cgi-bin/niceprot\.pl\?(\w+)"', html\_results)

**Retrieving Prosite and Prosite documentation records**

Prosite and Prosite documentation records can be retrieved either in HTML format, or in raw format. To parse Prosite and Prosite documentation records with Biopython, you should retrieve the records in raw format. For other purposes, however, you may be interested in these records in HTML format.

To retrieve a Prosite or Prosite documentation record in raw format, use get\_prosite\_raw(). For example, to download a Prosite record and print it out in raw text format, use

>>> from Bio import ExPASy

>>> handle = ExPASy.get\_prosite\_raw('PS00001')

>>> text = handle.read()

>>> print(text)

To retrieve a Prosite record and parse it into a Bio.Prosite.Record object, use

>>> from Bio import ExPASy

>>> from Bio import Prosite

>>> handle = ExPASy.get\_prosite\_raw('PS00001')

>>> record = Prosite.read(handle)

The same function can be used to retrieve a Prosite documentation record and parse it into a Bio.ExPASy.Prodoc.Record object:

>>> from Bio import ExPASy

>>> from Bio.ExPASy import Prodoc

>>> handle = ExPASy.get\_prosite\_raw('PDOC00001')

>>> record = Prodoc.read(handle)

For non-existing accession numbers, ExPASy.get\_prosite\_raw returns a handle to an emptry string. When faced with an empty string, Prosite.read and Prodoc.read will raise a ValueError. You can catch these exceptions to detect invalid accession numbers.

The functions get\_prosite\_entry() and get\_prodoc\_entry() are used to download Prosite and Prosite documentation records in HTML format. To create a web page showing one Prosite record, you can use

>>> from Bio import ExPASy

>>> handle = ExPASy.get\_prosite\_entry('PS00001')

>>> html = handle.read()

>>> output = open("myprositerecord.html", "w")

>>> output.write(html)

>>> output.close()

and similarly for a Prosite documentation record:

>>> 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.

**Scanning the Prosite database**

[ScanProsite](http://www.expasy.org/tools/scanprosite/) allows you to scan protein sequences online against the Prosite database by providing a UniProt or PDB sequence identifier or the sequence itself. For more information about ScanProsite, please see the [ScanProsite documentation](http://www.expasy.org/tools/scanprosite/scanprosite-doc.html) as well as the [documentation for programmatic access of ScanProsite](http://www.expasy.org/tools/scanprosite/ScanPrositeREST.html).

You can use Biopython’s Bio.ExPASy.ScanProsite module to scan the Prosite database from Python. This module both helps you to access ScanProsite programmatically, and to parse the results returned by ScanProsite. To scan for Prosite patterns in the following protein sequence:

MEHKEVVLLLLLFLKSGQGEPLDDYVNTQGASLFSVTKKQLGAGSIEECAAKCEEDEEFT

CRAFQYHSKEQQCVIMAENRKSSIIIRMRDVVLFEKKVYLSECKTGNGKNYRGTMSKTKN

you can use the following code:

>>> sequence = "MEHKEVVLLLLLFLKSGQGEPLDDYVNTQGASLFSVTKKQLGAGSIEECAAKCEEDEEFT

CRAFQYHSKEQQCVIMAENRKSSIIIRMRDVVLFEKKVYLSECKTGNGKNYRGTMSKTKN"

>>> from Bio.ExPASy import ScanProsite

>>> handle = ScanProsite.scan(seq=sequence)

By executing handle.read(), you can obtain the search results in raw XML format. Instead, let’s use Bio.ExPASy.ScanProsite.read to parse the raw XML into a Python object:

>>> result = ScanProsite.read(handle)

>>> type(result)

<class 'Bio.ExPASy.ScanProsite.Record'>

A Bio.ExPASy.ScanProsite.Record object is derived from a list, with each element in the list storing one ScanProsite hit. This object also stores the number of hits, as well as the number of search sequences, as returned by ScanProsite. This ScanProsite search resulted in six hits:

>>> result.n\_seq

1

>>> result.n\_match

6

>>> len(result)

6

>>> result[0]

{'signature\_ac': u'PS50948', 'level': u'0', 'stop': 98, 'sequence\_ac': u'USERSEQ1', 'start': 16, 'score': u'8.873'}

>>> result[1]

{'start': 37, 'stop': 39, 'sequence\_ac': u'USERSEQ1', 'signature\_ac': u'PS00005'}

>>> result[2]

{'start': 45, 'stop': 48, 'sequence\_ac': u'USERSEQ1', 'signature\_ac': u'PS00006'}

>>> result[3]

{'start': 60, 'stop': 62, 'sequence\_ac': u'USERSEQ1', 'signature\_ac': u'PS00005'}

>>> result[4]

{'start': 80, 'stop': 83, 'sequence\_ac': u'USERSEQ1', 'signature\_ac': u'PS00004'}

>>> result[5]

{'start': 106, 'stop': 111, 'sequence\_ac': u'USERSEQ1', 'signature\_ac': u'PS00008'}

Other ScanProsite parameters can be passed as keyword arguments; see the [documentation for programmatic access of ScanProsite](http://www.expasy.org/tools/scanprosite/ScanPrositeREST.html) for more information. As an example, passing lowscore=1 to include matches with low level scores lets use find one additional hit:

>>> handle = ScanProsite.scan(seq=sequence, lowscore=1)

>>> result = ScanProsite.read(handle)

>>> result.n\_match