

# Programming for Chemical and Life Science Informatics

I573 - Week 5  
(Programming with BioPython)

Rajarshi Guha

10<sup>th</sup> February, 2009

# Online Resources

- [Main Site](#)
- [Tutorial](#)
- [API Docs](#)

# What Does It Provide?

- Classes and methods to handle sequences
- Handling PDB files and various sequence formats
- Interacting with various online resources
  - ▶ SWISS-PROT, PubMed, GenBank, KEGG
- Substitution matrices
- Dealing with sequence databases ([BioSQL](#))
- Deal with restriction enzymes, microarray data
- Basic support for pathway analysis
- Classification and clustering
- Interfaces to a variety of programs (BLAST, Clustal, EMBOSS etc)

# Sequences

- Fundamental bioinformatics objects
- A sequence is constructed from an alphabet which might indicate
  - ▶ Protein, DNA, RNA
- BioPython sequences behave like strings, but also contain extra information regarding the alphabet

```
>> from Bio.Seq import Seq
>> seq = Seq('GCATGACGTTATTACGACTCTGTCACGCCGCGGTGCG')
>> print seq
Seq('GCATGACGTTATTACGACTCTGTCACGCCGCGGTGCG', Alphabet())
```

# Seq Objects as Strings

- Seq objects are designed to behave as strings
- Length of a sequence

```
>> seq = Seq('GCATGACGTTATTACGACTCTGTCACGCCGCGGTGCG')
>> print len(seq)
37
```

- Evaluate the GC content of a sequence

```
>>> (seq.count('G')+seq.count('C')) / float(len(seq))
0.59459459459459463
```

- Get the complement sequence

```
>>> print seq.complement()
Seq('CGTACTGCAATAATGCTGAGACAGTGCGGCGCCACGC', Alphabet())
```

# Sequences as Strings

- Seq objects also behave like traditional Python sequences
- Extract a subsequence

```
>>> subseq = seq[1:8]
>>> subseq
Seq('CATGACG', Alphabet())
```

- Add two sequences

```
>>> seq1 = Seq('GCATGGC')
>>> seq2 = Seq('AATGCGC')
>>> print seq1+seq2
Seq('GCATGGCAATGCGC', Alphabet())
```

# Sequences - Alphabets

- So far we have not defined an alphabet for our sequences
- The Seq object contains components to hold this information

```
>> print seq
Seq('GCATGACGTTATTACGACTCTGTCACGCCGCGGTGCG', Alphabet())
>> seq.__dict__
{'alphabet': Alphabet(), 'data': 'GCATGACGTTATTACGACTCTGTCACGCCGCGGTGCG'}
```

- The data component holds the actual sequence
- The alphabet component describes the meaning of the symbols
  - ▶ In this case the alphabet is empty

# Sequences - Alphabets

- A number of alphabets are defined
  - ▶ IUPACUnambiguousDNA - represents the 4 DNA bases
  - ▶ IUPACProtein - represents the single letter amino acid codes
  - ▶ ExtendedIUPACProtein - supports additional elements such as asparagine (Asx), selenocystein (Sec) etc
- Create a sequence using the unambiguous DNA alphabet

```
>>> from Bio.Alphabet import IUPAC
>>> dnaalpha = IUPAC.unambiguous_dna
>>> seq = Seq('GATCGATGGGCCTAT', dnaalpha)
>>> print seq
Seq('GATCGATGGGCCTAT', IUPACUnambiguousDNA())
```

# Sequences - Alphabets

- Alphabets are very useful, since they are carried over with various sequence operations

```
>>> seq = Seq('GACTGAGATCS', IUPAC.unambiguous_dna)
>>> subseq = seq[1:5]
>>> subseq
Seq('ATCG', IUPACUnambiguousDNA())
```

- Prevents you from mixing sequences of different types

```
>>> seq_dna = Seq('GCTGA', IUPAC.unambiguous_dna)
>>> seq_prot = Seq('EKVKVR', IUPAC.protein)
>>> seq_dna + seq_prot
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/usr/lib/python2.4/site-packages/Bio/Seq.py", line 63, in __add__
    raise TypeError, ("incompatable alphabets", str(self.alphabet),
TypeError: ('incompatable alphabets', 'IUPACUnambiguousDNA()', 'IUPACProtein()')
```

# Transcribing Sequences

- Generate an RNA sequence from a DNA sequence
- Since we use an unambiguous DNA alphabet we should use the corresponding transcriber

```
>>> seq = Seq('GCGATGGCAATGC', IUPAC.unambiguous_dna)
>>> from Bio import Transcribe
>>> t = Transcribe.unambiguous_transcriber
>>> rna_seq = t.transcribe(seq)
>>> rna_seq
Seq('GCGAUGGCAAUGC', IUPACUnambiguousRNA())
```

# Translating Sequences

- A common task is to convert a DNA sequence to a protein sequence
- Requires the use of translation tables since different organisms can use different codons to represent an amino acid
- Multiple tables are available in the `Translation` package
- The translation tables are available for ambiguous and unambiguous alphabets

```
>>> from Bio import Translate
>>> trans = Translate.unambiguous_dna_by_id
```

# Translating Sequences

- `trans` is a dictionary object, keyed by integers starting from 1
- Each value is a translation table taken from [here](#)

---

Key	Table
1	Standard
2	Vertebrate mitochondrial
3	Yeast mitochondrial
4	Mold, Protozoan, Coelenterate mitochondrial
5	Invertebrate mitochondrial
...	...

---

# Translating Sequences

- Once you've chosen which translation table you need, generate the protein sequence

```
>>> from Bio.Seq import Seq
>>> from Bio import Translate
>>> from Bio.Alphabet import IUPAC
>>> seq = Seq('GACGTGGCCATA', IUPAC.unambiguous_dna)
>>> std_trans = Translate.unambiguous_dna_by_id[1]
>>> print std_trans.translate(seq)
Seq('DVAI', HasStopCodon(IUPACProtein(), '*'))
```

# Translating Sequences

- By default, the translators will carry on through a sequence even if a stop codon (TAA, TAG, TGA) is present

```
>>> seq = Seq('GACGTGGCCATATAAACG', IUPAC.unambiguous_dna)
>>> print std_trans.translate(seq)
Seq('DVAI*T', HasStopCodon(IUPACProtein(), '*'))
```

- If you don't want to go past a stop codon

```
>>> seq = Seq('GACGTGGCCATATAAACG', IUPAC.unambiguous_dna)
>>> print std_trans.translate_to_stop(seq)
Seq('DVAI', IUPACProtein())
```

# Reading Sequences

- In general we'll get sequences from some database
- Formats include [FASTA](#), [GenBank](#), [SwissProt](#)
- Use the SeqIO package to read various formats

```
from Bio import SeqIO
handle = open('mtuber_h37rv_proteins.fasta')
for rec in SeqIO.parse(handle, "fasta"):
    print "%s %d" % (rec.id, len(rec.seq))
```

- You need to specify the file format being used
- rec is a SeqRecord object
  
- The file is linked on the course web page for this class
- Contains protein sequences for *M. tuberculosis* H37Rv

# Reading Sequences

- It's also possible to directly get sequences of the Internet
- Can access GenBank or SwissProt databases
- You'll need identifiers relevant for the database

```
from Bio import GenBank
from Bio import SeqIO
handle = GenBank.download_many(['115313949', '6680412'])
for seq in SeqIO.parse(handle, "genbank") :
    print '%s %d' % (seq.id, len(seq.seq))
    print "from: %s" % seq.annotations['source']
```

## Reading Sequences

- Depending on the source, a sequence will have extra information
- If you got the sequence from GenBank, the annotations component will be available

```
>>> handle = GenBank.download_many(['115313949', '6680412'])
>>> seqlist = list(SeqIO.parse(handle, 'genbank'))
>>> for key in seqlist[0].annotations:
...     print key, '->', seqlist[0].annotations[key]
sequence_version -> 1
source -> Mycobacterium tuberculosis str. Haarlem
keywords -> ['WGS']
date -> 01-FEB-2008
organism -> Mycobacterium tuberculosis str. Haarlem
gi -> 115313949
...
```

# Writing Sequences

- Writing out a sequence is pretty much the same as reading it in
- Specify a file object and an output format

```
ohandle = open('myseq.fasta', 'w')
ihandle = GenBank.download_many(['115313949', '6680412'])
seqiter = SeqIO.parse(ihandle, "genbank")
SeqIO.write(seqiter, ohandle, "fasta")
ohandle.close()
```

- You can also supply a list of SeqRecord objects

# Running BLAST

- **BLAST** is a ubiquitous tool for finding similar sequences
- Can be run locally or via web sites that host the program
- BioPython can interface with both
- *cheminfo* does not support local BLAST searches
- We'll focus on running BLAST remotely
  - ▶ BioPython uses the NCBI BLAST server

# Running BLAST

- First read in a sequence and retain just the raw sequence data

```
>>> handle = GenBank.download_many(['115313949', '6680412'])  
>>> seqlist = list(SeqIO.parse(handle, 'genbank'))  
>>> query_seq = seqlist[1].seq.data
```

- Run BLAST on it

```
>>> from Bio.Blast import NCBIWWW  
>>> handle = NCBIWWW.qblast("blastn", "nr", query_seq)
```

- You can specify which program to use, arguments for the program
- Here we use `blastn` and search the non-redundant database of sequences
- `handle` is just a file object - we don't have the results yet

# Using BLAST Results

- Once we've performed the BLAST run, we should save the results

```
results = handle.read()
```

- You can only call `read()` once - after that it returns an empty string
- Write the results out to a file
- Reuse this file to actually parse the results

```
ofile = open('blastresult.xml', 'w')  
ofile.write(results)  
ofile.close()
```

- We don't have to write out the results to a file
- It saves you from rerunning BLAST (which might take some time)

# Parsing BLAST Results

- Now that we have BLAST results, we need to parse it
- A BLAST run will generate one or more BLAST records
- If you just have one query sequence, you get one record

```
handle = open('blastresult.xml')
from Bio.Blast import NCBIXML
recs = NCBIXML.parse(handle)
```

- recs is an iterator

```
record = recs.next()
```

- record will contain the information regarding matching sequences
- See docs for [Record.Blast](#) for full details

# Extracting BLAST Results

- We will just look at the alignments
- How many alignments did we get?

```
>>> print len(record.alignments)
50
```

- The description member of the record object contains useful information such as title of the hit, E-value etc.
- What is the lowest E-value we got?

```
>>> evalues = [x.e for x in record.descriptions]
>>> print 'Max E-value = %E' % (max(evalues))
Max E-value = 7.213540E-11
>>> print 'Min E-value = %E' % (min(evalues))
Min E-value = 0.000000
```

# Extracting BLAST Results

- For a given alignment we're interested in the [High Scoring Sequence Pair's](#)
- We only consider those pairs whose score is below a cutoff
- Get the first alignment

```
align = record.alignments[0]
```

- Here `align` is an instance of class [Alignment](#)
- Contains a list of [HSP](#) objects

# Extracting BLAST Results

- Loop over the HSP's for this alignment
- Consider HSP's whose expectation value is less than 0.4
- Print out part of the alignment

```
align = record.alignments[0]
for hsp in align.hsps:
    if hsp.expect < 0.04:
        print align.title
        print align.length
        print hsp.expect
        print hsp.query[0:10]+'...'
        print hsp.match[0:10]+'...'
        print hsp.sbjct[0:10]+'...'
```



# Searching PubMed

- Getting article details from PubMed is possible in a number of ways
- BioPython supports direct queries

```
from Bio import PubMed

search_term = 'falciparum'
ids = PubMed.search_for(search_term)
```

- This returns a list of PubMed ID's

# Searching PubMed

- We need to create a parser object and then a PubMed *dictionary*
- The *dictionary* takes a single ID, connects to PubMed and returns an instance of a [Medline.Record](#)

```
>>> from Bio import PubMed
>>> from Bio import Medline

>>> parser = Medline.RecordParser()
>>> mdict = PubMed.Dictionary(parser = parser)
>>> article = mdict[ ids[0] ]
>>> article.title
'Transfusion-transmitted malaria in a kidney transplant recipient'
>>> article.authors
['Alkhunaizi AM', 'Al-Tawfiq JA', 'Al-Shawaf MH']
```

# Interacting with KEGG

- BioPython provides support for KEGG
- Handles ligand, enzyme and map files
- You have to have access to the raw KEGG data files
  - ▶ enzyme
  - ▶ compound
- Doesn't support genes (yet)
- *The current version of KEGG support cannot read the complete enzyme file*

# Interacting with KEGG

- First download the enzyme data file
- Get a file object to it, and then loop over the records

```
>>> from Bio.KEGG import Enzyme

>>> handle = open('enzyme')
>>> for rec in Enzyme.Iterator(handle):
...     print rec.entry, rec.pathway
1.1.1.3 [('PATH', 'map00260', 'Glycine, serine and threonine me
          ('PATH', 'map00300', 'Lysine biosynthesis')]
1.1.1.4 [('PATH', 'map00650', 'Butanoate metabolism')]
1.1.1.5 [('PATH', 'map00650', 'Butanoate metabolism')]
```

- Looping over the iterator returns an [KEGG.Enzyme.Record](#) object
- Easy to identify which pathways an enzyme is involved in

# Interacting with KEGG

- Working with the compound data is also easy
- Get the data file, create a file object and loop over an iterator

```
>>> from Bio.KEGG import Compound
>>> handle = open('compound')
>>> for rec in Compound.Iterator(handle):
...     print rec.entry, rec.pathway
C14449 []
C14450 [('PATH', 'map00627', '1,4-Dichlorobenzene degradation')]
C14451 []
C14452 []
C14453 [('PATH', 'map00980', 'Metabolism of xenobiotics by
        cytochrome P450')]
```

- `rec` is an instance of [KEGG.Compound.Record](#)
- Identifies which pathways and enzymes a compound is associated with

# Sequence Utility Methods

- The `SeqUtils` package provides a number of useful functions
- Good for characterizing various types of sequences
  - ▶ Evaluate GC content and skew
  - ▶ Convert 1-letter AA codes to 3-letter codes
  - ▶ Isoelectric point, melting temperature calculation
- Melting point calculation allows you to specify DNA and salt concentrations

```
>>> from Bio.SeqUtils import MeltingTemp
>>> MeltingTemp.Tm_staluc('GATCGCTCGCC')
37.541593985166344
```

# Sequence Utility Methods

- The `ProteinAnalysis` class is useful to calculate various protein sequence characteristics

```
>>> from Bio.SeqUtils.ProtParam import ProteinAnalysis
>>> analysis = ProteinAnalysis("ACFGDVKKWN")
>>> analysis.flexibility()
[0.97833333333333339]
>>> analysis.aromaticity()
0.20000000000000001
>>> analysis.instability_index()
62.399999999999991
```