13 Bioinformatics Pipelines II

13.1 Overview

Building on our initial forays into building bioinformatics pipelines last week, we now turn to a more complicated example that integrates BioPython along with several command line programs. This pipeline will incorporate such features as web based queries and conversion of information between different file formats.

13.2 The Pipeline

The tasks carried out by the pipeline will be as follows:

- Read in a nucleotide sequence from a FASTA file
- Translate the nucleotide sequence to an amino acid sequence
- Do a blastp search against human and fly proteins in the Swiss-Prot database using an interface to the NCBI web version of BLAST
- Download protein sequences for the best blast hits from Swiss-Prot
- Use MAFFT to do a multiple alignment of the original amino acid sequence and the presumed orthologs generated via the blast search
- Analyze the query protein for known protein domains using HMMER and Pfam

13.3 Installing Additional Software and Modules

Before we start building our pipeline, we'll need to install a few additional tools., including the Biopython library, and two commonly used bioinformatics sofware packages called HMMER and MAFFT. HMMER and MAFFT will be installed at using the system level package manager (Debian's APT system) using the command line tool, aptitude. Biopython will be installed using the Conda package manager, for managing the Python environment.

13.3.1 Installing HMMER

HMMER is an implementation of a profile Hidden Markov Model (HMM) for protein sequence analysis. You can read up on HMMER at the HMMER website. We will use it here for finding protein domains in sequences in conjuction with the PFAM database. To install HMMER type the following commands in the Bash shell (*not* from the Python interpreter):

```
$ sudo aptitude update # insures the list of packages is up to date
$ sudo aptitude install HMMER
```

Get the PFAM HMM library

We will be using the Pfam database (Release 27) in conjunction with HMMER to search for known protein domains in our sequences of interest. To download PFAM, you can use the `curl` command line tool:

```
$ curl -O ftp://ftp.sanger.ac.uk/pub/databases/Pfam/releases/Pfam27.0/Pfam-
    A.hmm.gz
```

This is a large file (202MB) and decompresses to an even larger file (approx. 1GB). Make sure you have adequate disk space. Once the download completes you can unzip it as follows:

```
$ gunzip Pfam-A.hmm.gz
```

13.3.2 Installing MAFFT

MAFFT is a multiple sequence alignment program. It's relatively fast and a number of studies have shown that it is amongst the best performing multiple sequence aligners. MAFFT is usually the sequence aligner I reach for first. Clustalw is the 'classic' alignment tool, so it's useful to have on your system, but MAFFT usually gives better alignments (though Clustalw2 is supposed to address some of the short-comings of the older versions of Clustalw). See the MAFFT website for additional references and information.

```
$ sudo aptitude install mafft
```

13.3.3 Installing Biopython

Biopython is easily installed using the `conda` package manager:

```
$ conda install biopython
```

13.4 Biopython

Now we turn our attention to Biopython. As we build our pipeline I will first demonstrate the use of various modules, classes, and functions in the interactive shell and then I will give a set of functions that consolidate the commands to make them convenient to use.

13.4.1 Test files

Download the file `unknown1.fas` and `unknown2.fas` from the class website. I recommend you place these in `~/tmp`.

13.4.2 Reading in a single sequence from a FASTA file

Fire up and ipython interpreter, either a text based command line (`ipython --pylab`) or an ipython notebook (`ipython notebook --pylab=inline`).

We'll start by showing how to read sequence data out of a FASTA file:

```
>>> cd ~/tmp
>>> from Bio import SeqIO
>>> u1 = SeqIO.read('unknown1.fas','fasta')
>>> type(u1)
<class 'Bio.SeqRecord.SeqRecord'>
>>> u1
SeqRecord(seq=Seq('ATGATGAATTTTTTTACATCAAAATCGTCGAAT
CAGGATACTGGATTTAGCTCT...TGA', SingleLetterAlphabet()),
id='YHR205W', name='YHR205W', description='YHR205W  Chr 8', dbxrefs=[])
>>> u1.name
'YHR205W'
>>> u1.description
'YHR205W  Chr 8'
>>> u1.seq
Seq('ATGATGAATTTTTTTACATCAAAATCGTCGAATCAGGATACTGG
ATTTAGCTCT...TGA', SingleLetterAlphabet())
>>> u1.seq[:10]
Seq('ATGATGAATT', SingleLetterAlphabet())
>>> u1.seq[0]
'A'
>>> u1.seq[9]
'T'
>>> u1.seq[:10].tostring()
'ATGATGAATT'
>>> u1.seq.translate()[:10]
Seq('MMNFFTSKSS', HasStopCodon(ExtendedIUPACProtein(), '*'))
```

`SeqIO` is a sub-module of the top-level module Biopython module `Bio`. The function `SeqIO.read()` reads a single sequence object from a file and returns an instance of a

`SeqRecord` class (defined in the Biopython package). A *class* is a programming concept that groups data and functions that operate on that data into a single object. For example, in the code above we used the `.name` and `.description` attributes to examine information about the sequence (this information was retrieved from the FASTA file itself). A `SeqRecord` holds a `Seq` object (yet another class!) as well as accessory information like the name of the sequence, a description, etc. `Seq` objects act very much like strings in terms of slicing and element access but they also have specialized function like `.translate()` that can be used to translate a nucleotide sequence into a peptide sequence.

Reading in multiple sequences from a FASTA file

In the code above we demonstrated how to read a single sequence from a FASTA file. Here we demonstrate how to read multiple sequences. The key difference is the use of the `SeqIO.parse()` function rather than `SeqIO.read()`.

```
>>> u2 = SeqIO.parse('unknown2.fas', 'fasta')
>>> type(u2)
<type 'generator'>
>>> s1 = u2.next()
>>> type(s1)
<class 'Bio.SeqRecord.SeqRecord'>
>>> s1
SeqRecord(seq=Seq('ATGTCATCAAAACCTGATACTGGTTCGGA
AATTTCTGGCCCTCAGCGACAGGAA...TGA', SingleLetterAlphabet()),
id='YJL005W', name='YJL005W', description='YJL005W', dbxrefs=[])
>>> s1.seq
Seq('ATGTCATCAAAACCTGATACTGGTTCGGAAATTTCTGGCC
CTCAGCGACAGGAA...TGA', SingleLetterAlphabet())
>>> s2 = u2.next()
>>> s2
SeqRecord(seq=Seq('ATGTCATCAAATCATGCTATTAGTCCAGAA
ACTTCTGGCTCTCATGAGCAACAA...TGA', SingleLetterAlphabet()),
id='MIT_Sbay_c342_13338', name='MIT_Sbay_c342_13338',
description='MIT_Sbay_c342_13338', dbxrefs=[])
>>> s3 = u2.next()
>>> s4 = u2.next()
>>> s5 = u2.next()
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
/Users/pmagwene/Desktop/tmp/<ipython console> in <module>()
StopIteration:
```

In this case the `SeqIO.parse` function returns an object that has *iterator* semantics (technically it's a 'generator' but this is a technical difference that you can ignore for now). An iterator is an object that 'acts like' a sequence (e.g. a list or tuple), but there are some major differences. The most important one is that an iterator does not have to compute the entire sequence at once. In the case of the `SeqIO.parse()` function that

means that if you have a FASTA file with thousands of sequence entries it wouldn't try to suck them all into memory. The `.next()` method is used to call successive sequence entries in the FASTA file. When you call `.next()` on the iterator(generator) instance you get back `SeqRecords`, one at a time. However, as the last call demonstrates if there is no 'next' item in the iterator it raises a `StopIteration` exception. For more info about iterators and generators see Norman Matloff's Tutorial on Python Iterators and Generators.

The steps for reading a FASTA sequence file can be wrapped up in the following function. We'll place each of the functions we develop in a module called `pipeline.py` (place this in your working directory or your `PYTHONPATH`). As you progress through the pipeline design you will add additional functions to this module.

```
# pipeline.py -- a simple bioinformatics pipeline
from Bio import SeqIO

def read_fasta(infile):
    """Read a single sequence from a FASTA file"""
    rec = SeqIO.read(infile,'fasta')
    return rec

def parse_fasta(infile):
    """Read multiple sequences from a FASTA file"""
    recs = SeqIO.parse(infile,'fasta')
    return [i for i in recs]
```

List comprehensions

The `parse_fasta()` function above introduces another new concept called *list comprehensions*. A list comprehension is a compact way of applying a function to each element in a sequence. In this case the list comprehension implicitly called `.next()` to get all the `SeqRecords` from the generator returned by `SeqIO.parse()`. You'll recall that most functions in R works in a vector-wise manner. List comprehensions provide similar semantics for Python. Below are some simpler examples of list comprehensions. Try and predict the output of each of these before typing them in:

```
In [1]: x = [2,4,6,8,10]
In [2]: [i**2 for i in x]
Out[2]: ???
In [3]: y = ['bob', 'tab', 'rob', 'snob']
In [4]: def juvenilize(s):
   ...:     return str(s) + "by"
   ...:
In [5]: [juvenilize(i) for i in y]
Out[5]: ???
```

You can use the `parse_fasta()` function as follows:

```
>>> import pipeline
>>> recs = pipeline.parse_fasta('unknown2.fas')
```

```
>>> len(recs)
4
>>> [i.name for i in recs]
['YJL005W', 'MIT_Sbay_c342_13338', 'MIT_Smik_c333_12160', '
    MIT_Spar_c300_12282']
```

Note that the `parse_fasta()` function will return a list of `SeqRecords` even when there is only a single sequence in the file. In contrast, if you use the function `read_fasta()` on a FASTA file with more than one sequence it will raise an error.

13.4.3 Translating nucleotide sequence to a protein sequence

The next step is to translate each DNA sequence into a corresponding protein sequence. This is very easy using the `.translate()` method associated with the `Seq` class.

```
>>> recs[0].seq.translate()
Seq('MSSKPDTGSEISGPQRQEEQEQQIEQSSPTEANDRSIHDEV
PKVKKRHEQNSGH...ST*', HasStopCodon(ExtendedIUPACProtein(), '*'))
```

Note that the above code returns an object of type `Seq`. That's usually what we want if we're manipulating nucleotide or protein sequences but if we want to write our translated sequences back out into a file we need to create new `SeqRecords`. I illustrate this in the function below (add this to `pipeline.py`).

```
from Bio import Seq
from Bio import SeqRecord

def translate_recs(seqrecs):
    """ nucleotide SeqRecords -> translated protein SeqRecords """
    proteins = []
    for rec in seqrecs:
        aaseq = rec.seq.translate()
        protrec = SeqRecord.SeqRecord(aaseq, id=rec.id, name=rec.name,
                                      description=rec.description)
        proteins.append(protrec)
    return proteins
```

We can then encapsulate the whole process of converting a nucleotide FASTA file to a peptide sequence FASTA file as so (add these to `pipeline.py`):

```
def translate_fasta(infile, outfile):
    """ nucleotide fasta file -> protein fasta file """
    nrecs = parse_fasta(infile)
    precs = translate_recs(nrecs)
    SeqIO.write(precs, outfile, 'fasta')
```

We can use our `translate_fasta` function from the Python interpreter like so:

```
>>> reload(pipeline)
<module 'pipeline' from '/Users/pmagwene/synchronized/pyth/pipeline.py'>
```

```
>>> pipeline.translate_fasta('unknown2.fas', 'unknown2-protein.fasta')
```

Take a moment to open the file `unknown2-protein.fasta` in a text editor to confirm that the file now hold amino acid sequences rather than nucleotide sequences.

In-class Assignment 13.1

Since DNA is a double-stranded molecule, protein coding information can be coded on either strand (referred to as the Watson and Crick strands, see Cartwright and Graur 2011, Biology Direct 6:7 for a discussion of the history of this terminology and how it's defined). Sequence data in genome databases such as GenBank are typically provided in their Watson strand encoding, so that if you are dealing with a coding gene of interest whose 5' to 3' encoding is on the Crick strand you will need to take the reverse complement of the sequence before translating it.

1. Using the Biopython documentation, figure out how to get the reverse complement of a sequence, and then write a Python function `reverse_complement_fasta` that takes as its input a file of nucleotide FASTA records (`infile`), calculates their reverse complements, and write those reverse complements out to the argument `outfile`. This should parallel the function `translate_fasta` given above.
2. Apply your function the the file `unknown3.fas` from the course website, producing the file `unknown3-revcomp.fas`.
3. Apply the `translate_fasta` function to `unknown3-revcomp.fas` producing the protein sequence file `unknown3-protein.fas`.

13.4.4 Turning a Python module into a command line program

A Python module can server both as a library of functions, as a well as a command line program that can be called like other Unix programs. The easiest way to do this is to use the argparse module, which is part of the Python standard library.

The following snippet shows how we can make our `pipeline.py` act as a command line program:

```
#!/usr/bin/env python

from Bio import SeqIO, Seq, SeqRecord

def parse_fasta(infile):
    recs = SeqIO.parse(infile, 'fasta')
    return [i for i in recs]

def translate_recs(seqrecs):
    proteins = []
    for rec in seqrecs:
        aaseq = rec.seq.translate()
        protrec = SeqRecord.SeqRecord(aaseq, id=rec.id, name=rec.name,
                                      description=rec.description)
        proteins.append(protrec)
```

```
    return proteins

def translate_fasta(infile, outfile):
    nrecs = parse_fasta(infile)
    precs = translate_recs(nrecs)
    SeqIO.write(precs, outfile, 'fasta')

def reverse_complement_fasta(infile, outfile):
    pass  # replace with your reverse_complement_fasta fxn


import sys
import argparse
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('command', choices=['translate', 'revcomp'], help='
        The command to run on input to the program')

    parser.add_argument('-i','--infile', nargs='?', type=argparse.FileType(
        'r'), default=sys.stdin)
    parser.add_argument('-o', '--outfile',nargs='?', type=argparse.FileType
        ('w'), default=sys.stdout)

    args = parser.parse_args()

    if args.command == 'translate':
        translate_fasta(args.infile, args.outfile)

    elif args.command == 'revcomp':
        reverse_complement_fasta(args.infile, args.outfile)
```

At the very top of our module we've added a line `#!/usr/bin/env python`. This instructs the shell to run this program with Python (`/usr/bin/env` figures out where Python is installed on your system).

At the end of our module we've imported a couple more modules, and then we have the statement `if __name__ == '__main__'`. This if statement simply says, "execute the following code only if this module is being run as a program" (as opposed to imported as a module). Nested within this if statement is the setup and logic associated with `argparse`. First we create and `ArgumentParser` object called `parser`, and then we add a number of arguments to the parser. Our first argument, `` `command` `` currently takes one of two possible choices, depending if we want our program to translate nucleotide sequences or take their reverse complement. The next two arguments specificy infile and outfile arguments, that can be called with either short names (`-i`/`-o`) or long names (`--infile`/`--outfile`). If file names aren't specified, the default behavior is to follow the Unix norm of taking input from `STDIN` and writing output to `STDOUT`.

The make this script executable in a Unix environment we need to set the executable bit by using the `chmod` command from the bash prompt as follows:

```
$ chmod +x ./pipeline.py
```

If you don't have any errors in your program, you should now be able to call it as so (assuming you're in the same directory as `pipeline.py`). First, let's get some help on using our function:

```
$ ./pipeline.py --help
```

By using `argparse` we get automatically generate a short description of the available arguments!

To use the program to translate a set of records:

```
$ ./pipeline.py -i unknown2.fas -o unknown2-proteins.fas translate
```

Because our program can read and write to `STDIN` and `STDOUT` we can combine multiple calls to `pipeline.py` with pipes and other Unix utilities. Here for example we first take the reverse complement and then translate `unknown3.fas`:

```
$ cat unknown3.fas | ./pipeline.py revcomp | ./pipeline.py translate
```

For more details and explanation about how `argparse` works see the argparse documentation.

13.4.5 Globbing to get multiple files of a given type

As an aside, what if we wanted to repeat this for a whole directory full of DNA sequences in separate FASTA files? Here's a function to help accomplish that task:

```
import glob

def inout_pairs(insuffix, outsuffix):
    """ Files in directory with given suffix -> list of tuples w/ (infile,
        outfile)"""
    infiles = glob.glob('*'+insuffix)
    pairs = []
    for infile in infiles:
        inprefix = infile[:-len(insuffix)]
        outfile = inprefix + outsuffix
        pairs.append((infile,outfile))
    return pairs
```

The `glob` module gives you filename 'globbing' functionality. Globbing is a means of matching specified file or pathnames; you can think about this as a simplified class of regular expressions. For example, you're probably familiar with command line searches like:

```
$ ls *.fas   # list all files with the extension .fas
$ ls unk*   # list all files that begin with 'unk'
```

The `inout_pairs()` function we defined above allows us to glob file files with the given `insuffix` and create a corresponding set of names for output files. The following illustrates this:

```
>>> pairs = pipeline.inout_pairs('.fas', '-protein.fasta')
>>> pairs
[('unknown1.fas', 'unknown1-protein.fasta'),
('unknown2.fas', 'unknown2-protein.fasta')]
>>> from Bio.Data.CodonTable import TranslationError
>>> for (i,o) in pairs:
...     try:
...         pipeline.translate_fasta(i,o)
...     except TranslationError:
...         continue
...
...
>>> ls *.fas*  # only works in ipython
unknown1-protein.fasta  unknown1.fas  unknown2-protein.fasta  unknown2.fas
```

Note that I changed the file suffix from `.fas` to `.fasta` on the output files. This isn't necessary but I find that doing so makes it easy to sort through large directories to distinguish generated files from the original files. The `inout_pairs()` function will come in handy when we combine our functions to generate a multi-sequence pipeline.

Another new concept I introduced in the for loop above is `try-except` block for exception handling. The Python starts by executing the code in the `try` clause. If there are no problems the `except` clause is ignored. However, if an exception (error) is raised than it evaluates the `except` clause. In this case, our `except` clause says if the error is an exception of type `TranslationError` (defined in `Bio.Data.CodonTable`) then ignore it and just keep working. However, any other exception will stop program execution, as we haven't included any general error handling code. See Downey, Chap 14 for more discussion of exception handling.

13.4.6 BLAST searches via the NCBI server

We can use Biopython do network based BLAST searches. Here we will use blastp to search against protein sequences in the Swiss-Prot database.

```
>>> from Bio.Blast import NCBIWWW, NCBIXML
>>> prot1 = pipeline.read_fasta('unknown1-protein.fasta')
>>> results_handle = NCBIWWW.qblast('blastp','swissprot',prot1.seq.tostring
    (), entrez_query='(Homo sapiens[ORGN])')
>>> results = results_handle.read()
>>> sfile = open('prot1_blast.out','w')
>>> sfile.write(results)
>>> sfile.close()
>>> blast_out = open('prot1_blast.out','r')
>>> brec = NCBIXML.read(blast_out)
>>> brec
<Bio.Blast.Record.Blast instance at 0x2ec22d8>
>>> len(brec.alignments) # we got 50 blast hits in the query
50
>>> brec.alignments[0]
<Bio.Blast.Record.Alignment instance at 0x2ec23a0>
```

```
>>> brec.alignments[0].accession
u'P31749'
```

This code introduces another concept we'll call the *Producer-Consumer* pattern. The Producer-Consumer pattern is a general programming concept, but the key here is that the pattern generalizes the problem of parsing complex biological data types. The producer does the work of getting the information from a file (or from the web in this case). The consumer process the information into a form we can use. In the code above the function `NCBIWWW.qblast()` is the producer and `NCBIXML.read()` plays the role of the consumer. This pattern is used over and over again in Biopython so you should spend some time trying to understand the general idea. See the Biopython tutorial for a more complete discussion.

Our BLAST query returned the information in the form of XML data. XML stands for 'Extensible Markup Language', and is a generic way to encode documents in machine-readable form. XML data is usually plain text – go ahead and open up the file `prot1_blast.out` in a text editor to see the output. Since XML is a generic format, specific types of XML documents need a 'schema' or 'grammar' that specifies how the document is to be read and interpretted. In the example above, the module `NCBIXML` knows how to handle XML data returned from NCBI, hence our use of the function `NCBIXML.read()`.

In the example given, we limited our query to sequences from humans. If we wanted to include all metazoan sequences we could pass `'(Metazoa[ORGN])'` as the argument to `entrez_query`. If we didn't want to limit our search at all we would simply not include that argument (i.e. accept the default). The BLAST output is fairly complicated. See the BioPython tutorial section 7.5 for a complete breakdown of all the fields in the BLAST output.

Again, the commands above are rather involved so let's wrap them up in a function:

```
from Bio.Blast import NCBIWWW, NCBIXML

def blastp(seqrec, outfile, database='nr', entrez_query='(none)'):
    handle = NCBIWWW.qblast('blastp', database, seqrec.seq.tostring(),
                                entrez_query=entrez_query)
    results = handle.read()
    sfile = open(outfile, 'w')
    sfile.write(results)
    sfile.close()
    bout = open(outfile, 'r')
    brecord = NCBIXML.read(bout)
    return brecord

def summarize_blastoutput(brecord):
    hits = []
    for alignment in brecord.alignments:
        expect = alignment.hsps[0].expect
        accession = alignment.accession
        hits.append((expect,accession))
    hits.sort() # will sort tuples by their first value (i.e. expect)
    return hits
```

We can use this code as follows:

```
>>> humanblast = pipeline.blastp(prot1, 'prot1-hum-blast.out', database='
    swissprot', entrez_query='(Homo sapiens[ORGN])')
>>> flyblast = pipeline.blastp(prot1, 'prot1-fly-blast.out', database='
    swissprot', entrez_query='(Drosophila melanogaster[ORGN])')
>>> humanhits = pipeline.summarize_blastoutput(humanblast)
>>> flyhits = pipeline.summarize_blastoutput(flyblast)
>>> humanhits[0] # the first number is the E-value for the BLAST search
(4.98013e-95, u'P31749')
>>> print humanhits[0][1]  # prints the swissprot accession number
P31749
>>> flyhits[0]
(4.09304e-96, u'Q8INB9')
```

Go to the UniProt website and use the search box to lookup those accession numbers.

13.4.7 Getting records from Swiss-Prot

For a small number of accession numbers it's easy to use the web interface to UniProt (Swiss-Prot). For hundred of blast hits that's just not an option. Conveniently, we can use Biopython to query the Swiss-Prot database to retrieve information about these presumed orthologs. You can access the Swiss-Prot database as follows:

```
>>> from Bio import ExPASy
>>> from Bio import SwissProt
>>> handle1 = ExPASy.get_sprot_raw(humanhits[0][1]) # access with the
    accession number
>>> rec1 = SwissProt.read(handle1)
>>> print rec1.description
RecName: Full=RAC-alpha serine/threonine-protein kinase; EC=2.7.11.1;
    AltName:
Full=RAC-PK-alpha; AltName: Full=Protein kinase B; Short=PKB; AltName: Full
    =C-
AKT;
>>> rec1.comments[0]
"FUNCTION: AKT1 is one of 3 closely related serine/threonine- protein
    kinases (AKT1, AKT2 and AKT3) called the AKT kinase, and which regulate
     many processes including metabolism, proliferation, cell survival,
    growth and angiogenesis.
... output truncated ..."
>>> print dir(rec1) # lets see what other attributes the record has
['__doc__', '__init__', '__module__', 'accessions', 'annotation_update', '
    comments', 'created', 'cross_references', 'data_class', 'description',
    'entry_name', 'features', 'gene_name', 'host_organism', '
    host_taxonomy_id', 'keywords', 'molecule_type', 'organelle', 'organism'
    , 'organism_classification', 'references', 'seqinfo', 'sequence', '
    sequence_length', 'sequence_update', 'taxonomy_id']
>>> print rec1.gene_name
```

```
Name=AKT1; Synonyms=PKB, RAC;
>>> print rec1.sequence[:25] # first 25 amino acids
MSDVAIVKEGWLHKRGEYIKTWRPR
```

Here's some functions to make this more convenient:

```
from Bio import ExPASy
from Bio import SwissProt

def get_swissrec(accession):
    handle = ExPASy.get_sprot_raw(accession)
    record = SwissProt.read(handle)
    return record

def swissrec2seqrec(record):
    seq = Seq.Seq(record.sequence, Seq.IUPAC.protein)
    s = SeqRecord.SeqRecord(seq, description=record.description,
                id=record.accessions[0], name=record.entry_name)
    return s
```

And here is an example of how we can apply these functions:

```
>>> ids = [humanhits[0][1], flyhits[0][1]]
>>> ids
[u'P31749', u'Q8INB9']
>>> swissrecs = [pipeline.get_swissrec(i) for i in ids]
>>> seqs = [pipeline.swissrec2seqrec(i) for i in swissrecs]
>>> seqs[0]
SeqRecord(seq=Seq('MSDVAIVKEGWLHKRGEYIKTWRPRYFLLKNDGTFIGYKERPQDVDQREAPLNN
    ...GTA', IUPACProtein()), id='P31749', name='AKT1_HUMAN', description='
    RecName: Full=RAC-alpha serine/threonine-protein kinase; EC=2.7.11.1;
    AltName: Full=Protein kinase B; Short=PKB; AltName: Full=Protein kinase
     B alpha; Short=PKB alpha; AltName: Full=Proto-oncogene c-Akt; AltName:
     Full=RAC-PK-alpha;', dbxrefs=[])
>>> seqs[1]
SeqRecord(seq=Seq('MNYLPFVLQRRSTVVASAPAPGSASRIPESPTTTGSNIINIIYSQSTHPNSSPT
    ...SMQ', IUPACProtein()), id='Q8INB9', name='AKT1_DROME', description='
    RecName: Full=RAC serine/threonine-protein kinase; Short=DAkt; Short=
    DRAC-PK; Short=Dakt1; EC=2.7.11.1; AltName: Full=Akt; AltName: Full=
    Protein kinase B; Short=PKB;', dbxrefs=[])
>>> seqs.append(prot1)  # add our original protein sequence to the list
>>> pipeline.write_fasta(seqs, 'unknown1-plus-human-fly.fasta')
```

In-class Assignment 13.2

Apply the BLAST and Swiss-Prot code above to the protein sequence you generated from `unknown3.fas`, searching against the genomes of H. sapiens, D. melanogaster, and E. coli. Based on the results of your search, what is the function of this protein?

13.4.8 Multiple sequence alignment via MAFFT

We've now generated a new FASTA file that includes our original protein sequence and the sequences for the human and fly BLAST best hits. We will use MAFFT to perform a multiple alignment. Biopython has built in code to simplify command line usage of common alignment programs like CLUSTALW, MAFFT, and MUSCLE. However I'll show you how to do this with your own code using the `subprocess` module. Knowing how the `subprocess` module works is useful because it allows you to interface with any command line program from within Python.

The `subprocess` module allows your Python code to start other programs (child processes) and send/get input and output from those same processes. When we use the subprocess module we're putting the Unix design element of 'Everything is a file or process' to use. Here's a simple example:

```
>>> import subprocess
>>> subprocess.call(["ls","-l"])
# on windows the equivalent command is
# subprocess.call(["dir",],shell=True)
# output is NOT shown in ipython notebook, instead
# a return code (0 if the command worked) is shown
total 11696
-rw-r--r--   1 pmagwene  staff    93514 Nov 22 19:36 prot1-fly-blast.out
-rw-r--r--   1 pmagwene  staff   109635 Nov 22 19:35 prot1-hum-blast.out
-rw-r--r--   1 pmagwene  staff   109635 Nov 22 19:19 prot1_blast.out
-rw-r--r--   1 pmagwene  staff     2308 Nov 22 20:07 unknown1-plus-human-
    fly.fasta
-rw-r--r--   1 pmagwene  staff      854 Nov 22 16:46 unknown1-protein.fasta
-rwx------   1 pmagwene  staff     2535 Nov 22 15:38 unknown1.fas
-rw-r--r--@  1 pmagwene  staff    24849 Nov 22 16:25 unknown2.fas
-rw-r--r--   1 pmagwene  staff     8331 Nov 22 16:46 unknowns-protein.fasta
```

The above code uses a convenience function `call()` in the `subprocess` module. We'll use the same function to run MAFFT:

```
import subprocess

def mafft_align(infile, outfile):
    ofile = open(outfile,'w')
    retcode = subprocess.call(["mafft",infile], stdout=ofile)
    ofile.close()
    if retcode != 0:
        raise Exception("Possible error in MAFFT alignment")
```

And we put it to use as follows:

```
In [8]: reload(pipeline)
In [8]: pipeline.mafft_align('unknown1-plus-human-fly.fasta', 'unknown1-
    alignment.fasta')
```

If all went well this should have created the file `unknown1-alignment.fasta` in your directory. Open this alignment using JalView to examine the alignment in more detail.

13.4.9 Searching for protein domains using HMMER and Pfam

As the final step of our pipeline we'll use HMMER and the Pfam database to search for known protein domains in our original protein. This assumes you have the HMMER binaries and Pfam database installed as demonstrated in last weeks exercises and that you've already run `hmmpress` against the Pfam database. Again we write a small wrapper function using the `subprocess` module. This time we'll use the `Popen` class to illustrate how we can capture the output produced by `hmmpfam`. Note that if you haven't installed the HMMER binaries to one of the standard locations you might need to specify the full path to the `hmmscan` executable in the code below.

```
def hmmer_pfam(infilename, outfilename, pfamdb):
    pipe = subprocess.Popen(["hmmscan", pfamdb, infilename],
            stdout=subprocess.PIPE).stdout
    output = pipe.read() # this gives us the output of our command
    outfile = open(outfilename, 'w')
    outfile.write(output)
    outfile.close()
```

This function can be called like this:

```
# change the last argument to match the path to your Pfam database.
>>> pipeline.hmmer_pfam('unknown1-protein.fasta', 'unknown1-domains.out', '
    /Users/pmagwene/tmp/Pfam-A.hmm')
```

As before this search may take several minutes.

13.4.10 Putting it all together

We've generated a variety of functions that take care of the major steps of our pipeline. It's time to put the steps together to automate the entire process.

```
def oneseq_pipeline(infilename, pfamdb=None,
                    compareto=['Homo sapiens','Drosophila melanogaster'],
                    skipHMMER = True,extension="XX"):
    # translate nucleotide sequence to protein seq
    protout = 'protein-' + infilename + extension
                # add the extension so all generated files have
                # different extension than input files

    translate_fasta(infilename, protout)

    # run blastp on protein sequence against swissprot and extract best
        hits
    protrec = parse_fasta(protout)[0]
    blastout ='blast-' + protout
    besthitids = []
    for organism in compareto:
        equery = '(%s[ORGN])' % organism # create the entrez organism query
        brecord = blastp(protrec, blastout, database='swissprot',
            entrez_query=equery)
```

```
        bhits = summarize_blastoutput(brecord)
        besthitids.append(bhits[0][1])

    # download corresponding records from Swiss-Prot
    swissrecs = [get_swissrec(i) for i in besthitids]
    seqs = [swissrec2seqrec(i) for i in swissrecs]
    seqs.append(protrec)

    # write FASTA file with best hits plus original protein sequence
    plusout = 'blasthits-' + protout + '.XML'
    write_fasta(seqs, plusout)

    # do multiple alignment via mafft
    mafft_align(plusout, 'aligned-' + protout)

    # search for domains via HMMER/Pfam
    if not skipHMMER:
        if pfamdb is not None:
            hmmerout = 'hmmer-' + protout
            hmmer_pfam(protout, hmmerout, pfamdb)
```

Our function can take as input a FASTA file with a single sequence or with multiple sequences. In the case of a multiple sequences it assumes that the 'target' sequence for the search is the first sequence in the file. Also, note the `skipHMMER` argument included in the function. The HMMER search takes a relatively long time and doing it sequence by sequence is not very efficient so by default the pipeline will skip this step. If you want to include the HMMER step than specify the Pfam database file and set `skipHMMER=False`.

Testing out the pipeline

To test out the function we do:

```
>>> reload(pipeline)
>>> pipeline.oneseq_pipeline('unknown1.fas')
```

This will create four new FASTA files:

1) `protein-unknown1.fasXX`
2) `blast-protein-unknown1.fasXX.XML`
3) `blasthits-protein-unknown1.fasXX`
4) `aligned-protein-unknown1.fasXX`

These respectively contain:

1) the amino acid sequence translated from the nucleotide sequence given as input
2) the XML output of the qblast query to NCBI
3) the amino acid sequences for the BLAST hits returned from NCBI
4) the MAFFT multiple alignment of the protein sequences.

Let's now test the pipeline using an alternate set organisms:

```
>>> pipeline.oneseq_pipeline('unknown1.fas', compareto=["Homo sapiens","Mus
     musculus","Caenorhabditis elegans"])
```

For completeness let's also test the pipeline with the HMMER step included:

```
>>> pipeline.oneseq_pipeline('unknown1.fas', '/home/pmagwene/tmp/Pfam-A.hmm
    ',
skipHMMER=False)
```

Extending the pipeline to deal with multiple inputs

Now that we're confident out single sequence pipeline function works it can be easily adapted to deal with multiple input files:

```
def multiseq_pipeline(inext, pfamdb=None,
                compareto=['Homo sapiens','Drosophila melanogaster'],
                skipHMMER=True):
    inout = inout_pairs(inext, 'XX')
    infiles = [i[0] for i in inout]
    for filename in infiles:
        print "Processing %s" % filename
        oneseq_pipeline(filename, pfamdb, compareto, skipHMMER)
```

To test the complete multi-sequence pipeline delete all the generated files (so that only `unknown1.fas` and `unknown2.fas` are in the unknowns directory) and try the following:

```
>>> pipeline.multiseq_pipeline('.fas')
```

Given our example data this function will process just two input files. However, you can add an arbitrary number of additional '.fas' files to the directory and the pipeline will process those as well with exactly the same command.

There are a number of ways the pipeline could be sped up. One obvious improvement would be to utilize a local installation of BLAST and the respective databases. However, optimization is often a complex task. The pipeline we developed here doesn't require us to install BLAST (which can be somewhat involved) and provides adequate performance for a modest number of sequences. It is possible to turn this set of Python functions into a program that you could run from the command line (rather than the Python interpeter) just like any other Unix program.

13.5 The pipeline.py module

The pages that follow give the complete code listing for the `pipeline.py` module.

```
"""
pipline.py --  An illustrative example of a bioinformatics pipeline.
Requires Python 2.6+ and BioPython 1.53+
(c) Copyright by Paul M. Magwene, 2009-2014  (mailto:paul.magwene@duke.edu)
"""

from Bio import Seq, SeqIO, SeqRecord
from Bio import ExPASy, SwissProt
from Bio.Blast import NCBIWWW, NCBIXML

import glob, subprocess

def read_fasta(infile):
    """Read a single sequence from a FASTA file"""
    rec = SeqIO.read(infile,'fasta')
    return rec

def parse_fasta(infile):
    """Read multiple sequences from a FASTA file"""
    recs = SeqIO.parse(infile,'fasta')
    return [i for i in recs]

def write_fasta(recs, outfile):
    SeqIO.write(recs, outfile, 'fasta')

def translate_recs(seqrecs):
    """ nucleotide SeqRecords -> translated protein SeqRecords """
    proteins = []
    for rec in seqrecs:
        aaseq = rec.seq.translate()
        protrec = SeqRecord.SeqRecord(aaseq, id=rec.id, name=rec.name,
                                      description=rec.description)
        proteins.append(protrec)
    return proteins

def translate_fasta(infile, outfile):
    """ nucleotide fasta file -> protein fasta file """
    nrecs = parse_fasta(infile)
    precs = translate_recs(nrecs)
    write_fasta(precs, outfile)

def inout_pairs(insuffix, outsuffix):
    """ Files in directory with given suffix -> list of tuples w/ (infile,
        outfile)"""
    infiles = glob.glob('*'+insuffix)
    pairs = []
    for infile in infiles:
        inprefix = infile[:-len(insuffix)]
        outfile = inprefix + outsuffix
```

```
        pairs.append((infile,outfile))
    return pairs

def blastp(seqrec, outfile, database='nr', entrez_query='(none)'):
    handle = NCBIWWW.qblast('blastp', database, seqrec.seq.tostring(),
                                entrez_query=entrez_query)
    results = handle.read()
    sfile = open(outfile, 'w')
    sfile.write(results)
    sfile.close()
    bout = open(outfile, 'r')
    brecord = NCBIXML.read(bout)
    return brecord

def summarize_blastoutput(brecord):
    hits = []
    for alignment in brecord.alignments:
        expect = alignment.hsps[0].expect
        accession = alignment.accession
        hits.append((expect,accession))
    hits.sort() # will sort tuples by their first value (i.e. expect)
    return hits

def get_swissrec(accession):
    handle = ExPASy.get_sprot_raw(accession)
    record = SwissProt.read(handle)
    return record

def swissrec2seqrec(record):
    seq = Seq.Seq(record.sequence, Seq.IUPAC.protein)
    s = SeqRecord.SeqRecord(seq, description=record.description,
                id=record.accessions[0], name=record.entry_name)
    return s

def mafft_align(infile, outfile):
    ofile = open(outfile,'w')
    retcode = subprocess.call(["mafft",infile], stdout=ofile)
    ofile.close()
    if retcode != 0:
        raise Exception("Possible error in MAFFT alignment")

def hmmer_pfam(infilename, outfilename, pfamdb):
    pipe = subprocess.Popen(["hmmscan", pfamdb, infilename],
            stdout=subprocess.PIPE).stdout
    output = pipe.read() # this gives us the output of our command
    outfile = open(outfilename, 'w')
    outfile.write(output)
    outfile.close()
```

```
def oneseq_pipeline(infilename, pfamdb=None,
                    compareto=['Homo sapiens','Drosophila melanogaster'],
                    skipHMMER = True,extension="XX"):
    # translate nucleotide sequence to protein seq
    protout = 'protein-' + infilename + extension
                # add the extension so all generated files have
                # different extension than input files

    translate_fasta(infilename, protout)

    # run blastp on protein sequence against swissprot and extract best
        hits
    protrec = parse_fasta(protout)[0]
    blastout ='blast-' + protout + '.XML'
    besthitids = []
    for organism in compareto:
        equery = '(%s[ORGN])' % organism # create the entrez organism query
        brecord = blastp(protrec, blastout, database='swissprot',
            entrez_query=equery)
        bhits = summarize_blastoutput(brecord)
        besthitids.append(bhits[0][1])

    # download corresponding records from Swiss-Prot
    swissrecs = [get_swissrec(i) for i in besthitids]
    seqs = [swissrec2seqrec(i) for i in swissrecs]
    seqs.append(protrec)

    # write Fasta file with best hits plus original protein sequence
    plusout = 'blasthits-' + protout
    write_fasta(seqs, plusout)

    # do multiple alignment via mafft
    mafft_align(plusout, 'aligned-' + protout)

    # search for domains via HMMER/Pfam
    if not skipHMMER:
        if pfamdb is not None:
            hmmerout = 'hmmer-' + protout
            hmmer_pfam(protout, hmmerout, pfamdb)

def multiseq_pipeline(inext, pfamdb=None,
                compareto=['Homo sapiens','Drosophila melanogaster'],
                skipHMMER=True):
    inout = inout_pairs(inext, 'XX')
    infiles = [i[0] for i in inout]
    for filename in infiles:
        print "Processing %s" % filename
        oneseq_pipeline(filename, pfamdb, compareto, skipHMMER)
```