Biopython

What is biopython?

Biopython is a collection of python modules that contain code for manipulating biological data. Many handle sequence data and common analysis and processing of the data including reading and writing all common file formats. Biopython will also run blast for you and parse the output into objects inside your script. This requires just a few lines of code.

Installing Biopython

This is very straightforward once you have anaconda or minconda installed. I use miniconda because it's smaller. We are going to use `sudo` because this will give us permission to install in the 'correct' directory python is expecting to find the modules. Other users will be able to use it too. Using `sudo` can cause problems, but it's ok here. You will need the administrator password for the machine. If you don't have this, ask the person who does administration on your machine.

```
% sudo conda install biopython
WARNING: Improper use of the sudo command could lead to data loss
or the deletion of important system files. Please double-check your
typing when using sudo. Type "man sudo" for more information.

To proceed, enter your password, or type Ctrl-C to abort.

Password:
Fetching package metadata ...........
Solving package specifications: .

Package plan for installation in environment /anaconda3:

The following NEW packages will be INSTALLED:

    biopython: 1.69-np113py36_0

The following packages will be UPDATED:

    conda:     4.3.29-py36hbf39572_0 anaconda --> 4.3.30-py36h173c244_0

The following packages will be SUPERSEDED by a higher-priority channel:

    conda-env: 2.6.0-h36134e3_0      anaconda --> 2.6.0-h36134e3_0

Proceed ([y]/n)?

conda-env-2.6. 100% |###################################################| Time: 0:00:00   4.02 MB/s
biopython-1.69 100% |###################################################| Time: 0:00:00  21.02 MB/s
conda-4.3.30-p 100% |###################################################| Time: 0:00:00  43.61 MB/s
```

See if the install worked

```
python3
>>> import Bio
>>> print(Bio.__version__)
1.69
```

If we get no errors, biopython is installed correctly.

Biopython documentation

Biopython wiki page

http://biopython.org/

Getting started

http://biopython.org/wiki/Category%3AWiki_Documentation

Biopython tutorial

ttp://biopython.org/DIST/docs/tutorial/Tutorial.html#chapter:Bio.SeqIO

Complete tree of Biopython Classes

http://biopython.org/DIST/docs/api/Bio-module.html

Working with DNA and protein sequences

This is the core of biopython. And uses the Seq object. Seq is part of Bio. This is denoted Bio.Seq

```
#!/usr/bin/env python3
import Bio.Seq                              # }
seqobj = Bio.Seq.Seq('ATGCGATCGAGC')        # } that's a lot of Seqs
# convert to string with str(seqobj)
seq_str = str(seqobj)
print('{:s} has {:d} nucleotides'.format( seq_str , len(seq_str)))
```

produces

```
ATGCGATCGAGC has 12 nucleotides
```

From ... import ...

Another way to import modules is with `from ... import ...` . This saves typing the Class name every time. Bio.Seq is the class name. Bio is the superclass. Seq is a subclass inside Bio. It's written Bio.Seq. Seq has several different subclasses, of which one is called Seq. So we have Bio.Seq.Seq. To make the creation simpler, we call Seq() after we import with `from ... import ...` like this

```
#!/usr/bin/env python3
from Bio.Seq import Seq
seqobj=Seq('ATGCGATCGAGC')
seq_str=str(seqobj)
protein = seqobj.translate()
prot_str = str(protein)
print('{:s} translates to {:s}'.format(seq_str,prot_str))
```

produces

```
ATGCGATCGAGC
```

Bio.Alphabets

A Seq object likes to know what alphabet it uses A,C,G,T for DNAAlpabet etc. Not essential for most uses, but prevents you trying to translate a protein sequence!

Specific Alphabets

```
>>> seqobj
Seq('ATG', Alphabet())
>>> from Bio.Alphabet import DNAAlphabet
>>> so=Seq('ATG',DNAAlphabet())
>>> so
Seq('ATG', DNAAlphabet())
>>> so.translate()
Seq('M', ExtendedIUPACProtein())

```

For proteins

```
from Bio.Alphabet import ProteinAlphabet
seqobj = Seq('MGT', ProteinAlphabet())
```

Generic Alphabets

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna, generic_protein
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq
Seq('AGTACACTGGT', Alphabet())
>>> my_dna = Seq("AGTACACTGGT", generic_dna)
>>> my_dna
Seq('AGTACACTGGT', DNAAlphabet())
>>> my_protein = Seq("AGTACACTGGT", generic_protein)
>>> my_protein
Seq('AGTACACTGGT', ProteinAlphabet())
```

Extracting a subsequence

You can use a range [0:3] to get the first codon

`seqobj[0:3]`

Read a FASTA file

We were learning how to read a fasta file line by line. SeqIO.parse() is the main method for reading from almost any file format. We'll need a fasta file. We can use Python_05.fasta which looks like this

```
>seq1
AAGAGCAGCTCGCGCTAATGTGATAGATGGCGGTAAAGTAAATGTCCTATGGGCCACCAATTATGGTGTATGAGTGAATCTCTGGTCCGAGATTCA
CTGAGTAACTGCTGTACACAGTAGTAACACGTGGAGATCCCATAAGCTTCACGTGTGGTCCAATAAAACACTCCGTTGGTCAAC
>seq2
GCCACAGAGCCTAGGACCCCAACCTAACCTAACCTAACCTAACCTACAGTTTGATCTTAACCATGAGGCTGAGAAGCGATGTCCTGACCGGCCTGT
CCTAACCGCCCTGACCTAACCGGCTTGACCTAACCGCCCTGACCTAACCAGGCTAACCTAACCAAACCGTGAAAAAAGGAATCT
>seq3
ATGAAAGTTACATAAAGACTATTCGATGCATAAATAGTTCAGTTTTGAAAACTTACATTTTGTTAAAGTCAGGTACTTGTGTATAATATCAACTAA
AT
>seq4
ATGCTAACCAAAGTTTCAGTTCGGACGTGTCGATGAGCGACGCTCAAAAAGGAAACAACATGCCAAATAGAAACGATCAATTCGGCGATGGAAATC
AGAACAACGATCAGTTTGGAAATCAAAATAGAAATAACGGGAACGATCAGTTTAATAACATGATGCAGAATAAAGGGAATAATCAATTTAATCCAG
GTAATCAGAACAGAGGT
```

Get help on the parse() method with

```
>>> help(SeqIO.parse)

Help on function parse in module Bio.SeqIO:

parse(handle, format, alphabet=None)
    Turns a sequence file into an iterator returning SeqRecords.

        - handle   - handle to the file, or the filename as a string
          (note older versions of Biopython only took a handle).
        - format   - lower case string describing the file format.
        - alphabet - optional Alphabet object, useful when the sequence type
          cannot be automatically inferred from the file itself
          (e.g. format="fasta" or "tab")

    Typical usage, opening a file to read in, and looping over the record(s):

    >>> from Bio import SeqIO
    >>> filename = "Fasta/sweetpea.nu"
    >>> for record in SeqIO.parse(filename, "fasta"):
    ...     print("ID %s" % record.id)
    ...     print("Sequence length %i" % len(record))
    ...     print("Sequence alphabet %s" % record.seq.alphabet)
    ID gi|3176602|gb|U78617.1|LOU78617
    Sequence length 309
    Sequence alphabet SingleLetterAlphabet()

...
```

This uses the old `"..." % var` formatting syntax.

Here's a script to read fasta records and print out some information

```
#!/usr/bin/env python3
# assumes we are in the pfb2017 directory
from Bio import SeqIO
for seq_record in SeqIO.parse("./files/Python_05.fasta", "fasta"):   # give filename and format
  print('ID',seq_record.id)
  print('Sequence',str(seq_record.seq))
  print('Length',len(seq_record))

```

Convert fasta file to python dictionary in one line

There are three ways of doing this that use up more memory if you want more flexibility. `Bio.SeqIO.to_dict()` is the most flexible but also reads the entire fasta file into memory as a python dictionary so might take a lot of time and memory.

```
>>> id_dict = SeqIO.to_dict(SeqIO.parse('files/Python_05.fasta', 'fasta'))
>>> id_dict
{'seq1':
SeqRecord(seq=Seq('AAGAGCAGCTCGCGCTAATGTGATAGATGGCGGTAAAGTAAATGTCCTATGGGC...AAC',
SingleLetterAlphabet()), id='seq1', name='seq1', description='seq1', dbxrefs=[]),
'seq2':
SeqRecord(seq=Seq('GCCACAGAGCCTAGGACCCCAACCTAACCTAACCTAACCTAACCTACAGTTTGA...TCT',
SingleLetterAlphabet()), id='seq2', name='seq2', description='seq2', dbxrefs=[]),
'seq3':
SeqRecord(seq=Seq('ATGAAAGTTACATAAAGACTATTCGATGCATAAATAGTTCAGTTTTGAAAACTT...AAT',
SingleLetterAlphabet()), id='seq3', name='seq3', description='seq3', dbxrefs=[]),
'seq4':
SeqRecord(seq=Seq('ATGCTAACCAAAGTTTCAGTTCGGACGTGTCGATGAGCGACGCTCAAAAAGGAA...GGT',
SingleLetterAlphabet()), id='seq4', name='seq4', description='seq4', dbxrefs=[])}
```

Other methods set up a database or a way to read data as you need it.

Seq methods

```
seqobj.count("A")  # counts how many As are in sequence
seqobj.find("ATG") # find coordinate of ATG (-1 for not found)
```

SeqRecord objects

SeqIO.Parse generates Bio.SeqRecord.SeqRecord objects. These are annotated Bio.Seq.Seq objects.

Main attributes:

- id - Identifier such as a locus tag (string)
- seq - The sequence itself (Seq object or similar)

Additional attributes:

- name - Sequence name, e.g. gene name (string)
- description - Additional text (string)
- dbxrefs - List of database cross references (list of strings)
- features - Any (sub)features defined (list of SeqFeature objects)
- annotations - Further information about the whole sequence (dictionary). Most entries are strings, or lists of strings.
- letter_annotations - Per letter/symbol annotation (restricted dictionary). This holds Python sequences (lists, strings or tuples) whose length matches that of the sequence. A typical use would be to hold a list of integers representing sequencing quality scores, or a string

representing the secondary structure.

Prints this output

```
ID seq1
Sequence AAGAGCAGCTCGCGCTAATGTGATAGATGGCGGTAAAGTAAATGTCCTATGGGCCACCAATTATGGTGTATGAGTGAATCTCTGGTCCGAGATTCACTGAGTAACTGCTGTACACAGTAGTAACACGTGGAGATCCCATAAGCTTCACGTGTGGTCCAATAAAACACTCCGTTGGTCAAC
Length 180
ID seq2
Sequence GCCACAGAGCCTAGGACCCCAACCTAACCTAACCTAACCTAACCTACAGTTTGATCTTAACCATGAGGCTGAGAAGCGATGTCCTGACCGGCCTGTCCTAACCGCCCTGACCTAACCGGCTTGACCTAACCGCCCTGACCTAACCAGGCTAACCTAACCAAACCGTGAAAAAAGGAATCT
Length 180
ID seq3
Sequence ATGAAAGTTACATAAAGACTATTCGATGCATAAATAGTTCAGTTTTGAAAACTTACATTTTGTTAAAGTCAGGTACTTGTGTATAATATCAACTAAAT
Length 98
ID seq4
Sequence ATGCTAACCAAAGTTTCAGTTCGGACGTGTCGATGAGCGACGCTCAAAAAGGAAACAACATGCCAAATAGAAACGATCAATTCGGCGATGGAAATCAGAACAACGATCAGTTTGGAAATCAAAATAGAAATAACGGGAACGATCAGTTTAATAACATGATGCAGAATAAAGGGAATAATCAATTTAATCCAGGTAATCAGAACAGAGGT
Length 209
```

SeqRecord objects have .format() to convert to a string in various formats

```
>>> seq.format('fasta')
'>seq1\nAAGAGCAGCTCGCGCTAATGTGATAGATGGCGGTAAAGTAAATGTCCTATGGGCCACCAA\nTTATGGTGTATGAGTGAATCTCTGGTCCGAGATTCACTGAGTAACTGCTGTACACAGTAG\nTAACACGTGGAGATCCCATAAGCTTCACGTGTGGTCCAATAAAACACTCCGTTGGTCAAC\n'
```

Retrieving annotations from GenBank file

To read sequences from a genbank file instead, not much changes.

```python
#!/usr/bin/env python3
from Bio import SeqIO
for seq_record in SeqIO.parse("test_genome.gb", "genbank"):
  print('ID',seq_record.id)
  print('Sequence',str(seq_record.seq))
  print('Length',len(seq_record))
```

File Format Conversions

Many are straightforward, others are a little more complicated because the alphabet can't be determined from the data. It's usually easier to go from richer formats to simpler ones.

```python
#!/usr/bin/env python3
from Bio import SeqIO
records = SeqIO.parse("./files/Python_05.fasta", "fasta")   # give filename and format
count = SeqIO.write(records , './files/seqs.tab' , 'tab')
```

Produces

```
% more seqs.tab
seq1 AAGAGCAGCTCGCGCTAATGTGATAGATGGCGGTAAAGTAAATGTCCTATGGGCCACCAATTATGGTGTATGAGTGAATCTCTGGTCCGAGATTCACTGAGTAACTGCTGTACACAGTAGTAACACGTGGAGATCCCATAAGCTTCACGTGTGGTCCAATAAAACACTCCGTTGGTCAAC
seq2 GCCACAGAGCCTAGGACCCCAACCTAACCTAACCTAACCTAACCTACAGTTTGATCTTAACCATGAGGCTGAGAAGCGATGTCCTGACCGGCCTGTCCTAACCGCCCTGACCTAACCGGCTTGACCTAACCGCCCTGACCTAACCAGGCTAACCTAACCAAACCGTGAAAAAAGGAATCT
seq3 ATGAAAGTTACATAAAGACTATTCGATGCATAAATAGTTCAGTTTTGAAAACTTACATTTTGTTAAAGTCAGGTACTTGTGTATAATATCAACTAAAT
seq4 ATGCTAACCAAAGTTTCAGTTCGGACGTGTCGATGAGCGACGCTCAAAAAGGAAACAACATGCCAAATAGAAACGATCAATTCGGCGATGGAAATCAGAACAACGATCAGTTTGGAAATCAAAATAGAAATAACGGGAACGATCAGTTTAATAACATGATGCAGAATAAAGGGAATAATCAATTTAATCCAGGTAATCAGAACAGAGGT
```

Even easier is the convert() method. Let's try fastq to fasta.

```python
#!/usr/bin/env python3
from Bio import SeqIO
count = SeqIO.convert('./files/pfb.fastq', 'fastq', './files/pfb.converted.fa', 'fasta')
```

Hmm, was that easy or what??!??!!?

Parsing BLAST output

For simple BLAST parsing, ask for output format in tab-separated columns (`-outfmt 6` or `-outfmt 7`) Both these formats are customizable! See next section.

If you want to parse the full output of blast with biopython, it's best to work with XML formatted BLAST output `-outfmt 5`. It breaks the parsing method less easily. Code is stable for working with NCBI blast.

You can get biopython to run the blast for you too. See `Bio.NCBIWWW`

To parse the output, you'll write something like this

```

>>> from Bio.Blast import NCBIXML
>>> result_handle = open("my_blast.xml")
>>> blast_records = NCBIXML.parse(result_handle)
>>> for blast_record in blast_records:
>>>   for alignment in blast_record.alignments:
>>>     for hsp in alignment.hsps:
>>>         if hsp.expect < 1e-10:
>>>            print('id', alignment.title)
>>>            print('E = ' , hsp.expect)
```

You can also use the more general SearchIO

The code exists, but is likely to change over the next few versions of biopython.

It will handle other sequence search tools such as FASTA, HMMER etc as well as BLAST. ReturnsQuery objects that contain one or more Hit objects that contain one or more HSP objects, like in a blast report. Can handle blast tab-separated text output.

You'll write something like this

```
>>> from Bio import SearchIO
>>> idx = SearchIO.index('tab_2226_tblastn_001.txt', 'blast-tab')
>>> sorted(idx.keys())
['gi|11464971:4-101', 'gi|16080617|ref|NP_391444.1|']
>>> idx['gi|16080617|ref|NP_391444.1|']
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> idx.close()
```

There are many other uses for Biopython

- reading multiple sequence alignments
- searching on remote biological sequence databases
- working with protein structure (requires numpy to be installed)
- biochemical pathways (KEGG)
- drawing pictures of genome and sequence features
- population genetics

Why use biopython

Massive time saver once you know your way around the classes.

Reuse someone else's code. Very quick parsing of many common file formats.

Clean code.