LAB MANUAL

Course: BIO-310 Introduction to Bioinformatics

Learning Procedure

1) Stage J (Journey inside out to the concept)
2) Stage a_1 (Apply the learned)
3) Stage v (Verify the accuracy)
4) Stage a_2 (Asses your work)

Department of Computer Science

COMSATS University Islamabad, Attock Campus

Table of Contents

LAB #1 (Introduction to Biopython)

Statement Purpose

The purpose of the lab is to introduce the environment of Biopython. Students will be familiar with the installation and basics of Bio python.

Activity Outcomes

- Install Biopython
- Read different types of sequences
- DNA transcription
- DNA Translation

Stage (J)

Introduction

This activity consists of installation of Biopython and creating simple sequence operations like reading, printing, copying etc. DNA transcription and translation will also be discussed.

Stage (a1)

Lab Activities

Activity 1.

Install Biopython and working with sequences

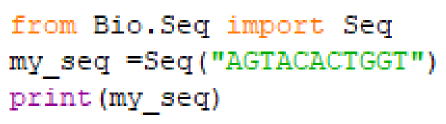
Solution

Biopython is a set of libraries to provide the ability to deal with “things" of interest to biologists working on the computer.
Biopython can be installed by either downloading from the official website or via the following command

pip install biopython

Working with sequences

Sequences are the central object in bioinformatics. Biopython provides Seq objects to deal with sequences. We know that sequences are string of letters like `AGTACACTGGT'.



```
from Bio.Seq import Seq
my_seq =Seq("AGTACACTGGT")
print(my_seq)
```

Parsing sequence fille formats

DNA data are available in different file formats. Parsing these files for useful manipulation is a challenge. Biopython supports to read and parse different types of files.

Simple FASTA parsing example

FASTA files usually looks like this. A sequence in FASTA format begins with a single-line description, followed by lines of sequence data. The description line is distinguished from the sequence data by a greater-than (">") symbol in the first column.

```
>gi|2765658|emb|Z78533.1|CIZ78533 C.irapeanum 5.8S rRNA gene and ITS1
and ITS2 DNA
CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGGAATAAACGATCGAGTG
AATCCGGAGGACCGGTGTACTCAGCTCACCGGGGGCATTGCTCCCGTGGTGACCCTGATTTGTTGTTGGG
```

The file contains records, each has a line starting with “>" (greater-than symbol) followed by the sequence on one or more lines. Now try this in Python:

```
from Bio import SeqIO
for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta"):
    print(seq_record.id)
    print(repr(seq_record.seq))
    print(len(seq_record))
```

You should get something like this on your screen:

```
gi|2765658|emb|Z78533.1|CIZ78533
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', SingleLetter
Alphabet())
740
gi|2765657|emb|Z78532.1|CCZ78532
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGACAACAG...GGC', SingleLetter
Alphabet())
753
gi|2765656|emb|Z78531.1|CFZ78531
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGACAGCAG...TAA', SingleLetter
Alphabet())
748
```

Notice that the FASTA format does not specify the alphabet, so Bio.SeqIO has defaulted to the rather generic SingleLetterAlphabet() rather than something DNA speci_c.

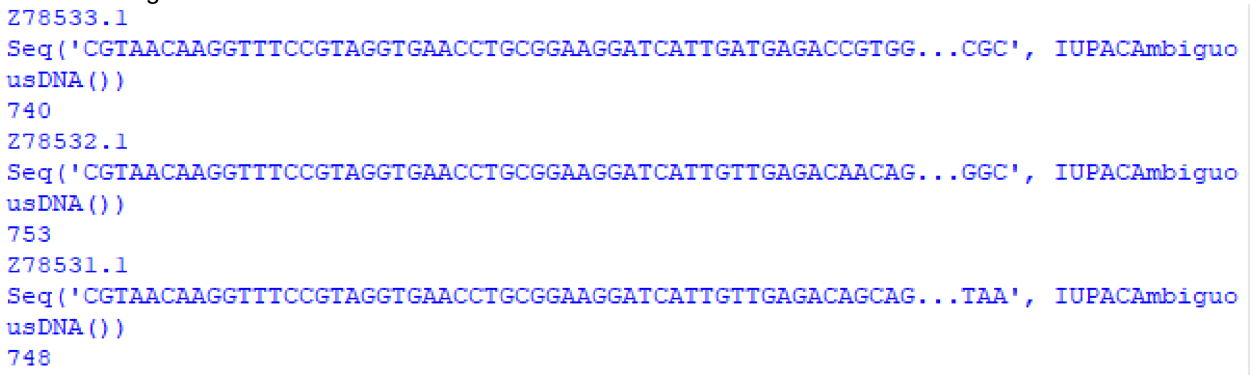
Simple GenBank parsing example

GenBank format (GenBank Flat File Format) consists of an annotation section and a sequence section. The start of the annotation section is marked by a line beginning with the word "LOCUS". The start of sequence section is marked by a line beginning with the word "ORIGIN" and the end of the section is marked by a line with only "//".

Now let's load the GenBank _le ls orchid.gbk instead - notice that the code to do this is almost identical to the snippet used above for the FASTA _le - the only difference is we change the file name and the format string:

```
from Bio import SeqIO
for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank"):
    print(seq_record.id)
    print(repr(seq_record.seq))
    print(len(seq_record))
```

This should give:



```
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', IUPACAmbiguo
usDNA())
740
Z78532.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGACAACAG...GGC', IUPACAmbiguo
usDNA())
753
Z78531.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGACAGCAG...TAA', IUPACAmbiguo
usDNA())
748
```

LAB #2 (Sequence Objects)

Statement Purpose

The purpose of the lab is to introduce the sequence objects available in biopython. This lab will be used as building block for coming lab activities.

Activity Outcomes

- Handling Sequences
- Dealing with sequences and alphabets
- Slicing Sequences
- Relationship between strings and sequences
- Concatenating or adding sequences
- Changing case
- Nucleotide sequences and (reverse) complements
- Transcription
- Translation

Stage (J)

Introduction

Biological sequences are arguably the central object in Bioinformatics, and in this chapter we'll introduce the Biopython mechanism for dealing with sequences, the Seq object.
There are two important differences between Seq objects and standard Python strings. First of all, they have different methods. Although the Seq object supports many of the same methods as a plain string, its translate() method differers by doing biological translation, and there are also additional biologically relevant methods like reverse_complement(). Secondly, the Seq object has an important attribute, alphabet, which is an object describing what the individual characters making up the sequence string “mean", and how they should be interpreted. For example, is AGTACACTGGT a DNA sequence, or just a protein sequence that happens to be rich in Alanines, Glycines, Cysteines and Threonines.

Stage (a1)

Lab Activities

Sequences and Alphabets

The alphabet object is perhaps the important thing that makes the Seq object more than just a string. The currently available alphabets for Biopython are defined in the Bio.Alphabet module that uses IUPAC alphabets (http://www.sbcs.qmul.ac.uk/iupac/) to deal with some of our favorite objects: DNA, RNA and Proteins.

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC
#my_seq = Seq("AGTACACTGGT")
#my_seq = Seq("AGTACACTGGT", IUPAC.unambiguous_dna)
my_seq = Seq("AGTACACTGGT", IUPAC.protein)
print(my_seq)
print(my_seq.alphabet)
```

Sequences act like strings

In many ways, we can deal with Seq objects as if they were normal Python strings, for example getting the length, or iterating over the elements:

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC
from Bio.SeqUtils import GC
my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
#for index, letter in enumerate(my_seq):
   # print("%i %s" % (index, letter))
print("Length of seq is ",len(my_seq))
print("first letter ",my_seq[0])
print("third letter ",my_seq[2])
print("last letter",my_seq[-1])
print("Count of G is ",my_seq.count("G"))#coutns occurance of a letter
print("Count of C is ",my_seq.count("C"))
print(my_seq.count("G") + my_seq.count("C"))
print(GC(my_seq))#built in funtion that tells the percentage
```

Slicing a sequence

Two things are interesting to note. First, this follows the normal conventions for Python strings. So the first element of the sequence is 0 (which is normal for computer science, but not so normal for biology). When you do a slice the first item is included (i.e. 4 in this case) and the last is excluded (12 in this case), which is the way things work in Python, but of course not necessarily the way everyone in the world would expect. The main goal is to stay consistent with what Python does.
The second thing to notice is that the slice is performed on the sequence data string, but the new object produced is another Seq object which retains the alphabet information from the original Seq object. Also like a Python string, you can do slices with a start, stop and stride (the step size, which defaults to one). For example, we can get the first, second and third codon positions of this DNA sequence:

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC
my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
print(my_seq[4:12])#from 4 to 11
print(my_seq[0::3])#starts from 0 till the end and step of 3
print(my_seq[1::3])
```

Concatenating or adding sequences

Naturally, you can in principle add any two Seq objects together - just like you can with Python strings to concatenate them. However, you can't add sequences with incompatible alphabets, such as a protein sequence and a DNA sequence:

```
from Bio.Alphabet import IUPAC
from Bio.Seq import Seq
protein_seq = Seq("EVRNAK", IUPAC.protein)
dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
dna_seq1 = Seq("TCGA", IUPAC.unambiguous_dna)
#protein_seq + dna_seq#this operation is not allowed
print(dna_seq+dna_seq1)
```

Changing case

Python strings have very useful upper and lower methods for changing the case. As of Biopython 1.53, the Seq object gained similar methods which are alphabet aware. For example,

```
from Bio.Seq import Seq
from Bio.Alphabet import generic_dna
dna_seq = Seq("acgtACGT", generic_dna)
print(dna_seq)
print(dna_seq.upper())
print(dna_seq.lower())
```

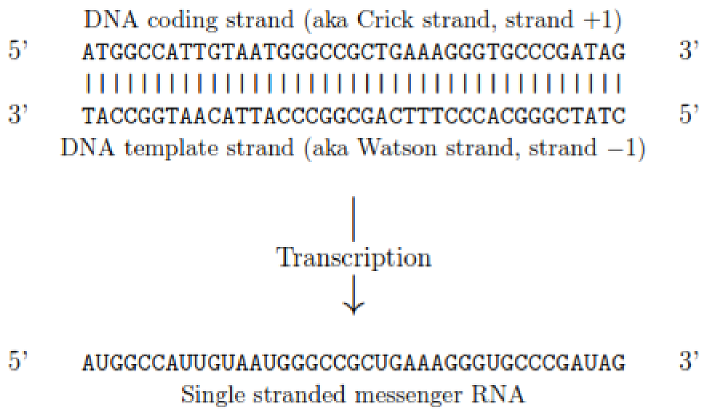
Nucleotide sequences and (reverse) complements

For nucleotide sequences, you can easily obtain the complement or reverse complement of a Seq object using its built-in methods:
In all of these operations, the alphabet property is maintained. This is very useful in case you accidentally end up trying to do something weird like take the (reverse) complement of a protein sequence:

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC
my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
print(my_seq)
print(my_seq.complement())
print(my_seq.reverse_complement())
```

Transcription

Before talking about transcription, lets try to clarify the strand issue. Consider the following (made up) stretch of double stranded DNA which encodes a short peptide:


DNA coding strand (aka Crick strand, strand +1)
5' ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG 3'
3' TACCGGTAACATTACCCGGCGACTTTCCCACGGGCTATC 5'
DNA template strand (aka Watson strand, strand −1)
Transcription
5' AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG 3'
Single stranded messenger RNA

The actual biological transcription process works from the template strand, doing a reverse complement (TCAG → CUGA) to give the mRNA. However, in Biopython and bioinformatics in general, we typically work directly with the coding strand because this means we can get the mRNA sequence just by switching T → U.

Now let's actually get down to doing a transcription in Biopython. First, let's create `Seq` objects for the coding and template DNA strands:

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC
from Bio.Alphabet import _verify_alphabet
coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG",
                 IUPAC.unambiguous_dna)
print(coding_dna)
template_dna = coding_dna.reverse_complement()
print(template_dna)
messenger_rna = coding_dna.transcribe()
print(messenger_rna)
print(messenger_rna.alphabet)#changed from DNA to RNA
print(_verify_alphabet(coding_dna))#verify alphabet in sequnce
```

Translation

Seq object also provides method for translation. The translation tables available in Biopython are based on those from the NCBI. By default, translation will use the standard genetic code (NCBI table id 1). Suppose we are dealing with a mitochondrial sequence. We need to tell the translation function to use the relevant genetic code instead:

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC
messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG",
                    IUPAC.unambiguous_rna)#can also be translated from DNA
print(messenger_rna)
#protien_Seq=messenger_rna.translate()#table options are also available
#e.g. table="Vertebrate Mitochondrial" or table=2, 2 is table number in NCBI
#protien_Seq=messenger_rna.translate(to_stop=True)
print(protien_Seq)
print(protien_Seq.alphabet)
```

Translation Tables

In the previous sections we talked about the Seq object translation method (and mentioned the equivalent function in the Bio.Seq module { see Section3.14). Internally these use codon table objects derived from the NCBI information at ftp://ftp.ncbi.nlm.nih.gov/entrez/misc/data/gc.prt, also shown on https: //www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi in a much more readable layout. As before, let's just focus on two choices: the Standard translation table, and the translation table for Vertebrate Mitochondrial DNA.

```
from Bio.Data import CodonTable
standard_table = CodonTable.unambiguous_dna_by_name["Standard"]
mito_table = CodonTable.unambiguous_dna_by_name["Vertebrate Mitochondrial"]
#mito_table = CodonTable.unambiguous_dna_by_id[2]#alternative option
#print(standard_table)
#print(mito_table)
#print(standard_table.stop_codons)
print(standard_table.start_codons)
```

You will see output like this

Table 1 Standard, SGC0

	T	C	A	G	
T	TTT F	TCT S	TAT Y	TGT C	T
T	TTC F	TCC S	TAC Y	TGC C	C
T	TTA L	TCA S	TAA Stop	TGA Stop	A
T	TTG L(s)	TCG S	TAG Stop	TGG W	G
C	CTT L	CCT P	CAT H	CGT R	T
C	CTC L	CCC P	CAC H	CGC R	C
C	CTA L	CCA P	CAA Q	CGA R	A
C	CTG L(s)	CCG P	CAG Q	CGG R	G
A	ATT I	ACT T	AAT N	AGT S	T
A	ATC I	ACC T	AAC N	AGC S	C
A	ATA I	ACA T	AAA K	AGA R	A
A	ATG M(s)	ACG T	AAG K	AGG R	G
G	GTT V	GCT A	GAT D	GGT G	T
G	GTC V	GCC A	GAC D	GGC G	C
G	GTA V	GCA A	GAA E	GGA G	A
G	GTG V	GCG A	GAG E	GGG G	G

LAB #3 Sequence annotation objects

Statement Purpose

In previous lab, we introduced the sequence classes. Immediately “above" the Seq class is the Sequence Record or SeqRecord class, defined in the Bio.SeqRecord module. This class allows higher level features such as identifiers and features (as SeqFeature objects) to be associated with the sequence and is used throughout the sequence input/output interface Bio.SeqIO. The purpose of this lab is to explore these advanced features.

Activity Outcomes

- The SeqRecord object
- Creating a SeqRecord
- Feature, location and position objects
- Comparison
- References
- The format method
- Slicing a SeqRecord

Stage (J)

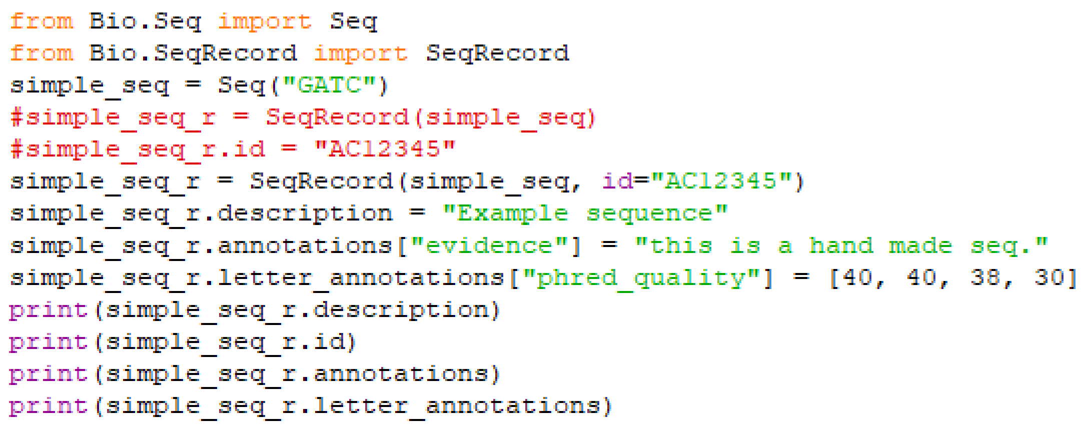
The SeqRecord object

The SeqRecord (Sequence Record) class is defined in the Bio.SeqRecord module. This class allows higher level features such as identifiers and features to be associated with a sequence (Lab 2), and is the basic data type for the Bio.SeqIO sequence input/output interface. The SeqRecord class itself is quite simple, and offers the following information as attributes:

.seq: The sequence itself, typically a Seq object.
.id: The primary ID used to identify the sequence string. In most cases this is something like an accession number.
.name A “common" name/id for the sequence string. In some cases, this will be the same as the accession number, but it could also be a clone name. I think of this as being analogous to the LOCUS id in a GenBank record.
.description A human readable description or expressive name for the sequence string.
.letter annotations Holds per-letter-annotations using a (restricted) dictionary of additional information about the letters in the sequence. The keys are the name of the information, and the information is contained in the value as a Python sequence (i.e. a list, tuple or string) with the same length as the sequence itself. This is often used for quality scores (to be discussed latter).
.annotations A dictionary of additional information about the sequence. The keys are the name of the information, and the information is contained in the value. This allows the addition of more “unstructured" information to the sequence.
.features A list of SeqFeature objects with more structured information about the features on a sequence (e.g. position of genes on a genome, or domains on a protein sequence).
.dbxrefs -A list of database cross-references as strings.

Creating a SeqRecord

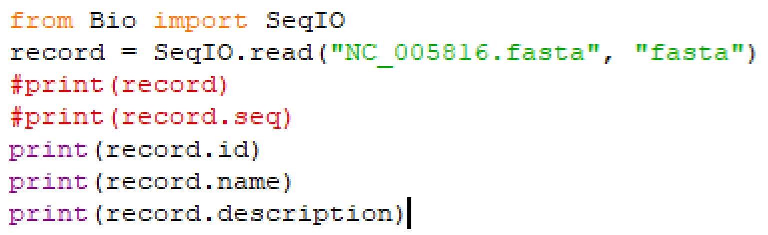
SeqRecord objects from scratch
To create a SeqRecord at a minimum you just need a Seq object: Additionally, you can also pass the id, name and description to the initialization function, but if not they will be set as strings indicating they are unknown, and can be modified subsequently:



```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
simple_seq = Seq("GATC")
#simple_seq_r = SeqRecord(simple_seq)
#simple_seq_r.id = "AC12345"
simple_seq_r = SeqRecord(simple_seq, id="AC12345")
simple_seq_r.description = "Example sequence"
simple_seq_r.annotations["evidence"] = "this is a hand made seq."
simple_seq_r.letter_annotations["phred_quality"] = [40, 40, 38, 30]
print(simple_seq_r.description)
print(simple_seq_r.id)
print(simple_seq_r.annotations)
print(simple_seq_r.letter_annotations)
```

SeqRecord objects from FASTA files

Back in Lab 3 you have seen the function Bio.SeqIO.parse(...) used to loop over all the records in a file as SeqRecord objects. The Bio.SeqIO module has a sister function for use on files which contain just one record which we'll use here.

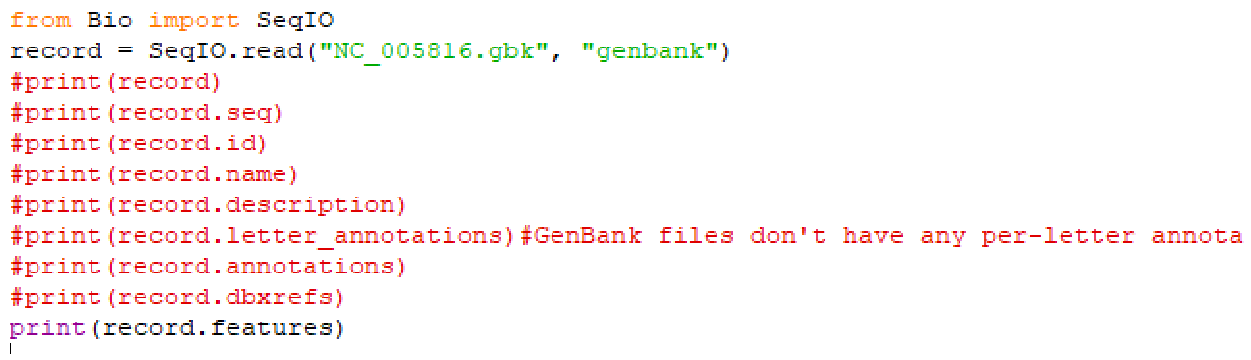


```
from Bio import SeqIO
record = SeqIO.read("NC_005816.fasta", "fasta")
#print(record)
#print(record.seq)
print(record.id)
print(record.name)
print(record.description)
```

As you can see above, the firrst word of the FASTA record's title line (after removing the greater than symbol) is used for both the id and name attributes. The whole title line (after removing the greater than symbol) is used for the record description. This is deliberate, partly for backwards compatibility reasons:
Note that none of the other annotation attributes get populated when reading a FASTA file:

SeqRecord objects from GenBank files

As in the previous example, we're going to look at the whole sequence for Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, originally downloaded from the NCBI, but this time as a GenBank file.



```
from Bio import SeqIO
record = SeqIO.read("NC_005816.gbk", "genbank")
#print(record)
#print(record.seq)
#print(record.id)
#print(record.name)
#print(record.description)
#print(record.letter_annotations)#GenBank files don't have any per-letter annota
#print(record.annotations)
#print(record.dbxrefs)
print(record.features)
```

Feature, location and position objects
SeqFeature objects

Sequence features are an essential part of describing a sequence. Once you get beyond the sequence itself, you need some way to organize and easily get at the more “abstract" information that is known about the sequence. While it is probably impossible to develop a general sequence feature class that will cover everything, the Biopython SeqFeature class attempts to encapsulate as much of the information about the sequence as possible. The design is heavily based on the GenBank/EMBL feature tables, so if you understand how they look, you'll probably have an easier time grasping the structure of the Biopython classes. The key idea about each SeqFeature object is to describe a region on a parent sequence, typically a SeqRecord object. That region is described with a location object, typically a range between two positions.
The SeqFeature class has a number of attributes, so first we'll list them and their general features,

.type This is a textual description of the type of feature (for instance, this will be something like `CDS' or `gene').
.location The location of the SeqFeature on the sequence that you are dealing with. The SeqFeature delegates much of its functionality to the location object, and includes a number of shortcut attributes for properties of the location:
.ref shorthand for .location.ref i.e. reference sequence the location is referring to.
Usually just None.
.ref_db shorthand for .location.ref_db { specifies the database any identifier in .ref refers to.
Usually just None.
.strand shorthand for .location.strand, the strand on the sequence that the feature is located on. For double stranded nucleotide sequence this may either be 1 for the top strand, 1 for the bottom strand, 0 if the strand is important but is unknown, or None if it doesn't matter. This is None for proteins, or single stranded sequences.

Positions and locations
The key idea about each SeqFeature object is to describe a region on a parent sequence, for which we use a location object, typically describing a range between two positions.
Position: This refers to a single position on a sequence, which may be fuzzy or not. For instance, 5, 20, <100 and >200 are all positions.
Location: A location is region of sequence bounded by some positions. For instance 5..20 (i. e. 5 to 20) is a location.

FeatureLocation object
Unless you work with eukaryotic genes, most SeqFeature locations are extremely simple - you just need start and end coordinates and a strand. That's essentially all the basic FeatureLocation object does.

Compound Location object
Biopython 1.62 introduced the CompoundLocation as part of a restructuring of how complex locations made up of multiple regions are represented. The main usage is for handling `join' locations in EMBL/GenBank files.

Fuzzy Positions
So far we've only used simple positions. One complication in dealing with feature locations comes in the positions themselves. In biology many times things aren't entirely certain. Basically there are several types of fuzzy positions, so we have five classes do deal with them:
ExactPosition: As its name suggests, this class represents a position which is specified as exact along the sequence. This is represented as just a number, and you can get the position by looking at the position attribute of the object.
BeforePosition: This class represents a fuzzy position that occurs prior to some specified site. In GenBank/EMBL notation, this is represented as something like `<13', signifying that the real position is located somewhere less than 13. To get the speci_ed upper boundary, look at the position attribute of the object.
AfterPosition: Contrary to BeforePosition, this class represents a position that occurs after some specified site. This is represented in GenBank as `>13', and like BeforePosition, you get the boundary number by looking at the position attribute of the object.
WithinPosition: Occasionally used for GenBank/EMBL locations, this class models a position which occurs somewhere between two specified nucleotides. In GenBank/EMBL notation, this would be represented as `(1.5)', to represent that the position is somewhere within the range 1 to 5. To get the information in this class you have to look at two attributes. The position attribute specifies the lower boundary of the range we are looking at, so in our example case this would be one. The extension attribute specifies the range to the higher boundary, so in this case it would be 4. So object.position is the lower boundary and object.position + object.extension is the upper boundary.
OneOfPosition: Occasionally used for GenBank/EMBL locations, this class deals with a position where several possible values exist, for instance you could use this if the start codon was unclear and there where two candidates for the start of the gene. Alternatively, that might be handled explicitly as two related gene features.
UnknownPosition: This class deals with a position of unknown location. This is not used in GenBank/EMBL,
but corresponds to the `?' feature coordinate used in UniProt.

```
from Bio import SeqFeature
start_pos = SeqFeature.AfterPosition(5)
end_pos = SeqFeature.BetweenPosition(9, left=8, right=9)
my_location = SeqFeature.FeatureLocation(start_pos, end_pos)
print(my_location)
print(my_location.start)#prints fuzzy location
print(my_location.end)
print(int(my_location.start))#prints int loacation
print(int(my_location.end))
exact_location = SeqFeature.FeatureLocation(5, 9)
print(exact_location)#prints exact location
```

Sequence described by a feature or location

A SeqFeature or location object doesn't directly contain a sequence, instead the location describes how to get this from the parent sequence. For example consider a (short) gene sequence with location 5:18 on the reverse strand.

```
from Bio.Seq import Seq
from Bio.SeqFeature import SeqFeature, FeatureLocation
example_parent = Seq("ACCGAGACGGCAAAGGCTAGCATAGGTATGAGA \
CTTCCTTCCTGCCAGTGCTGAGGAACTGGGAGCCTAC")
example_feature = SeqFeature(FeatureLocation(5, 18))
feature_seq = example_parent[example_feature.location. \
start:example_feature.location.end].reverse_complement()
print(feature_seq)
```

Comparison

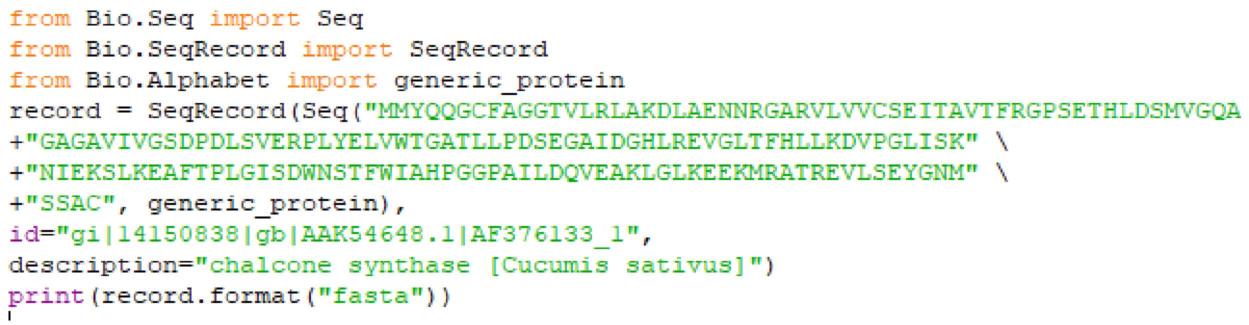
The SeqRecord objects can be very complex, but here's a simple example:

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
record1 = SeqRecord(Seq("ACGT"), id="test")
record2 = SeqRecord(Seq("ACGT"), id="test")
#print(record1 == record2)#this operation is not allowed
print(record1.id == record2.id)
print(record1.seq == record2.seq)
```

SeqRecord object cannot be compared directly. Instead you should check the attributes you are interested in.

The format method

The format() method of the SeqRecord class gives a string containing your record formatted using one of the output file formats supported by Bio.SeqIO, such as FASTA:



```python
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Alphabet import generic_protein
record = SeqRecord(Seq("MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQA
+"GAGAVIVGSDPDLSVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKDVPGLISK" \
+"NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRATREVLSEYGNM" \
+"SSAC", generic_protein),
id="gi|14150838|gb|AAK54648.1|AF376133_1",
description="chalcone synthase [Cucumis sativus]")
print(record.format("fasta"))
```

LAB #4 Sequence Input/Output

Statement Purpose

The purpose of this lab is to discuss the sequence input/output in more details. This aims to provide a simple interface for working with assorted sequence file formats in a uniform way.

Activity Outcomes

- Parsing or Reading Sequences
- Parsing or Reading Sequences
- Reading Sequence Files
- Iterating over the records in a sequence file
- Getting a list of the records in a sequence file
- Parsing sequences from compressed files
- Parsing sequences from the net
- Writing Sequence Files
- Converting between sequence file formats

Parsing or Reading Sequences

The workhorse function Bio.SeqIO.parse() is used to read in sequence data as SeqRecord objects. This function expects two arguments:
1.The first argument is a handle to read the data from, or a filename. A handle is typically a file opened for reading but could be the output from a command line program, or data downloaded from the internet (to be discussed later).
2.The second argument is a lower-case string specifying sequence format we don't try and guess the file format for you! See http://biopython.org/wiki/SeqIO for a full listing of supported formats.
There is an optional argument alphabet to specify the alphabet to be used. This is useful for file formats like FASTA where otherwise Bio.SeqIO will default to a generic alphabet.
The Bio.SeqIO.parse() function returns an iterator which gives SeqRecord objects. Iterators are typically used in a for loop as shown below.
Sometimes you'll find yourself dealing with files which contain only a single record. For this situation use the function Bio.SeqIO.read() which takes the same arguments. Provided there is one and only one record in the file, this is returned as a SeqRecord object. Otherwise an exception is raised.

Reading Sequence Files

In general Bio.SeqIO.parse() is used to read in sequence files as SeqRecord objects, and is typically used with a for loop like this:

```
from Bio import SeqIO
for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta"):
    print(seq_record.id)
    print(repr(seq_record.seq))
    print(len(seq_record))
```

The above example is repeated from the introduction in lab 2.

Iterating over the records in a sequence file

In the above examples, we have usually used a for loop to iterate over all the records one by one. You can use the for loop with all sorts of Python objects (including lists, tuples and strings) which support the iteration interface. The object returned by Bio.SeqIO is actually an iterator which returns SeqRecord objects. You get to see each record in turn, but once and only once. The plus point is that an iterator can save you memory when dealing with large files.
Instead of using a for loop, can also use the next () function on an iterator to step through the entries, like this:

```
from Bio import SeqIO
record_iterator = SeqIO.parse("ls_orchid.fasta", "fasta")
first_record = next(record_iterator)
print(first_record.id)
print(first_record.description)
second_record = next(record_iterator)
print(second_record.id)
print(second_record.description)
```

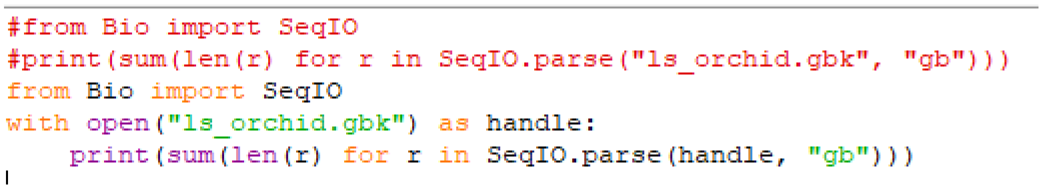
Getting a list of the records in a sequence file

we previously talked about the fact that Bio.SeqIO.parse() gives you a SeqRecord iterator, and that you get the records one by one. Very often you need to be able to access the records in any order. The Python list data type is perfect for this, and we can turn the record iterator into a list of SeqRecord objects using the built-in Python function list() like so:

```
from Bio import SeqIO
records = list(SeqIO.parse("ls_orchid.gbk", "genbank"))
print("Found %i records" % len(records))
print("The last record")
last_record = records[-1] #using Python's list tricks
print(last_record.id)
print(repr(last_record.seq))
print(len(last_record))
print("The first record")
first_record = records[0] #remember, Python counts from zero
print(first_record.id)
print(repr(first_record.seq))
print(len(first_record))
```

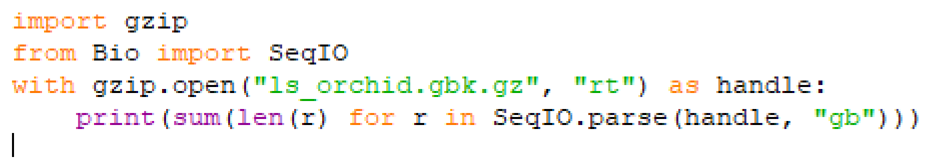
Parsing sequences from compressed files

In the previous examples, we looked at parsing sequence data from a file. Instead of using a file name, you can give Bio.SeqIO a handle, and in this example, we'll use handles to parse sequence from compressed files. As you'll have seen above, we can use Bio.SeqIO.read() or Bio.SeqIO.parse() with a file name for instance this quick example calculates the total length of the sequences in a multiple record GenBank file using a generator expression: Here we use a file handle instead, using the with statement to close the handle automatically:



```
#from Bio import SeqIO
#print(sum(len(r) for r in SeqIO.parse("ls_orchid.gbk", "gb")))
from Bio import SeqIO
with open("ls_orchid.gbk") as handle:
    print(sum(len(r) for r in SeqIO.parse(handle, "gb")))
```

Now, suppose we have a gzip compressed file instead? These are very commonly used on Linux. We can use Python's gzip module to open the compressed file for reading - which gives us a handle object:



```
import gzip
from Bio import SeqIO
with gzip.open("ls_orchid.gbk.gz", "rt") as handle:
    print(sum(len(r) for r in SeqIO.parse(handle, "gb")))
```

Parsing sequences from the net

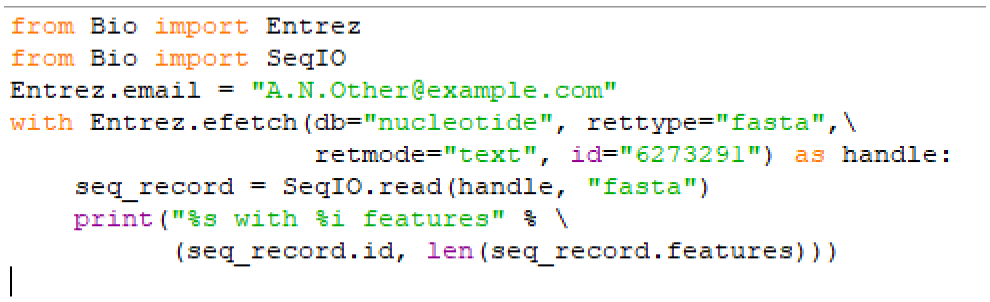
In the previous sections, we looked at parsing sequence data from a file (using a file name or handle), and from compressed files (using a handle). Here we'll use Bio.SeqIO with another type of handle, a network connection, to download and parse sequences from the internet.

Note that just because you can download sequence data and parse it into a SeqRecord object in one go doesn't mean this is a good idea. In general, you should probably download sequences once and save them to a file for reuse.

Parsing FASTA and GenBank records from the net

We will discuss Entrez EFetch interface in more detail later, but for now let's just connect to the NCBI and get a few Opuntia (prickly-pear) sequences from GenBank using their GI numbers.

First of all, let's fetch just one record. If you don't care about the annotations and features downloading a FASTA file is a good choice as these are compact. Now remember, when you expect the handle to contain one and only one record, use the Bio.SeqIO.read() function:



```
from Bio import Entrez
from Bio import SeqIO
Entrez.email = "A.N.Other@example.com"
with Entrez.efetch(db="nucleotide", rettype="fasta",\
                   retmode="text", id="6273291") as handle:
    seq_record = SeqIO.read(handle, "fasta")
    print("%s with %i features" % \
          (seq_record.id, len(seq_record.features)))
```

Genbank

```
from Bio import Entrez
from Bio import SeqIO
Entrez.email = "A.N.Other@example.com"
with Entrez.efetch(db="nucleotide", rettype="gb", \
                   retmode="text", id="6273291") as handle:
   # seq_record = SeqIO.read(handle, "gb")
   # print("%s with %i features" % \
         # (seq_record.id, len(seq_record.features)))

    for seq_record in SeqIO.parse(handle, "gb"):
        print("%s %s..." % (seq_record.id, seq_record.description[:50]))
        print("Sequence length %i, %i features, from: %s"%\
              (len(seq_record), len(seq_record.features),\
               seq_record.annotations["source"]))
```

Parsing SwissProt sequences from the net

Now let's use a handle to download a SwissProt file from ExPASy, something covered in more depth later. As mentioned above, when you expect the handle to contain one and only one record, use the Bio.SeqIO.read() function:

```
from Bio import ExPASy
from Bio import SeqIO
with ExPASy.get_sprot_raw("O23729") as handle:
    seq_record = SeqIO.read(handle, "swiss")
    print(seq_record.id)
    print(seq_record.name)
    print(seq_record.description)
    print(repr(seq_record.seq))
    print("Length %i" % len(seq_record))
    print(seq_record.annotations["keywords"])
```

Writing Sequence Files

We've talked about using Bio.SeqIO.parse() for sequence input (reading files), and now we'll look at Bio.SeqIO.write() which is for sequence output (writing files). This is a function taking three arguments: some SeqRecord objects, a handle or filename to write to, and a sequence format.
Here is an example, where we start by creating a few SeqRecord objects the hard way (by hand, rather than by loading them from a file):

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Alphabet import generic_protein
rec1 = SeqRecord(Seq("MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFR"\
+"GAGAVIVGSDPDLSVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKDVPGLISK" \
+"NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRATREVLSEYGNM" \
+"SSAC", generic_protein),
id="gi|14150838|gb|AAK54648.1|AF376133_1",
description="chalcone synthase [Cucumis sativus]")
rec2 = SeqRecord(Seq("YPDYYFRITNREHKAELKEKFQRMCDKSMIKKRYMYLTEEILK"\
+"DMVVVEIPKLGKEAAVKAIKEWGQ", generic_protein),
id="gi|13919613|gb|AAK33142.1|",
description="chalcone synthase [Fragaria vesca subsp. bracteata]")
rec3 = SeqRecord(Seq("MVTVEEFRRAQCAEGPATVMAIGTATPSNCVDQSTYPDYYFR"\
+"EKSMIKKRYMHLTEEILKENPNICAYMAPSLDARQDIVVVEVPKLGKEAAQKAIKEWGQP" \
+"KSKITHLVFCTTSGVDMPGCDYQLTKLLGLRPSVKRFMMYQQGCFAGGTVLRMAKDLAEN" \
+"NKGARVLVVCSEITAVTFRGPNDTHLDSLVGQALFGDGAAAVIIGSDPIPEVERPLFELV" \
+"SAAQTLLPDSEGAIDGHLREVGLTFHLLKDVPGLISKNIEKSLVEAFQPLGISDWNSLFW" \
+"IAHPGGPAILDQVELKLGLKQEKLKATRKVLSNYGNMSSACVLFILDEMRKASAKEGLGT" \
+"TGEGLEWGVLFGFGPGLTVETVVLHSVAT", generic_protein),
id="gi|13925890|gb|AAK49457.1|",
description="chalcone synthase [Nicotiana tabacum]")
my_records = [rec1, rec2, rec3]
from Bio import SeqIO
SeqIO.write(my_records, "my_example.faa", "fasta")
```

Converting between sequence file formats

In previous example we used a list of SeqRecord objects as input to the Bio.SeqIO.write() function, but it will also accept a SeqRecord iterator like we get from Bio.SeqIO.parse() { this lets us do file conversion by combining these two functions. For this example we'll read in the GenBank format file orchid.gbk and write it out in FASTA format:

```
from Bio import SeqIO
records = SeqIO.parse("ls_orchid.gbk", "genbank")
#count = SeqIO.write(records, "my_example.fasta", "fasta")
count = SeqIO.convert("ls_orchid.gbk", "genbank",\
                      "my_example.fasta", "fasta")
print("Converted %i records" % count)
```

LAB #5 Multiple Sequence Alignment objects

Statement Purpose

The purpose of this lab is to discuss the Multiple Sequence Alignment objects details. This aims to introduce MSA which is a collection of multiple sequences which have been aligned together usually with the insertion of gap characters, and addition of leading or trailing gaps such that all the sequence strings are the same length. Such an alignment can be regarded as a matrix of letters, where each row is held as a SeqRecord object internally.

Activity Outcomes

- Parsing or Reading Sequence Alignments
- Single Alignments
- Multiple Alignments
- Writing Alignments
- Converting between sequence alignment file formats
- Getting your alignment objects as formatted strings
- Slicing alignments

Parsing or Reading Sequence Alignments

We have two functions for reading in sequence alignments, Bio.AlignIO.read() and Bio.AlignIO.parse() which following the convention introduced in Bio.SeqIO are for files containing one or multiple alignments respectively.

Using Bio.AlignIO.parse() will return an iterator which gives MultipleSeqAlignment objects. Iterators are typically used in a for loop. Examples of situations where you will have multiple different alignments include resampled alignments from the PHYLIP tool seqboot, or multiple pairwise alignments from the EMBOSS tools water or needle, or Bill Pearson's FASTA tools.

However, in many situations you will be dealing with files which contain only a single alignment. In this case, you should use the Bio.AlignIO.read() function which returns a single MultipleSeqAlignment object.

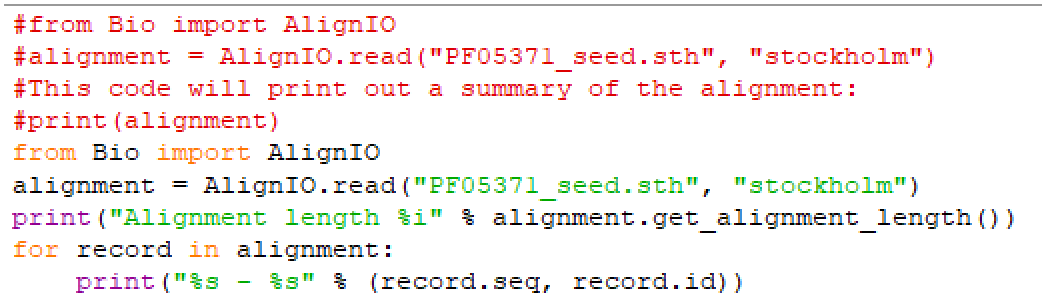
Both functions expect two mandatory arguments:

- The first argument is a handle to read the data from, typically an open file, or a filename.
- The second argument is a lower case string specifying the alignment format. As in Bio.SeqIO we don't try and guess the file format for you! See http://biopython.org/wiki/AlignIO for a full listing of supported formats.

Single Alignments

As an example, consider the annotation rich protein alignment in the PFAM or Stockholm file format:

This is the seed alignment for the Phage Coat Gp8 (PF05371) PFAM entry, saved as .sth file.



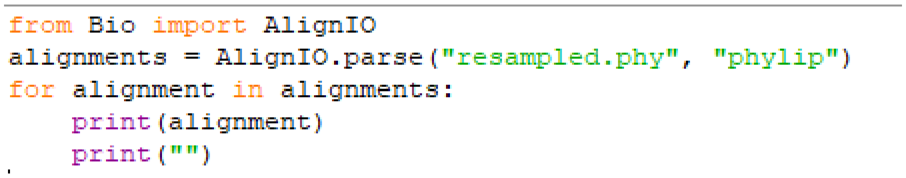
```
#from Bio import AlignIO
#alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
#This code will print out a summary of the alignment:
#print(alignment)
from Bio import AlignIO
alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
print("Alignment length %i" % alignment.get_alignment_length())
for record in alignment:
    print("%s - %s" % (record.seq, record.id))
```

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:

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 give output something like this.

If you wanted to read this in using Bio.AlignIO you could use:



```
from Bio import AlignIO
alignments = AlignIO.parse("resampled.phy", "phylip")
for alignment in alignments:
    print(alignment)
    print("")
```

This would give the following output.

```
SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAACCA Alpha
AAACCC Beta
ACCCCA Gamma
CCCAAC Delta
CCCAAA Epsilon

SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAACAA Alpha
AAACCC Beta
ACCCAA Gamma
CCCACC Delta
CCCAAA Epsilon

SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAAAAC Alpha
AAACCC Beta
AACAAC Gamma
CCCCCA Delta
CCCAAC Epsilon

SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAAACC Alpha
ACCCCC Beta
AAAACC Gamma
CCCCAA Delta
CAAACC Epsilon
```

As with the function Bio.SeqIO.parse(), using Bio.AlignIO.parse() returns an iterator. If you want to keep all the alignments in memory at once, which will allow you to access them in any order, then turn the iterator into a list:

```
from Bio import AlignIO
alignments = list(AlignIO.parse("resampled.phy", "phylip"))
last_align = alignments[-1]
first_align = alignments[0]
print(last_align)
print(first_align)
```

Writing Alignments

We've talked about using Bio.AlignIO.read() and Bio.AlignIO.parse() for alignment input (reading files), and now we'll look at Bio.AlignIO.write() which is for alignment output (writing files). This is a function taking three arguments: some MultipleSeqAlignment objects (or for backwards compatibility the obsolete Alignment objects), a handle or file name to write to, and a sequence format. Here is an example, where we start by creating a few MultipleSeqAlignment objects the hard way (by hand, rather than by loading them from a file). Note we create some SeqRecord objects to construct the alignment from.

```
from Bio.Alphabet import generic_dna
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio import AlignIO
from Bio.Align import MultipleSeqAlignment
align1 = MultipleSeqAlignment([
SeqRecord(Seq("ACTGCTAGCTAG", generic_dna), id="Alpha"),
SeqRecord(Seq("ACT-CTAGCTAG", generic_dna), id="Beta"),
SeqRecord(Seq("ACTGCTAGDTAG", generic_dna), id="Gamma"),])
align2 = MultipleSeqAlignment([
SeqRecord(Seq("GTCAGC-AG", generic_dna), id="Delta"),
SeqRecord(Seq("GACAGCTAG", generic_dna), id="Epsilon"),
SeqRecord(Seq("GTCAGCTAG", generic_dna), id="Zeta"),])
align3 = MultipleSeqAlignment([
SeqRecord(Seq("ACTAGTACAGCTG", generic_dna), id="Eta"),
SeqRecord(Seq("ACTAGTACAGCT-", generic_dna), id="Theta"),
SeqRecord(Seq("-CTACTACAGGTG", generic_dna), id="Iota"),])
my_alignments = [align1, align2, align3]
AlignIO.write(my_alignments, "my_example.phy", "phylip")
```

If you open this file in your favourite text editor, it should look like this:

```
 3 12
Alpha      ACTGCTAGCT AG
Beta       ACT-CTAGCT AG
Gamma      ACTGCTAGDT AG
 3 9
Delta      GTCAGC-AG
Epislon    GACAGCTAG
Zeta       GTCAGCTAG
 3 13
Eta        ACTAGTACAG CTG
Theta      ACTAGTACAG CT-
Iota       -CTACTACAG GTG
```

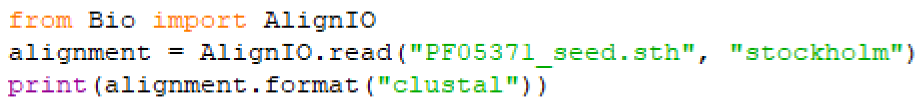
Converting between sequence alignment file formats

Converting between sequence alignment file formats with Bio.AlignIO works in the same way as converting between sequence file formats with Bio.SeqIO. We load generally the alignment(s) using Bio.AlignIO.parse() and then save them using the Bio.AlignIO.write() or just use the Bio.AlignIO.convert() helper function. For this example, we'll load the PFAM/Stockholm format file used earlier and save it as a ClustalW format file:

```
from Bio import AlignIO
count = AlignIO.convert("PF05371_seed.sth", "stockholm",\
                        "PF05371_seed.aln", "clustal")
print("Converted %i alignments" % count)
#or
#alignments = AlignIO.parse("PF05371_seed.sth", "stockholm")
#count = AlignIO.write(alignments, "PF05371_seed.aln", "clustal")
#print("Converted %i alignments" % count)
```

Getting your alignment objects as formatted strings

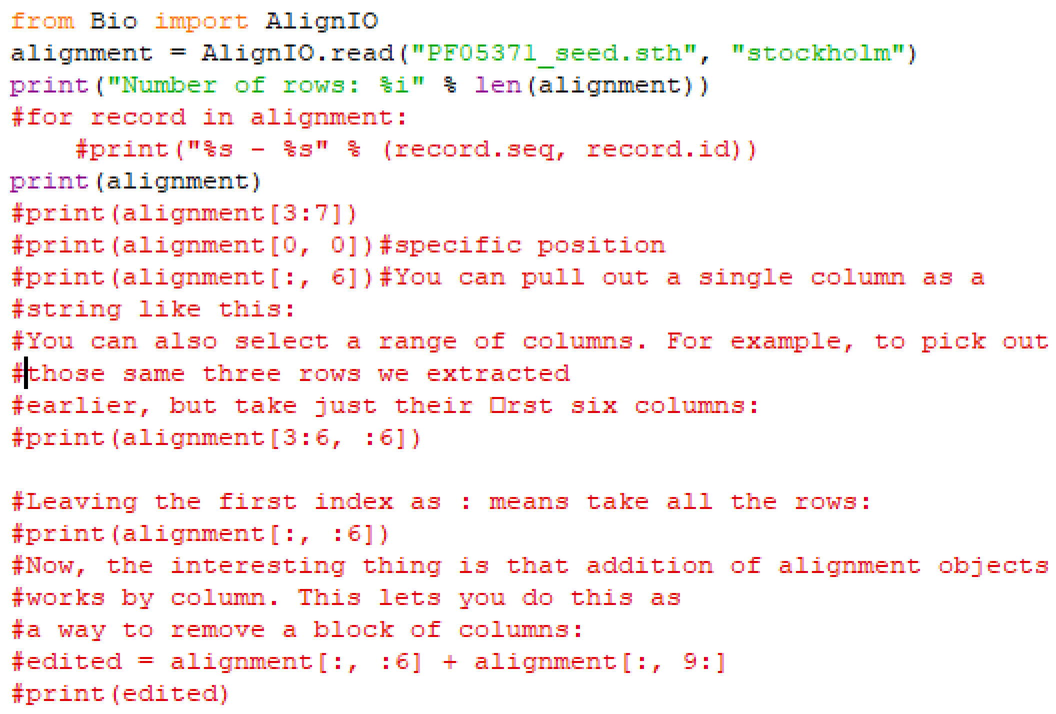
The Bio.AlignIO interface is based on handles, which means if you want to get your alignment(s) into a string in a particular file format you need to do a little bit more work (see below). However, you will probably prefer to take advantage of the alignment object's format() method. This takes a single mandatory argument, a lower-case string which is supported by Bio.AlignIO as an output format. For example:



```
from Bio import AlignIO
alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
print(alignment.format("clustal"))
```

Slicing alignments

First of all, in some senses the alignment objects act like a Python list of SeqRecord objects (the rows). With this model in mind hopefully the actions of len() (the number of rows) and iteration (each row as a SeqRecord) make sense:



```
from Bio import AlignIO
alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
print("Number of rows: %i" % len(alignment))
#for record in alignment:
    #print("%s - %s" % (record.seq, record.id))
print(alignment)
#print(alignment[3:7])
#print(alignment[0, 0])#specific position
#print(alignment[:, 6])#You can pull out a single column as a
#string like this:
#You can also select a range of columns. For example, to pick out
#those same three rows we extracted
#earlier, but take just their □rst six columns:
#print(alignment[3:6, :6])

#Leaving the first index as : means take all the rows:
#print(alignment[:, :6])
#Now, the interesting thing is that addition of alignment objects
#works by column. This lets you do this as
#a way to remove a block of columns:
#edited = alignment[:, :6] + alignment[:, 9:]
#print(edited)
```

LAB #6 Sequence Alignment Tools

Statement Purpose

The purpose of this lab is to discuss the Multiple Sequence Alignment tools in detail. There are lots of algorithms out there for aligning sequences, both pairwise alignments and multiple sequence alignments. We will implement some of these tools to get familiar with MSA.

Activity Outcomes

- Pairwise sequence alignment
- Biopython's pairwise2
- ClustalW
- MUSCLE
- MUSCLE using stdout
- MUSCLE using stdin and stdout
- EMBOSS needle and water

Pairwise sequence alignment

Pairwise sequence alignment is the process of aligning two sequences to each other by optimizing the similarity score between them. Biopython includes a built-in pairwise aligner that implements the Needleman-Wunsch, Smith-Waterman.

Global alignment

This method finds the best alignment over the entire lengths of the 2 sequences. What is the maximum similarity between sequence.

Local alignment

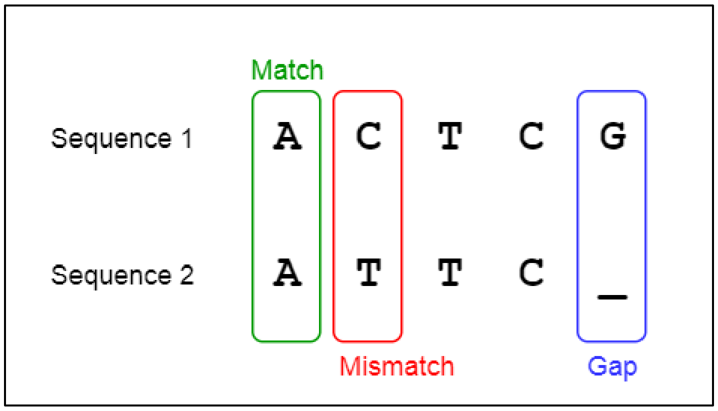
This method finds the most similar sub-sequences among the 2 sequences.

Three basic aspects are considered when assigning scores. They are,

Match value—Value assigned for matching characters

Mismatch value—Value assigned for mismatching characters

Gap penalty—Value assigned for spaces

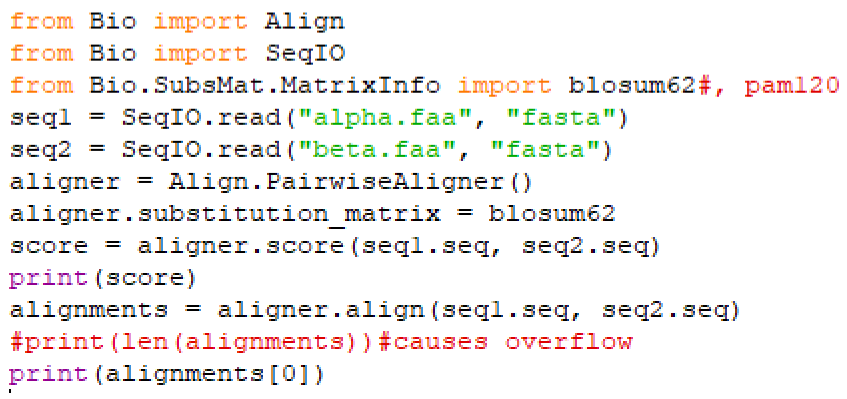


```
from Bio import Align
aligner = Align.PairwiseAligner()
seq1 = "GAACT"
seq2 = "GAT"
score = aligner.score(seq1, seq2)
print(score)
#alignments = aligner.align(seq1, seq2)
#for alignment in alignments:
   # print(alignment)
#print(aligner)
#print(aligner.algorithm)
```

By default, a global pairwise alignment is performed, which finds the optimal alignment over the whole length of seq1 and seq2. Instead, a local alignment will finds the subsequence of seq1 and seq2 with the highest alignment score. Local alignments can be generated by setting aligner.mode to "local":

```
from Bio import Align
aligner = Align.PairwiseAligner()
aligner.mode = 'local'
seq1 = "AGAACTC"
seq2 = "GAACT"
score = aligner.score(seq1, seq2)
print(score)
alignments = aligner.align(seq1, seq2)
for alignment in alignments:
    print(alignment)
```

An Example with substitution matrices



```
from Bio import Align
from Bio import SeqIO
from Bio.SubsMat.MatrixInfo import blosum62#, pam120
seq1 = SeqIO.read("alpha.faa", "fasta")
seq2 = SeqIO.read("beta.faa", "fasta")
aligner = Align.PairwiseAligner()
aligner.substitution_matrix = blosum62
score = aligner.score(seq1.seq, seq2.seq)
print(score)
alignments = aligner.align(seq1.seq, seq2.seq)
#print(len(alignments))#causes overflow
print(alignments[0])
```

Biopython's pairwise2

Biopython has its own module to make local and global pairwise alignments, Bio.pairwise2. This module contains essentially the same algorithms as water (local) and needle (global) from the EMBOSS suite (discussed later) and should return the same results.

```
from Bio import pairwise2
from Bio import SeqIO
seq1 = SeqIO.read("alpha.faa", "fasta")
seq2 = SeqIO.read("beta.faa", "fasta")
alignments = pairwise2.align.globalxx(seq1.seq, seq2.seq)
#print(len(alignments))
#print(alignments[0])
print(pairwise2.format_alignment(*alignments[0]))
```

The function align.globalxx has two letters of the function name (here: xx), which are used for decoding the scores and penalties for matches (and mismatches) and gaps. The first letter decodes the match score, e.g. x means that a match counts 1 while mismatches have no costs. With m general values for either matches or mismatches can be defined (for moreoptions seeBiopython's API). The second letter decodes the cost for gaps; x means no gap costs at all, with s different penalties for opening and extending a gap can be assigned. So, globalxx means that only matches between both sequences are counted.

x	No parameters. Identical characters have score of 1, otherwise 0.
m	A match score is the score of identical chars, otherwise mismatch score.
d	A dictionary returns the score of any pair of characters.
c	A callback function returns scores.

The gap penalty parameters are:

x	No gap penalties.
s	Same open and extend gap penalties for both sequences.

d The sequences have different open and extend gap penalties.

c A callback function returns the gap penalties.

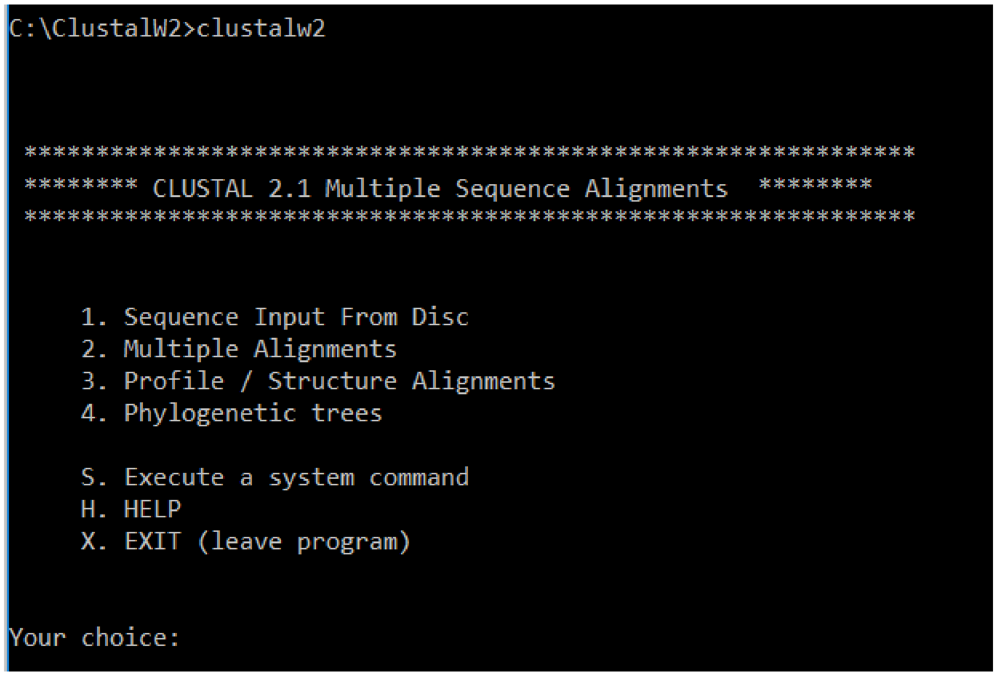
Each alignment is a tuple consisting of the two aligned sequences, the score, the start and the end positions of the alignment (in global alignments the start is always 0 and the end the length of the alignment). Bio.pairwise2 has a function format_alignment for a nicer printout:

```
from Bio import pairwise2
from Bio import SeqIO
from Bio.SubsMat.MatrixInfo import blosum62
seq1 = SeqIO.read("alpha.faa", "fasta")
seq2 = SeqIO.read("beta.faa", "fasta")
alignments = pairwise2.align.globalds\
(seq1.seq, seq2.seq, blosum62, -10, -0.5)
len(alignments)
print(pairwise2.format_alignment(*alignments[0]))
```

ClustalW

ClustalW is a popular command line tool for multiple sequence alignment (there is also a graphical interface called ClustalX). Biopython's Bio.Align.Applications module has a wrapper for this alignment tool (and several others). Before trying to use ClustalW from within Python. First, we will look at the ClustalW tool by hand at the command line, to be familiar with the other options. You'll find the Biopython wrapper is very faithful to the actual command line API.

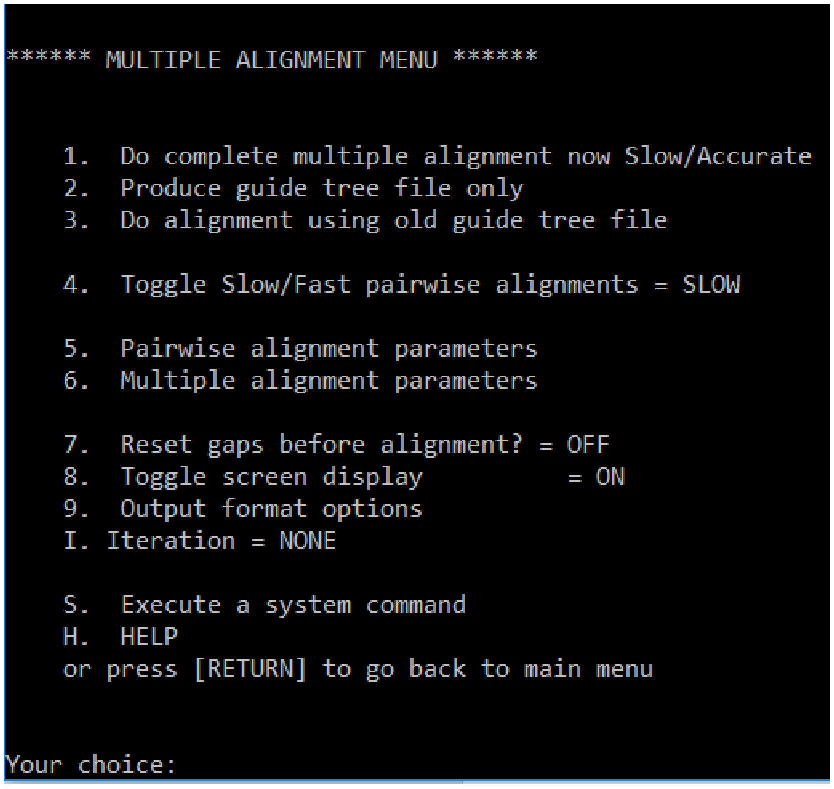
Clutalw can be downloaded and installed in windows very easily. After installing move to the installation directory in command prompt and execute the executable. You will see the output like this.



```
C:\ClustalW2>clustalw2



 **************************************************************
 ******** CLUSTAL 2.1 Multiple Sequence Alignments  ********
 **************************************************************


     1. Sequence Input From Disc
     2. Multiple Alignments
     3. Profile / Structure Alignments
     4. Phylogenetic trees

     S. Execute a system command
     H. HELP
     X. EXIT (leave program)


Your choice:
```

Here we are working with MSA, so we our choice is 2. The program displays the following output.

We can select different options for out alignment.



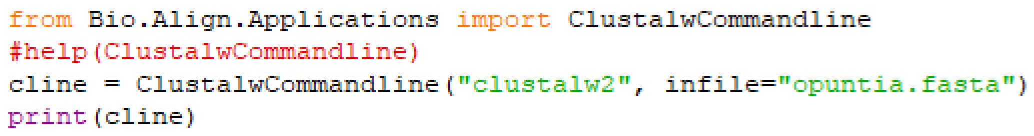
```
****** MULTIPLE ALIGNMENT MENU ******


    1.  Do complete multiple alignment now Slow/Accurate
    2.  Produce guide tree file only
    3.  Do alignment using old guide tree file

    4.  Toggle Slow/Fast pairwise alignments = SLOW

    5.  Pairwise alignment parameters
    6.  Multiple alignment parameters

    7.  Reset gaps before alignment? = OFF
    8.  Toggle screen display          = ON
    9.  Output format options
    I. Iteration = NONE

    S.  Execute a system command
    H.  HELP
    or press [RETURN] to go back to main menu


Your choice:
```

To run an alignment using clustalw following command is used.

Clustalw2 input file name

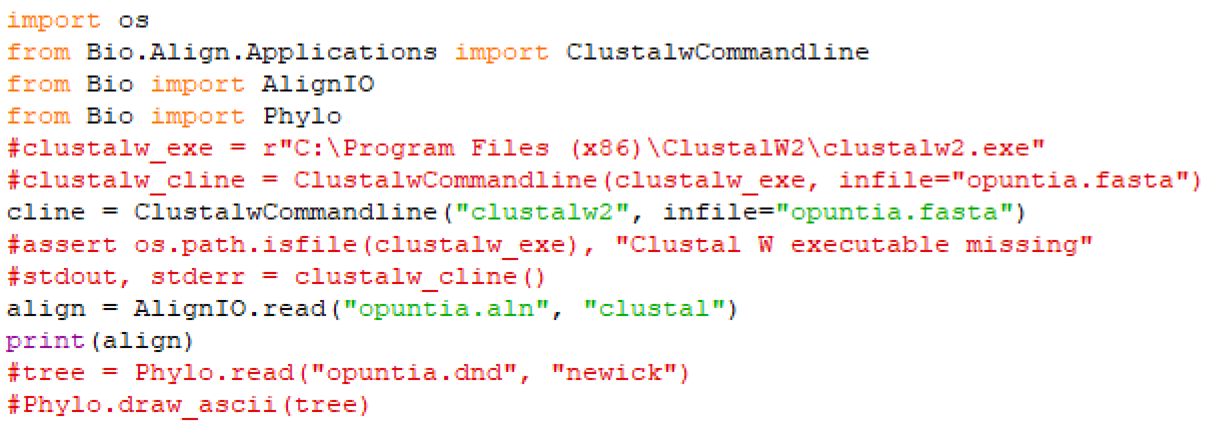
The program will create 2 files with extensions .aln and dnd containing the alignment and the phylogenetic tree respectively.

For the most basic usage, all you need is to have a FASTA input file, such asopuntia.fasta (available online or in the Doc/examples subdirectory of the Biopython source code). This is a small FASTA file containing seven prickly-pear DNA sequences (from the cactus family Opuntia). By default, ClustalW will generate an alignment and guide tree fil with names based on the input FASTA file, in this case opuntia.aln and opuntia.dnd, but you can override this or make it explicit.



```
from Bio.Align.Applications import ClustalwCommandline
#help(ClustalwCommandline)
cline = ClustalwCommandline("clustalw2", infile="opuntia.fasta")
print(cline)
```

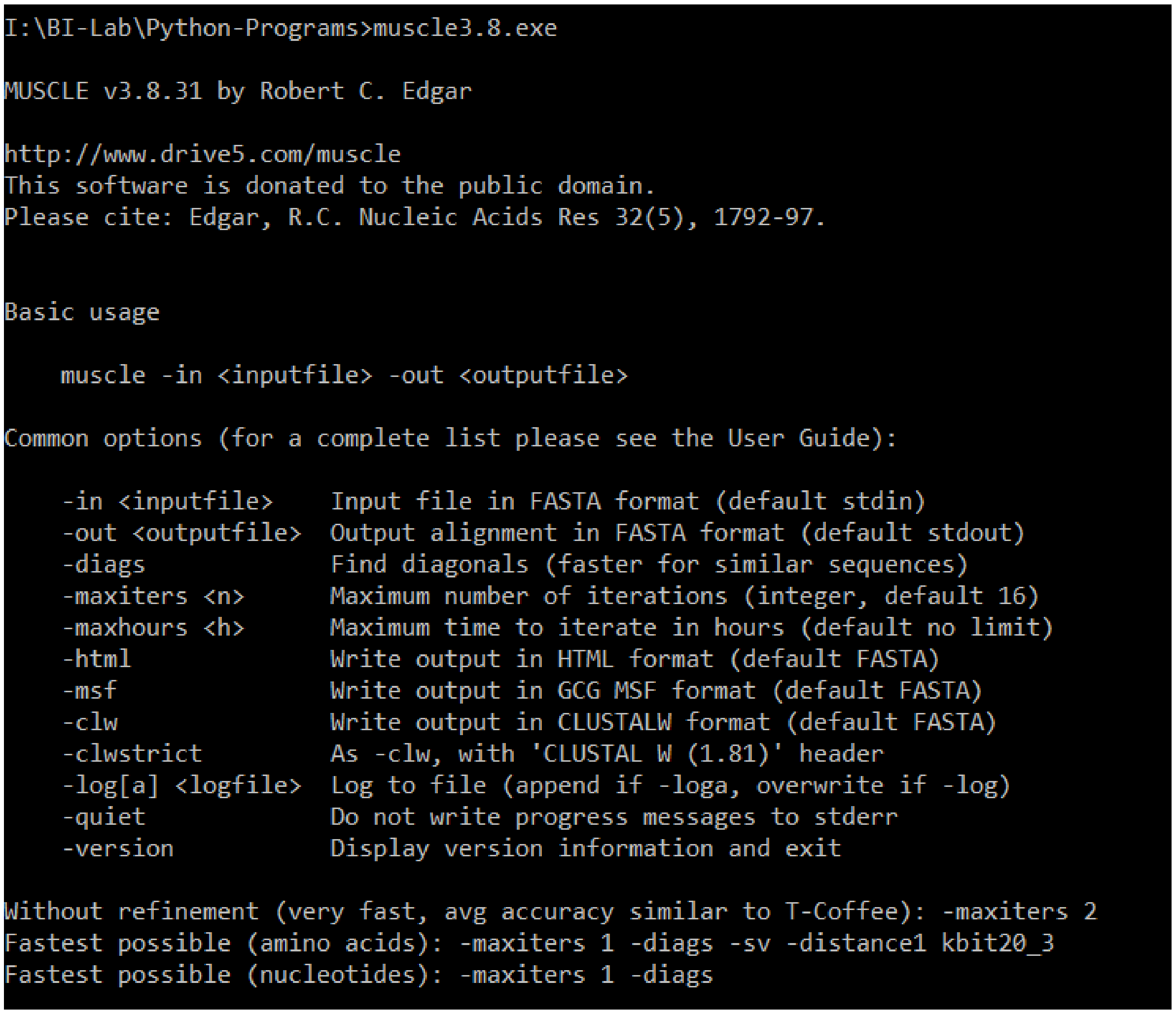
Notice here we have given the executable name as clustalw2, indicating we have version two installed, which has a different file name to version one (clustalw, the default). Fortunately, both versions support the same set of arguments at the command line (and indeed, should be functionally identical).



```
import os
from Bio.Align.Applications import ClustalwCommandline
from Bio import AlignIO
from Bio import Phylo
#clustalw_exe = r"C:\Program Files (x86)\ClustalW2\clustalw2.exe"
#clustalw_cline = ClustalwCommandline(clustalw_exe, infile="opuntia.fasta")
cline = ClustalwCommandline("clustalw2", infile="opuntia.fasta")
#assert os.path.isfile(clustalw_exe), "Clustal W executable missing"
#stdout, stderr = clustalw_cline()
align = AlignIO.read("opuntia.aln", "clustal")
print(align)
#tree = Phylo.read("opuntia.dnd", "newick")
#Phylo.draw_ascii(tree)
```

MUSCLE

MUSCLE is a more recent multiple sequence alignment tool than ClustalW, and Biopython also has a wrapper for it under the Bio.Align.Applications module. As before, we will first run the tool by using MUSCLE from the command line before trying it from within Python, as the Biopython wrapper is very faithful to the actual command line API. Muscle can be executed by providing the command.


```
I:\BI-Lab\Python-Programs>muscle3.8.exe

MUSCLE v3.8.31 by Robert C. Edgar

http://www.drive5.com/muscle
This software is donated to the public domain.
Please cite: Edgar, R.C. Nucleic Acids Res 32(5), 1792-97.


Basic usage

    muscle -in <inputfile> -out <outputfile>

Common options (for a complete list please see the User Guide):

    -in <inputfile>    Input file in FASTA format (default stdin)
    -out <outputfile>  Output alignment in FASTA format (default stdout)
    -diags             Find diagonals (faster for similar sequences)
    -maxiters <n>      Maximum number of iterations (integer, default 16)
    -maxhours <h>      Maximum time to iterate in hours (default no limit)
    -html              Write output in HTML format (default FASTA)
    -msf               Write output in GCG MSF format (default FASTA)
    -clw               Write output in CLUSTALW format (default FASTA)
    -clwstrict         As -clw, with 'CLUSTAL W (1.81)' header
    -log[a] <logfile>  Log to file (append if -loga, overwrite if -log)
    -quiet             Do not write progress messages to stderr
    -version           Display version information and exit

Without refinement (very fast, avg accuracy similar to T-Coffee): -maxiters 2
Fastest possible (amino acids): -maxiters 1 -diags -sv -distance1 kbit20_3
Fastest possible (nucleotides): -maxiters 1 -diags
```

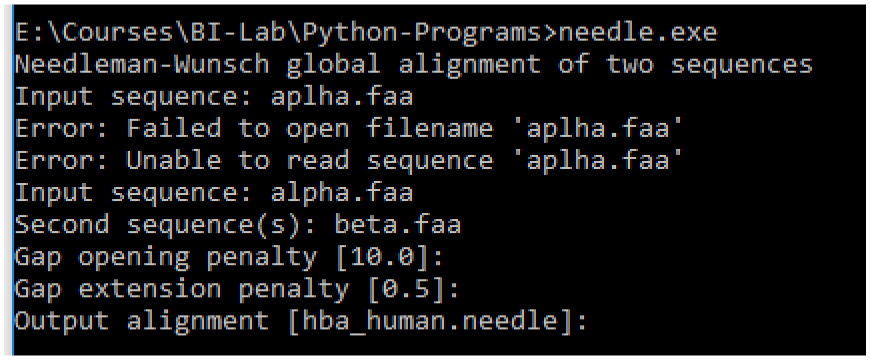
Muscle.exe -in input file name, -out output file name (optional format)

```
from io import StringIO
from Bio import AlignIO
from Bio.Align.Applications import MuscleCommandline
muscle_exe = "muscle3.8.exe"
cline = MuscleCommandline(muscle_exe,\
input="opuntia.fasta", out="opuntia.txt")
stdout, stderr = cline()
```

EMBOSS needle and water

The EMBOSS suite includes the water and needle tools for Smith-Waterman algorithm local alignment, and Needleman-Wunsch global alignment. The tools share the same style interface, so switching between the two is trivial, we’ll just use needle here. Suppose you want to do a global pairwise alignment between two sequences, prepared in FASTA format as follows:

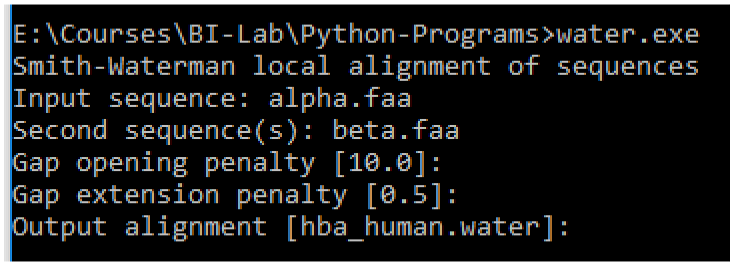
Following is an example execution of Needle Program in Emboss



```
E:\Courses\BI-Lab\Python-Programs>needle.exe
Needleman-Wunsch global alignment of two sequences
Input sequence: aplha.faa
Error: Failed to open filename 'aplha.faa'
Error: Unable to read sequence 'aplha.faa'
Input sequence: alpha.faa
Second sequence(s): beta.faa
Gap opening penalty [10.0]:
Gap extension penalty [0.5]:
Output alignment [hba_human.needle]:
```

```
from Bio.Emboss.Applications import NeedleCommandline
needle_exe="needle.exe"
needle_cline = NeedleCommandline(needle_exe,asequence="alpha.faa",\
bsequence="beta.faa",gapopen=10, gapextend=0.5, outfile="needle.txt")
print(needle_cline)
stdout, stderr = needle_cline()
```

Waterman program can be executed from command line as follows.



```
E:\Courses\BI-Lab\Python-Programs>water.exe
Smith-Waterman local alignment of sequences
Input sequence: alpha.faa
Second sequence(s): beta.faa
Gap opening penalty [10.0]:
Gap extension penalty [0.5]:
Output alignment [hba_human.water]:
```

```
from Bio.Emboss.Applications import WaterCommandline
water_exe="water.exe"
water_cline = WaterCommandline(water_exe,asequence="alpha.faa",\
bsequence="beta.faa",gapopen=10, gapextend=0.5, outfile="water.txt")
print(water_cline)
stdout, stderr = water_cline()
```

Reading the alignments

```
from Bio import AlignIO
align = AlignIO.read("needle.txt", "emboss")
print(align)
```

LAB #7 BLAST

Statement Purpose

The purpose of this lab is to discuss Basic Local Sequence Alignment Tool (BLAST) in detail. The activity will cover both online and biopython version of the tool.

Activity Outcomes

- Running BLAST over the Internet
- Running BLAST locally
- Parsing BLAST output
- The BLAST record class

Running BLAST over the Internet

We use the function qblast() in the Bio.Blast.NCBIWWW module to call the online version of BLAST. This has three non-optional arguments:

- The first argument is the blast program to use for the search, as a lower-case string. The options and descriptions of the programs are available at https://blast.ncbi.nlm.nih.gov/Blast.cgi. Currently qblast only works with blastn, blastp, blastx, tblast and tblastx.
- The second argument specifes the databases to search against. Again, the options for this are available on the NCBI Guide to BLAST ftp://ftp.ncbi.nlm.nih.gov/pub/factsheets/HowTo_BLASTGuide. pdf.
- The third argument is a string containing your query sequence. This can either be the sequence itself, the sequence in fasta format, or an identifier like a GI number.

The qblast function also take a number of other option arguments which are basically analogous to the different parameters you can set on the BLAST web page. We'll just highlight a few of them here:

The argument url_base sets the base URL for running BLAST over the internet. By default it connects to the NCBI, but one can use this to connect to an instance of NCBI BLAST running in the cloud. Please refer to the documentation for the qblast function for further details.

The qblast function can return the BLAST results in various formats, which you can choose with the optional format_type keyword: "HTML", "Text", "ASN.1", or "XML". The default is "XML", as that is the format expected by the parser.

The argument expect sets the expectation or e-value threshold.

```
from Bio.Blast import NCBIWWW
result_handle = NCBIWWW.qblast("blastn", "nt", "8332116")
```

Alternatively, if we have our query sequence already in a FASTA formatted file, we just need to open the file and read in this record as a string, and use that as the query argument:

```
from Bio.Blast import NCBIWWW
fasta_string = open("m_cold.fasta").read()
result_handle = NCBIWWW.qblast("blastn", "nt", fasta_string)
```

We could also have read in the FASTA file as a SeqRecord and then supplied just the sequence itself:

```
from Bio.Blast import NCBIWWW
from Bio import SeqIO
record = SeqIO.read("m_cold.fasta", format="fasta")
result_handle = NCBIWWW.qblast("blastn", "nt", record.seq)
```

Supplying just the sequence means that BLAST will assign an identifier for your sequence automatically. You might prefer to use the SeqRecord object's format method to make a FASTA string (which will include the existing identifier):

```
from Bio.Blast import NCBIWWW
from Bio import SeqIO
record = SeqIO.read("m_cold.fasta", format="fasta")
result_handle = NCBIWWW.qblast("blastn", "nt", record.format("fasta"))
```

Whatever arguments you give the qblast() function, you should get back your results in a handle object (by default in XML format). The next step would be to parse the XML output into Python objects representing the search results. We need to be a bit careful since we can use result_handle.read() to read the BLAST output only once, calling result_handle.read() again returns an empty string.
After doing this, the results are in the file my_blast.xml and the original handle has had all its data extracted (so we closed it). However, the parse function of the BLAST parser takes a file-handle-like object, so we can just open the saved file for input:

```
from Bio.Blast import NCBIWWW
from Bio import SeqIO
record = SeqIO.read("m_cold.fasta", format="fasta")
result_handle = NCBIWWW.qblast("blastn", "nt", record.format("fasta"))
with open("my_blast.xml", "w") as out_handle:
    out_handle.write(result_handle.read())
result_handle.close()
result_handle = open("my_blast.xml")
blast_record = NCBIXML.read(result_handle)
blast_records = NCBIXML.parse(result_handle)#for multiple queries
```

Running BLAST locally

Running BLAST locally has at least major two advantages:

- Local BLAST may be faster than BLAST over the internet;
- Local BLAST allows you to make your own database to search for sequences against.

Unfortunately, there are some major drawbacks too, installing all the bits and getting it setup right takes some effort:

- Local BLAST requires command line tools to be installed.
- Local BLAST requires (large) BLAST databases to be setup (and potentially kept up to date).

LAB #8 Accessing NCBI's Entrez databases

Statement Purpose

The purpose of this lab is to discuss NCBI's Entrez databases in detail. The activity will cover different features available in biopython.

Activity Outcomes

- EInfo: Obtaining information about the Entrez databases
- ESearch: Searching the Entrez databases
- EPost: Uploading a list of identifiers
- ESummary: Retrieving summaries from primary IDs
- EFetch: Downloading full records from Entrez
- ELink: Searching for related items in NCBI Entrez
- EGQuery: Global Query - counts for search terms
- ESpell: Obtaining spelling suggestions
- Parsing huge Entrez XML files
- HTML escape characters
- Specialized parsers

Introduction

Entrez (https://www.ncbi.nlm.nih.gov/Web/Search/entrezfs.html) is a data retrieval system that provides users access to NCBI's databases such as PubMed, GenBank, GEO, and many others. You can access Entrez from a web browser to manually enter queries, or you can use Biopython's Bio.Entrez module for programmatic access to Entrez. The latter allows you for example to search PubMed or download GenBank records from within a Python script.

EInfo: Obtaining information about the Entrez databases

EInfo provides field index term counts, last update, and available links for each of NCBI's databases. In addition, you can use EInfo to obtain a list of all database names accessible through the Entrez utilities:

```
from Bio import Entrez
Entrez.email = "your email"
handle = Entrez.einfo()
result = handle.read()
handle.close()
print(result)
```

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:

```
from Bio import Entrez
Entrez.email = "your email"
handle = Entrez.einfo()
record = Entrez.read(handle)
handle.close()
print(record.keys())
print(record["DbList"])
```

For each of these databases, we can use EInfo again to obtain more information:

```
from Bio import Entrez
Entrez.email = "your email"
handle = Entrez.einfo(db="pubmed")
record = Entrez.read(handle)
print(record["DbInfo"]["Description"])
print(record["DbInfo"]["Count"])
print(record["DbInfo"]["LastUpdate"])
```

ESearch: Searching the Entrez databases

To search any of these databases, we use Bio.Entrez.esearch(). For example, let's search in PubMed for publications related to Biopython:

```
from Bio import Entrez
Entrez.email = "your email"
handle = Entrez.esearch(db="pubmed", term="biopython")
record = Entrez.read(handle)
print(record["IdList"])
```

EPost: Uploading a list of identifiers

EPost uploads a list of UIs for use in subsequent search strategies. To give an example of when this is useful, suppose you have a long list of IDs you want to download using EFetch (maybe sequences, maybe citations (anything). When you make a request with EFetch your list of IDs, the database etc., are all turned into a long URL sent to the server. If your list of IDs is long, this URL gets long, and long URLs can break (e.g. some proxies don't cope well).

Instead, you can break this up into two steps, first uploading the list of IDs using EPost (this uses an \HTML post" internally, rather than an \HTML get", getting around the long URL problem). With the history support, you can then refer to this long list of IDs and download the associated data with EFetch. Let's look at a simple example to see how EPost works uploading some PubMed identifiers:

```
from Bio import Entrez
Entrez.email = "your email"
id_list = ["19304878", "18606172", "16403221", "16377612",
           "14871861", "14630660"]
print(Entrez.epost("pubmed", id=",".join(id_list)).read())
```

The returned XML includes two important strings, QueryKey and WebEnv which together define your history session. You would extract these values for use with another Entrez call such as EFetch:

```python
from Bio import Entrez
Entrez.email = "your email"
id_list = ["19304878", "18606172", "16403221",
           "16377612", "14871861", "14630660"]
search_results = Entrez.read(Entrez.epost("pubmed",
                                           id=",".join(id_list)))
webenv = search_results["WebEnv"]
query_key = search_results["QueryKey"]
```

ESummary: Retrieving summaries from primary IDs

ESummary retrieves document summaries from a list of primary IDs. In Biopython, ESummary is available as Bio.Entrez.esummary(). Using the search result above, we can for example find out more about the journal with ID 30367:

```python
from Bio import Entrez
Entrez.email = "your email"
handle = Entrez.esummary(db="nlmcatalog",
                         id="101660833")
record = Entrez.read(handle)
info = record[0]["TitleMainList"][0]
print("Journal info\nid: {}\nTitle: {}"
      .format(record[0]["Id"], info["Title"]))
```

EFetch: Downloading full records from Entrez

EFetch is what you use when you want to retrieve a full record from Entrez. This covers several possible databases, as described on the mainEFetch Help page. For most of their databases, the NCBI support several different file formats. Requesting a specific file format from Entrez using Bio.Entrez.efetch() requires specifying the rettype and/or retmode optional arguments. The different combinations are described for each database type on the pages linked to on NCBI efetch webpage.

One common usage is downloading sequences in the FASTA or GenBank/GenPept plain text formats. we can download GenBank record EU490707 using Bio.Entrez.efetch:

```python
from Bio import Entrez
Entrez.email = "A.N.Other@example.com"
handle = Entrez.efetch(db="nucleotide",
                       id="EU490707", rettype="gb",
                       retmode="text")
print(handle.read())
```

EGQuery: Global Query - counts for search terms

EGQuery provides counts for a search term in each of the Entrez databases (i.e. a global query). This is particularly useful to find out how many items your search terms would find in each database without actually, performing lots of separate searches with ESearch. In this example, we use Bio.Entrez.egquery() to obtain the counts for “Biopython":

```python
from Bio import Entrez
Entrez.email = "your email"
handle = Entrez.egquery(term="biopython")
record = Entrez.read(handle)
for row in record["eGQueryResult"]:
    print(row["DbName"], row["Count"])
```

ESpell: Obtaining spelling suggestions

ESpell retrieves spelling suggestions. In this example, we use Bio.Entrez.espell() to obtain the correct spelling of Biopython:

```python
from Bio import Entrez
Entrez.email = "your email"
handle = Entrez.espell(term="biopythooon")
record = Entrez.read(handle)
record["Query"]
'biopythooon'
print(record["CorrectedQuery"])
```

LAB #9 Phylogenetic Trees

Statement Purpose

The purpose of this lab is to discuss phylogenetic trees in detail. The activity will cover different methods for phylogenetic analysis.

Activity Outcomes

- EInfo: Obtaining information about the Entrez databases

Introduction

The Bio.Phylo module was introduced in Biopython 1.54. Following the lead of SeqIO and AlignIO, it aims to provide a common way to work with phylogenetic trees independently of the source data format, as well as a consistent API for I/O operations.

To get acquainted with the module, let's start with a tree that we've already constructed, and inspect it a few different ways. Then we'll colorize the branches, to use a special phyloXML feature, and finally save it.

Create a simple Newick file named simple.dnd using your favorite text editor, and add the following data.

```
(((A,B),(C,D)),(E,F,G));
```

This tree has no branch lengths, only a topology and labelled terminals. (we can use the file we have created in previous labs)