CPSC 301: Computing in the Life Sciences
Lecture Notes 15: Biopython

2017-03-30

Jessica Dawson
jqdawson@cs.ubc.ca
http://www.cs.ubc.ca/~jqdawson

University of British Columbia
Department of Computer Science

2016 W2
Objectives

• After competing this unit you will be able to read and modify simple programs that use Biopython to
  – create and manipulate biological sequences using the BioPython Sequences and Sequence Records
  – find and retrieve sequences from local files or from public biological databases over the Internet
  – store sequences in files in most popular formats
  – given a sequence, search known public databases to retrieve similar sequences to that sequence using the popular tool BLAST and process the results.
What is Biopython

• Biopython is a collection of Python modules for biological data.
• Has data structures to deal with a variety of file formats, including
  – FASTA, GenBank, SwissProt, ExPASy, Blast output, and more
• Code to allow Python programs to connect to popular sites like NCBI and ExPASy
• Interfaces to popular programs like BLAST, Clustalw, etc.
• A sequence class to represent bio-sequences and operations like transcription and translation
• Code for dealing with classification, alignments and more
• Sources of information:
  – wiki,
  – application programming interface (API)
    • Explains classes, methods, functions, etc.
• To use Biopython from home you need to install it following the instructions listed in lab 6.
BioPython’s Sequences

• See Biopython tutorial chapter 3
• Inside the Bio directory of a Biopython installation, the module Seq.py contains the class Seq that represents biological sequences (immutable objects)
• An instance of Seq contains two pieces of information: a sequence and an alphabet from which that sequence is drawn
  – The alphabet defines the characters which are valid to appear in the sequence
  – The available alphabets are defined in the Bio.Alphabet module.
• Sequences are not strings. They have most of the same methods as strings, plus more
  – String methods: length, slicing, concatenation, change case, etc.
  – Other methods: transcribe, translate, reverse complement, etc.
Sequence Examples

• Defining a sequence of any alphabet:

```python
>>> from Bio.Seq import Seq
>>> s = Seq("AACGCGCATTTG")
>>> s
Seq('AACGCGCATTTG', Alphabet())
```  
```
>>> s.alphabet
Alphabet()
```  

• Defining a sequence of a specific alphabet:

```python
>>> from Bio.Seq import Seq
>>> from Bio import Alphabet
>>> s1 = Seq("AACGCGCATTTG", Alphabet.generic_dna)
>>> s1
Seq('AACGCGCATTTG', DNAAlphabet())
```
Sequence Operations

• Can apply most of string operations. If s is a sequence:
  – \( s[i] \) returns the i-th character
  – \( s[i:j] \) returns a slice of s
  – \( \text{len}(s) \) returns the length
  – \( s.\text{count}("C") \) returns the number of C's in s
  – \( \text{str}(s) \) or \( s.\text{tostring}() \) returns the string part of s
  – \( s1 + s2 \) concatenates two sequences of the same alphabet (will throw an error otherwise).
  – \( s.\text{upper}(), s.\text{lower}(), \) etc work as well

• Special operation for bio sequences only:
  – \( s.\text{complement}() \) returns the complement of s
  – \( s.\text{reverse_complement}() \) returns the rev. complement of s
  – \( s.\text{transcribe}() \) returns the transcription of s (switches T \( \rightarrow \) U)
  – \( s.\text{transate}() \) returns the translation of s
    translation tables are in \( \text{Bio.Data.CodonTable} \)

• If you need to make a sequence into just string:
  – \( \text{str}(s) \): returns a copy of the sequence as string only
  – \( \text{print}(s) \): outputs the sequence to the console. Automatically converts a Seq object to a string.
Sequence Records

• A sequence record contains a sequence and other information about the sequence
• Is defined by the `SeqRecord` class in the `Bio.SeqRecord` module
• Each sequence record has as attributes:
  – `seq` the sequence (is a Seq object)
  – `id` the primary id for the sequence
  – `name` the common name for the sequence
  – `description` a string describing the sequence
  – and more
• Recall that we access attributes using the syntax `object.attribute`
• A constructor allows users to create a sequence record from a sequence and set the rest of the attributes later
• Usually, we use `Bio.SeqIO` to read and create SeqRecords from FASTA or GenBank files
Using SeqIO to read Sequences

- Bio.SeqIO module allows users to retrieve sequences from files
- To read sequence from a FASTA file with only one sequence we usually do:
  ```python
  >>> from Bio import SeqIO
  >>> record = SeqIO.read("myfile.fasta", "fasta")
  >>> s = record.seq
  ```
  then s is the sequence in that file
- To read a sequence from a GenBank file with only one sequence we do:
  ```python
  >>> from Bio import SeqIO
  >>> record = SeqIO.read("myfile.gb", "genbank")
  >>> s = record.seq
  ```
  then s is the sequence in that file

Q1
Reading Multiple Sequences

• To read more than one sequence from a file we use Bio.SeqIO.parse():

```python
from Bio import SeqIO
records = SeqIO.parse("myfile.fasta", "fasta")
```

– Now records is a collection of SeqRecords, one for each sequence in the file

• We can access one record at a time using

```python
rec = records.next()
```

Or, we can go over all of them using:

```python
for rec in records:
    print(rec.id, rec.seq)
```
Getting Sequences from the Internet

• NCBI has an API called Entrez that allow users to connect to their databases and download sequences and other information
• For instance, to download a FASTA record for a sequence with known id we can write:
  
  ```python
  from Bio import Entrez
  from Bio import SeqIO
  handle = Entrez.efetch(db="nucleotide", rettype="fasta", id="6273291")
  seq_record = SeqIO.read(handle, "fasta")
  handle.close()
  ```

• efetch() returns something like a file that can be opened by the SeqIO methods. For instance, to download many sequences:
  
  ```python
  handle = Entrez.efetch(db="nucleotide", rettype="gb", id="6273291,6273290,6273289")
  for rec in SeqIO.parse(handle, "gb"):
    print(rec.id, rec.seq)
  handle.close()
  ```

• SeqIO.parse() returns a collection of sequence records
Writing Sequences to a File

• We can download sequence records and write them to a file using the `Bio.SeqIO.write()` method:

```python
from Bio import SeqIO
from Bio import Entrez
handle = Entrez.efetch(db="nucleotide", rettype="fasta", id="6273291,6273290,6273289")
records = SeqIO.parse(handle, "fasta")
outfile = open("orchid.fasta", "w")
SeqIO.write(records, outfile, "fasta")
handle.close()
outfile.close()
```

• All three sequences are written in the orchid.fasta file
Change log

• Using SeqIO to read sequences: Fixed typo in ”myfile.gb”
Finding Similar Sequences - BLAST

• Given a sequence, we can use the Basic Local Alignment Search Tool (BLAST) to search for similar sequences in one or more databases

• A BLAST session consists of two steps
  1. running BLAST on our sequence and get the matching results
  2. processing the results for further analysis

• To use BLAST over the Internet, we can call the method `qblast()` in the module `Bio.Blast.NCBIWWW`

• For instance, to compare a nucleotide sequence given by its id against the non-redundant databases we can write:
  ```python
  from Bio.Blast import NCBIWWW
  result = NCBIWWW.qblast("blastn", "nr", "1705278")
  ```

• You should check section 7.1 of the tutorial for other options for the `qblast` parameters
If the sequence we want to blast is in a file we can use:

```python
from Bio.Blast import NCBIWWW
fstring = open("fruitfly.fasta" ).read()
result = NCBIWWW.qblast("blastn", "nr", fstring)
```

The result contains a collection of sequence alignments between the given sequence and the matching sequences which blast found. But this information is in XML format.

To process the result we need first to create a blast record (an object of the Blast class ) with the results using the `NCBIXML.read` method:

```python
from Bio.Blast import NCBIXML
blast_record = NCBIXML.read(result)
```

Then we can process the blast record
Sometimes we'd like to save the blast results to a file before we process them. In this case we can do:

```python
result = NCBIWWW.qblast("blastn", "nr", "1705278")
result_file = open("blastresult.xml", "w")
result_file.write(result.read())
result_file.close()
result.close()
```

The results are in an XML file. To process the results we need to open the xml file and read the results in using the `NCBIXML.read` method:

```python
result_file = open("blastresult.xml")
from Bio.Blast import NCBIXML
blast_record = NCBIXML.read(result_file)
```

We now need to know what are the components of a blast record
Blast Records

• The structure of a blast record is shown on the figure on the right (taken from the tutorial).
• A Blast object consists of a description and a list of alignments (matching sequences).
• Each Alignment object has a title, a length and list of high scoring pairs (HSP), i.e. matching segments.
• Each HSP object has the set of attributes like:
  – HSP.score: The BLAST score for the segment match.
  – HSP.expect: A number. The e-value for the segment match.
  – HSP.query: A string. The sequence that you used to query the search term.
  – HSP.match: A string. The corresponding match sequence between query and subject.
  – HSP.sbjct: A string. The subject sequence aligned to the query sequence.
BLAST – Processing a Record

• Suppose we want to print the best HSPs, that is, those whose expect values are 0.0 (i.e. no of matches by chance). We can write:

```python
for alignment in blast_record.alignments:
    for hsp in alignment.hsps:
        if hsp.expect == 0.0:
            print('****Alignment****')
            print('sequence:', alignment.title)
            print('length:', alignment.length)
            print('e value:', hsp.expect)
            print(hsp.query[0:75] + '...')
            print(hsp.match[0:75] + '...')
            print(hsp.sbjct[0:75] + '...')
```