BCH2203 Python - 7. Biopython

Ramses van Zon

28 February 2024

Biopython

- Biopython is a package of freely available Python tools for (mostly) genomic data.
- Motivation: dealing with sequences as text requires a lot of explicit manipulation and coding, and much of that code could be reused in other biomolecular computations.
- Well documented:
  biopython.org
  Biopython Tutorial and Cookbook
- Latest release is Biopython 1.83 (January 10, 2024)
  Requires Python version 3.8 or higher.

Tools provided by Biopython

- Working with sequences:
  Translation, transcription, annotation, locations, features
- Reading and writing biological data files
- Connecting to online biological databases
- Sequence alignments
- Interface with external alignment tools, like ClustalW, Muscle, EMBOSS, BLAST.
- Working with crystal structures of biological macromolecules.
- Population genetics
- Working with phylogenetic trees
- Motif analysis
- Analysis of phenotypic data
- Some machine learning methods (cluster analysis, supervised learning)

Installing Biopython

On the SciNet Teach cluster:

```
$ ssh -Y USERNAME@teach.scinet.utoronto.ca
$ module load python/3
$ virtualenv --system-site-packages biopython183 # once
$ source biopython183/bin/activate
$ pip install biopython==1.83 # once
```

On your own computer with virtual environments

Assuming you have Python >= 3.8, you can do as above, but without `ssh` or `module load` commands.

```
$ virtualenv --system-site-packages biopython183 # once
$ source biopython183/bin/activate
$ pip install biopython==1.83 # once
```

On your own computer with conda environments

Conda environments can work as well:

```
conda create -n biopython183 biopython==1.83 # once
conda activate biopython183
```

Why specific version?

- You may think you always want the latest version of a software or python package.
- However, versions are not always backwards compatible.
- For reproducibility, always record which versions of applications, libraries, and packages you used.
- For python packages, the virtualenv (or conda env) allows you to keep the environment as it.
- So you could several environments with different versions of packages.
- This is why we encoded the version of biopython in the name of the virtual environment.

Sequences in BioPython

Working with Sequences

Biopython has Seq objects, whose definition are in the submodule Bio.Seq.

They are essentially strings with some extra methods.

```
>>> from Bio.Seq import Seq
>>> seq = Seq('AGTACACTGTAG')
>>> type(seq)
Bio.Seq.Seq
>>> seq
Seq('AGTACACTGTAG')
>>> print(seq)
AGTACACTGTAG
>>> seq.complement()
Seq('TCATGTGACCA')
>>> seq.reverse_complement()
Seq('ACCAGTGTACT')
```

Note: earlier versions of Biopython (<=1.77) used the concept of an Alphabet object that was always associated with any sequence. Newer version do not; in those cases where it matters, functions will have additional arguments to indicate whether it is a “DNA”, “RNA” or “protein” sequence.

Sequences are like strings

Almost all string manipulations are possible with Seq.

For example:

```
>>> seq = Seq('AGTACACTGTAG')
>>> seq[0:4]
Seq('AGTA')
>>> seq*2
Seq('AGTACACTGTAGAGTACACTGTAG')
>>> seq.find("ACA")
3
>>> seq.split("ACA")
[Seq('AGT'), Seq('CTGGT')]
```

For any string manipulation that might not work for Seq, you can always convert the Seq to a string:

```
>>> seq = Seq('AGTACAcTGGTAG')
>>> seq.swapcase()
AttributeError  - Traceback (most recent call last)
...
AttributeError: 'Seq' object has no attribute 'swapcase'
>>> str(seq).swapcase()
'agtacaCtggtag'
```

Note the difference between

1. typing the name of an object in an interactive python sessions; and
2. using the print statement with that object.

#1 gives you more information about the object in a "pythonic" form, while #2 just prints its content.

When used in a script, #1 does not print or do anything.

Transcription and Translation

Biopython know how to transcribe DNA to RNA and translate it to protein sequences.

```
>>> seq = Seq('AGTACACTGTAG')
>>> rna = seq.transcribe()   # assumes sequence is coding dna
>>> rna
Seq('AGUACACUGGU')
```

```
>>> prot = seq.translate()  # assumes sequence is coding dna or rna
>>> prot
Seq('STL*')
```

This uses a one-letter alphabet for the protein sequence. If you want the 3-letter abbreviation, you can use:

```
>>> from Bio.SeqUtils import seq3
>>> seq3(prot)
'SerThrTrpTer'
```

The Seq object does not distinguish the kind sequence, so sometimes we need to be specific, e.g.

```
>>> from Bio.SeqUtils import molecular_weight
>>> molecular_weight(seq)
3749.4022000000004
>>> molecular_weight(prot[:-1])
...
ValueError: 'S' is not a valid unambiguous letter for DNA
>>> molecular_weight(prot[:-1], "protein")
319.3541
```

SeqRecord

Annotation, locations, features

The SeqRecord provides a standard way to add information to a sequence. It contains:

- .seq

  The sequence itself, typically a Seq object.

- .id

  The primary ID (a string) to identify the sequence. In most cases something like an accession number.

- .name

  A “common” name/id for the sequence.

- .description

  A human readable description or expressive name for the sequence.

- .letter_annotations

  A dict of information about the letters in the sequence. Keys are the name of the information, and the information is as a Python sequence with the same length as the sequence. Used for quality scores, secondary structure, ...

- .annotations

  A dict additional information about the sequence.

- .features

  A list of SeqFeature objects with more structured information about the features on a sequence.

- .dbxrefs

  A list of database cross-references as strings.

Creating SeqRecords

You can create SeqRecords yourself, e.g.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> seq = Seq('AGTACACTGTAG')
>>> seqrec = SeqRecord(seq)
```

You do not need to provide all the data at creation time, you can fill those in later:

```
>>> seqrec.id = 'ABCDEFG'
>>> seqrec.description = 'Nonsensical sequence'
>>> seqrec
SeqRecord(seq=Seq('AGTACACTGTAG'), id='ABCDEFG', name='<unknown name>', description='Nonsensical sequence', dbxrefs=[])
>>> print(seqrec)
ID: ABCDEFG
Name: <unknown name>
Description: Nonsensical sequence
Number of features: 0
Seq('AGTACACTGTAG')
```

More commonly, you'll get the SeqRecord information from a data file that Biopython can read in for you, filling in the information into SeqRecords using Bio.SeqIO .

Bio.SeqIO

- Bio.SeqIO aims to provide a simple uniform interface to input and output of sequence file formats.
- It always returns SeqRecord's.
- Can deal with files containing multiple sequences.
- Many file formats are supported:

  `abi ace clustal fasta fastq genbank ...`

- https://biopython.org/wiki/SeqIO

Example: Reading a FASTA file into a SeqRecord

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("chromosome1.fa", "fasta")
>>> type(record)
<class 'Bio.SeqRecord.SeqRecord'>
>>> print(record)
ID: 1
Name: 1
Description: 1 dna:chromosome chromosome:Galgal4:1:1:1952767
Number of features: 0
Seq('CCGACCAGTTGTAACTCAAAAACCAAAAGAAACGCAGGACAGGCCAGCGGGGCT.
>>> # Do something with the sequence:
>>> record.seq.count("ACT")
331810
```

Reading multiple sequences with Bio.SeqIO

If a file contains multiple sequences, you can read them one by one using the `Bio.SeqIO.parse` function.

This function returns a SeqRecord iterator, which allows you to go over the list of sequences in the file.

Example

```
>>> from Bio import SeqIO
>>> parser = SeqIO.parse("egg1.mfa", "fasta")
>>> for seqrec in parser:
...     print(seqrec.description, len(seqrec.seq))
...
sample 1 fragment 0 232
sample 1 fragment 1 235
sample 1 fragment 2 123
...
sample 1 fragment 46 168
>>>
```

Writing Sequences with Bio.SeqIO

You can write one or a list of sequences to a file as well.

For instance, suppose we wanted **only long** reads from a file:

```
>>> from Bio import SeqIO
>>> parser = SeqIO.parse("egg1.mfa", "fasta")
>>> longseqrecs = []
>>> for seqrec in parser:
...     if len(seqrec.seq) > 200:
...         longseqrecs.append(seqrec)
```

We can write this to a new file using SeqIO.write:

```
>>> SeqIO.write(longseqrecs,"longegg1.mfa","fasta")
```

You can of course also write just one sequence, e.g.

```
>>> SeqIO.write(SeqRecord(Seq("ATCGTACC")),"example.fa","fasta")
```

Instead of filenames, you can also use Python file handles.

Online Genomic Data with Biopython

Online Genomic Databases with Biopython

- Databases are structured collections of data and have a well-define way (an “API”) to get data out.
- A lot of genomic data is available in online databases on the internet.
- It would be too much data to download all of it on your own computer.
- Specific bits of data can be requested from these online databases.
- With Biopython, we can do this from within Python.

(note: on many supercomputers you cannot access the internet from compute jobs so you’ll still have to download the data you need to the supercomputer before submit compute jobs.)

Databases of “Text”

- A lot of genomic data still comes as text.
- So these genomic databases are databases of text (for the most part).
- That’s convenient in an online setting, as the internet is text-based.

So far, we’ve covered quite a bit on how data is stored within Python, but not much outside it.

In fact, with Biopython you often do not need to know these details as long as you know the file format.

However, to interact with online databases, we need to look at what kind of data we can find in online genomic databases and how the data will be delivered.

Text Data Formats

- Humans made computers that agree on how to interpret text data as characters to be drawn on a screen in forms that humans can read.
- Despite the large amounts of encoding and decoding going on in silico and in vivo, we call text data 'human readable'.
- But further meaning can be given by the way the data is layed out, or by characters or strings with special meanings: this give rise to text formats.
- Humans are very flexible decoders, so you can easily put anything in a text file, and by looking at it, figure out much of its meaning.

Key-value pairs

In some text data formats, pieces of data are given a name, or label, or key that gives meaning to that piece of data. For instance, on the right is a reference in BibTeX format:

The keys are before the colon, the value follow the colon.

Which keys are valid can bepart of the format specification.

When key-values are use in addition to some other data, they are sometimes called "tags", or "metadata".

```
@article{
  author: {C.R. Harris, K.J. Millman, S.J. van der Walt, et
  title: {Array programming with NumPy},
  journal: {Nature},
  volume: {585}
  pages: {357-362}
  year: {2020}
}
```

XML

Values in key-values format rarely contain key-value pairs themselves.

XML is a general format that more naturally allows nested tags.

- Tags are names surrounded with angular brackets.
- The tags's end is the same except the name is prepended by a slash ("/").
- Between the start-tag and the end-tag of the tag there can be text, which is the content of the tag. This text can contain other tags.
- tags can have attributes, which are key=value pairs inside the start-tag.

Which tags are allowed can be part of the format, a.k.a., the schema.

HTML is form of XML.

```
<html>
 <head>
  <title>My Site</title>
 </head>
 <body>
  <p>
  All information that would be here is
  </p>
  <ol>
   <li>Not relevant, or</li>
   <li>can be found on
    <a href='https://google.com'>google</a>.
   </li>
  </ol>
 </body>
</html>
```

This format specifies essentially a tree of properties.

Relational databases

By this, folks mean tables of rows and columns where data in one column of a table may refere to data in a column of a different table. They are (still) the backbone of more web services.

The columns would be properties of the different rows would be different instances or entities. They are sometimes called **fields** as well.

They are rarely text-based, but from the restriction of everything being a table came some sound best practices that are useful for other formats too.

For instance:

- Don't repeat data.
- Each row should be an entity, a thing.
- Each thing should have a unique identifier by which it can be referenced.

CSV, TSV, JSON, XML

CSV and TSV

A kind, of poor-man's database, a **Comma-Separated-Values** or **Tab-Separated-Values** file contains a (single) table with row and columns, where the value of the columns of each row are separated by a comma or a tab character.

The first column often defines the column names, which should describe what property the values in that column hold.

CSV and TSV

Same as CSV except using a 'tab' character as column separator. Makes it easier to include commans in fields.

JSON

A loosely structured, hierarchical format based on JavaScript's Object Notation.

Genomic Data Formats

```
>1 dna:chromosome chromosome:Galgal4:1:1:195276750:1 REF
CCGACCAGTTGTAACTCAAAAACCAAAAGAAACGCAGGACAGGCCAGCGGGGCTGCCCCC
GCAGGAGCTGGAGAGAGTAGGGATTATTAGACCTGCACACAGCCCATACAACTCCCCCAT
...
```

FASTA

- One line starts with '>'. Following the '>' is the identifier, the rest is software-dependent.
- The next lines that do not start with '>' are part of the sequence, which can be split over several lines.
- Several sequences are allowed in one file.

Other formats

- Genbank
- BED format
- GFF3
- FASTQ

Public Online Genomic Databases

What could you expect to find in these?

- Complete reference genome sequences
- Protein structure
- Genotypes
- SNPs
- Databases (tables)

Typically have a web GUI and a web API.

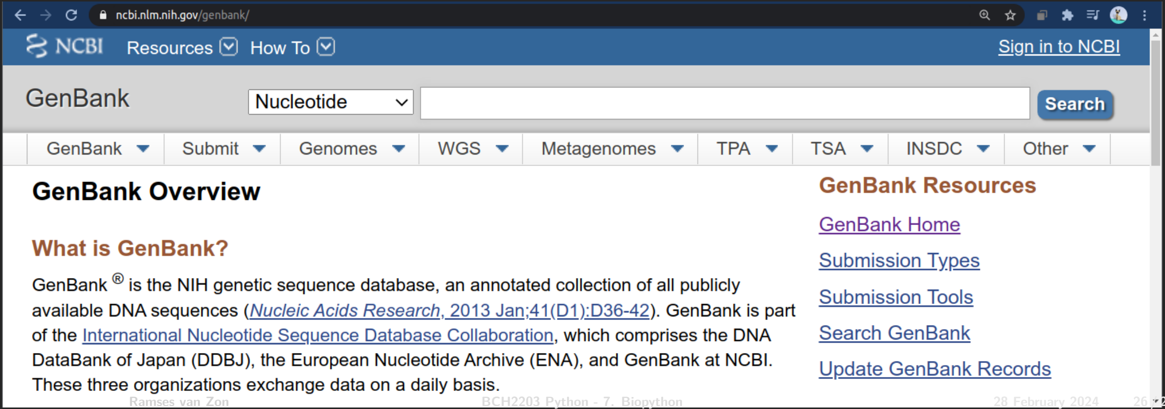
Genbank and Entrez

Genbank is an open access database of all publicly available nucleotide sequences and their protein translations

Maintained by NCBI.

www.ncbi.nlm.nih.gov/genbank

Entrez is its search and retrieval tool



GenBank Overview

What is GenBank?

GenBank ® is the NIH genetic sequence database, an annotated collection of all publicly available DNA sequences (*Nucleic Acids Research*, 2013 Jan;41(D1):D36-42). GenBank is part of the International Nucleotide Sequence Database Collaboration, which comprises the DNA DataBank of Japan (DDBJ), the European Nucleotide Archive (ENA), and GenBank at NCBI. These three organizations exchange data on a daily basis.

GenBank Resources

GenBank Home

Submission Types

Submission Tools

Search GenBank

Update GenBank Records

Access Entrez from Python

- Use Bio.Entrez

```
>>> from Bio import Entrez
```

- Let Genbank know your email (yes, they insist)

```
>>> Entrez.email = "rzon@scinet.utoronto.ca"
```

- Start a search, e.g. for *E. coli* DNA sequences, with at most 13 results:

```
>>> handle = Entrez.esearch(db="nucleotide", term="E. coli", retmax=13)
```

- Read the result:

```
>>> output = handle.read()
>>> print(output)
b'<?xml version="1.0" encoding="UTF-8" ?>\n<!DOCTYPE eSearchResult PUBLIC "-//NLM//DTD esearch 2006 0628//EN"
"https://eutils.ncbi.nlm.nih.gov/eutils/dtd/20060628/esearch.dtd">\n<eSearchResult><Count>17572373</Count>
<RetMax>13</RetMax><RetStart>0</RetStart><IdList>\n<Id>1906573107</Id>\n<Id>1906520706</Id>\n<Id>1906485224
</Id>\n<Id>1906425573</Id>\n<Id>1906410586</Id>\n<Id>1906400389</Id>\n<Id>1906377346</Id>\n<Id>1906368402</Id>
\n<Id>1906361046</Id>\n<Id>1906353597</Id>\n<Id>1906347957</Id>\n<Id>1906327159</Id>\n<Id>1906325706</Id>\n
</IdList><TranslationSet><Translation>     <From>E. coli</From>     <To>"Escherichia coli"[Organism] OR E.
coli[All Fields]</To>    </Translation></TranslationSet><TranslationStack>   <TermSet>    <Term>"Escherichia
coli"[Organism]</Term>    <Field>Organism</Field>    <Count>8450531</Count>    <Explode>Y</Explode>
</TermSet>   <TermSet>    <Term>E. coli[All Fields]</Term>    <Field>All Fields</Field>    <Count>10324226
</Count>    <Explode>N</Explode>    </TermSet>   <OP>OR</OP>   <OP>GROUP</OP>  </TranslationStack>
<QueryTranslation>"Escherichia coli"[Organism] OR E. coli[All Fields]</QueryTranslation></eSearchResult>\n'
```

Let's unpack this "mess"

- We get a string back from Entrez, but it's prepended with "b".
- That indicates that this is a bytes string, not a character string.
- Bytes strings can be converted to character strings with .decode()

```
>>> output = output.decode()
>>> print(output)
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE eSearchResult PUBLIC "-//NLM//DTD esearch 20060628//EN" "https://eutils.ncbi.nlm.nih.gov/eutils/dtd/20060
<eSearchResult><Count>17572373</Count><RetMax>13</RetMax><RetStart>0</RetStart><IdList>
<Id>1906573107</Id>
<Id>1906520706</Id>
<Id>1906485224</Id>
<Id>1906425573</Id>
<Id>1906410586</Id>
<Id>1906400389</Id>
<Id>1906377346</Id>
<Id>1906368402</Id>
<Id>1906361046</Id>
<Id>1906353597</Id>
<Id>1906347957</Id>
<Id>1906327159</Id>
<Id>1906325706</Id>
</IdList><TranslationSet><Translation>     <From>E. coli</From>     <To>"Escherichia coli"[Organism] OR E. coli[All
```

That's XML!

- Entrez returns search results as XML.
- Python has a built-in module called `xml` to deal with that.
- `xml` can either read xml from a file or from a string.
- It has a few sub modules:

  xml.etree.ElementTree, xml.dom, xml.dom.minidom, xml.dom.pulldom, xml.sax, xml.parsers.expat
- We really only need the basic submodule, etree

```
>>> import xml.etree.ElementTree as ET
>>> searchtree = ET.ElementTree(ET.fromstring(output))
>>> print(searchtree)
<xml.etree.ElementTree.ElementTree object at 0x2ac8d2ae4d68>
```

- This has read it as a tree for use by Python, it's no longer plain text.