BINF 5007

Lecture 11: BioPython

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 1.85 (Jan 15, 2025)
  Requires Python version 3.6 or higher.

Learning Objects

1. Understand Sequence Manipulations in Biopython:
   - Perform translation and transcription of nucleotide sequences.
   - Work with sequence annotations, locations, and features.

2. Work with `SeqRecord` Objects:
   - Create and manage `SeqRecord` objects, including adding metadata and annotations.

3. Handle Multiple Sequences Using `Bio.SeqIO`:
   - Read and parse multiple sequences from biological data files using `Bio.SeqIO`.
   - Write sequences back into standard formats such as FASTA.

4. Parse and Filter Biological Data:
   - Parse and filter sequences from a FASTA file while excluding invalid or empty entries.
   - Monitor and optimize memory usage during file parsing and sequence processing.

5. Read and Write Biological Data Files:
   - Understand the process of reading and writing various biological data formats efficiently.

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()
TCATGTGACATC

>>> seq.reverse_complement()
CTACAGTGTACT
```

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('CTGTAG')]
```

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

```
>>> seq = Seq('AGTACAcTGGTAG')
>>> seq.isupper()
AttributeError - Traceback (most recent call last)
...
AttributeError: 1Seq1 object has no attribute 1isupper1
>>> str(seq).isupper()
False
```

Transcription and Translation

Biopython knows 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('AGUACACUGUAG')
```

```
>>> 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)
 SerThrLeuTer
```

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: 1S1 is not a valid unambiguous letter for DNA
>>>    molecular_weight(prot[:-1],    "protein")
9.3541
```

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. o

- .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:Galgal4:1:1:195276750:1 REF
Number of features: 0
Seq('CCGACCAGTTGTAACTCAAAAACCAAAAGAAACGCAGGACAGGCCA
GCGGGGCT...GGA')
```

```
>>> # 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 a file with only long reads:

```
>>> 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.

Your Turn

```
from Bio import SeqIO
from memory_profiler import memory_usage
# pip3 install memory_profiler

def retseq(seqfh):
    """ Parse a fasta file and store non empty records
        into the fullseqs list.
    """

    # Empty list to store good sequences
    fullseqs = []
    for record in SeqIO.parse(seqfh, 'fasta'):
        if len(record.seq)!=0:
            fullseqs.append(record)

    seqfh.close()
    return fullseqs

print(f'Memory usage to start: {memory_usage()}MB')
# Name of the input file
fh = open('../assignment3/influenza.fasta')
# Name of the output file
newfh = open('test.fasta', 'w')
seqs = retseq(fh)
SeqIO.write(seqs, newfh, 'fasta')
newfh.close()
print(f'Memory usage after: {memory_usage()}MB')
```

Although this program does its job, it is not an example of efficient use of computer resources

The list `fullseqs` ends up with the information on every non-empty sequence in the file

For short sequence files this is not noticieable. In a realistic scenario, an input file of 100 Gigabytes can be very problematic!