Python for Advanced Beginners

Fernando Pineda 140.636 Oct. 15, 2017 version 0.1

Here we will use Biopython learn more Python and how to use Python on the cluster.

The go-to document for learning Biopython is the Biopython tutorial which can be found at:

http://biopython.org/DIST/docs/tutorial/Tutorial.pdf

9. Python via Biopython

Installing/accessing Biopython

Biopython is already installed on the cluster, but you need to use our site specific python installations to gain access to the BioPython libraries. To see what python versions are available do the following:

```
module avail
```

You should see a list of available modules. Let's use Python 3.4.3

```
module load python/3.4.3
```

If you want to install Biopython on your mac or another machine the following command(s) should work for a specific version of python

```
python2.7 -m pip install biopython
python3.6 -m pip install biopython
```

You can uninstall biopython by replacing `install` with `uninstall` in the above commands.

Seq objects

Similar to Bioperl, Biopython has `Seq` objects that hold sequences, `SeqRecord` objects that hold annotated seqquences, and `SeqIO` objects for file i/o. Let's start with `Seq` objects.

Seq objects only hold a string and an alphabet (corresponding to symbols for DNA, RNA, and proteins)

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq
Seq('AGTACACTGGT', Alphabet())
>>> str(my_seq)
'AGTACACTGGT'
>>> my_seq.alphabet
Alphabet()
```

Alphabet() is the generic alphabet. You can specify specific alphabets that correspond to IUPAC conventions (see e.g. http://www.chem.qmw.ac.uk/iupac/). See the Bioperl tutorial for more info.

Finally, Seq objects come with a set of useful methods e.g. (transalate(), reverse_complement(), etc.).

```
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq.reverse_complement()
Seq('ACCAGTGTACT', Alphabet())
```

SeqRecord objects

The `SeqRecord` class allows for rich annotation of sequences. A minimum SeqRecord object just has the info contained in a Seq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> my_seq = Seq("AGTACACTGGT")
r = SeqRecord(my_seq)
```

Probably the most common application of SeqRecord objects is to hold parsed sequence records from genomic sequence files.

SeqIO objects

There four main tools for reading sequences from files. Three of them return results in dictionaries or have dictionary-like read-only interfaces. One parses the input file and is useful for, e.g. format conversion.

Which tool you use will depend on what you want to do with the data once you have it. The following brief descriptions are based on the descriptions in the Biopython tutorial:

1. The `Bio.SeqIO.to_dict()` method is a helper function to read a file and build a normal Python dictionary with each entry held as a SeqRecord object. Thus the entire input file is parsed at once and held in memory. Since it is an actual Python dictionary the record can be edited (see Section 5.4.1 of the Biopython tutorial).
2. The `Bio.SeqIO.index()` is similar to the `Bio.SeqIO.to_dict()`. It provides and interface that looks like a dictionary to the calling routine, but the data remains on the disk and can't be edited. Record identifiers and file offset are stored in memory and it parses records, on demand, into SeqRecord objects as they are requested. In this respect, it acts like a read-only dictionary (see Section 5.4.2 of the BioPython tutorial).
3. The `Bio.SeqIO.index_db()` also acts like a read only dictionary but stores the identifiers and file offsets as an SQLite3 database on the disk, meaning it has *very* low memory requirements compared to the two prevous methods but will be slower (see Section 5.4.3 of the Biopython tutorial).
4. Finally, if you don't need to access the sequence records in random order, the `Bio.SeqIO.parse()` method is your friend. Here is how to convert a file from genbank format to fasta format using the `SeqIO.parse()` and `SeqIO.write()` methods.

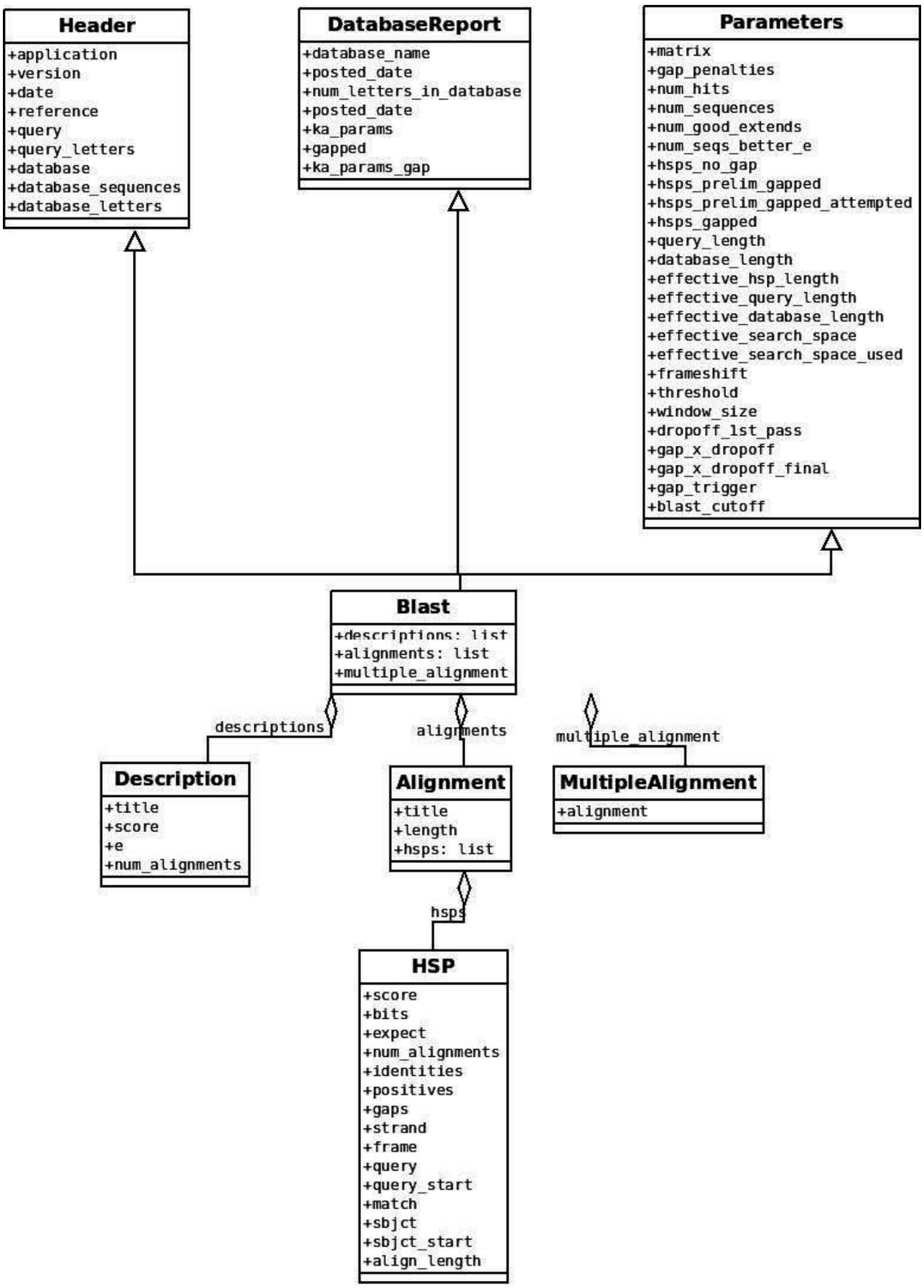
```
from Bio import SeqIO
records = SeqIO.parse("ls_orchid.gbk", "genbank")
count = SeqIO.write(records, "my_example.fasta", "fasta")
```

Standalone NCBI Blast+

```
from Bio.Blast.Applications import NcbiblastnCommandline
from Bio.Blast import NCBIXML

# let's run blastn and put the results into an xml output file
blaster = NcbiblastnCommandline(query="test_query.fa", db="refseq_rna.00",  evalue=0.001,
outfmt=5, out="results.xml")
stdout, stderr = blaster()


# now that the results are in the xml file, let's open the file, read it and get the
alignment data
result_handle = open("results.xml")
blast_record = NCBIXML.read(result_handle)

# step through each of the local alignments in the blastrecord and step through their
high-scoring-pairs (hsp).
E_VALUE_THRESH = 0.04

for alignment in blast_record.alignments:
        for hsp in alignment.hsps:
                if hsp.expect < E_VALUE_THRESH:
                        print("\n"+"-"*80)
                        print('sequence:', alignment.title)
                        print('length:', alignment.length)
                        print('e value:', hsp.expect)
```

```
                    # here we print the first 75 chars of the query, match and subject
strings
                    print(hsp.query[0:75] + '...')
                    print(hsp.match[0:75] + '...')
                    print(hsp.sbjct[0:75] + '...')
```

Blast record



10. A small digression to look at eval() and repr()

The `eval()` function evaluates a string as if it were a python statement, e.g.

```
>>> statement = "print('Hello world')"
>>> eval(statement)
Hello world
```

What if we passed a string to eval without a print statement.

```
>>> eval('hello world')
```

This would be like typing hello world at the python command line:

```
hello world
File "<stdin>", line 1
  hello world
            ^
SyntaxError: invalid syntax
```

Obviously a syntax error because neither hello nor world are python objects or reserved words. But this works

```
>>> eval("'hello world'")
```

Because this is like typing 'hello world' at the command line. It would simply create a str object with the value 'hello world'.

Now we can understand what the `repr()` function does. It attempts to create a string that, if you evaluated it, would recreate the object that you passed to it. We already saw above, that the string containing the command to create the 'hello world' string is "'hello world'", so this is what repr() returns

```
repr('hello world')
"'hello world'"
```

The idea of repr is to give a string which contains a series of symbols which we can type in the interpreter and get the same value which was sent as an argument to repr.

The `repr()` function an argument and attempts to build a string, which if given to the python interpreter would give the same value that was sent to the repr() that evaluates to the object if given to `eval()`. For example

```
>>> eval(repr('hello world'))
'hello world'
```


1. The Python language reference https://docs.python.org/3/reference/index.html
2. Biopython tutorial http://biopython.org/DIST/docs/tutorial/Tutorial.pdf
3. Illustrating Python via Bioinformatics Examples http://hplgit.github.io/bioinf-py/doc/pub/html/index.html