Biopython Tutorial (II): from beginner to advanced.

MONDAY. JANUARY 18, 2021 - 13 MINS

PYTHON BIOPYTHON BIOINFORMATICS

Welcome back to another post about Biopython. Biopython is a Python library for reading and writing many common biological data formats and on this post we will be explaining some basic and advanced concepts and tricks. It is a extremely powerful library which provides a wide range of functions and can be used from working with sequences to evolutionary genomics or structural proteomics. And many more!

These are the contents we will be covering. If you haven't read part 1 of this tutorial, I highly recommend it! I hope you enjoy and learn something new today.

1. First Steps

- What is Biopython
- Installing and upgrading

2. Working with Sequences

- Creating a sequence
- Parsing a sequence file
- Counting sequence length & number of occurrences of a nucleotide
- Calculating GC-content
- Calculating molecular weight
- Get the reverse complement of a sequence, transcription & translation
- Slicing a sequence
- Concatenating sequences
- Finding the starting index of a subsequence
- Writing sequences to a file
- Converting a FASTQ file to FASTA file & other formats

3. NCBI Entrez databases

- General Guidelines
- Accessing Pubmed, Medline, Genbank & others

4. BLAST

- Running a web BLAST
- Parsing a BLAST output

5. Multiple Sequence Alignment

- Pairwise sequence alignment
- Multiple sequence alignment: creating alignment using different algorithms: ClustalW, MUSCLE…

- Reading a MSA

6. Phylogenetics

- Constructing a phylogenetic tree
- Modifying an existing tree

7. Sequence motif analysis

8. PDB: 3D structure protein analysis

- Count atoms in a PDB structure

---

4. BLAST

Running Web BLAST

BLAST is an algorithm and program for comparing primary biological sequence information (i.e, protein or aminoacid sequences). Using Biopython, you can align sequences with Web BLAST which is the online version of BLAST, using the `Bio.Blast.NCBIWWW` module. Here is an example of a BLASTN (Nucleotide BLAST), searching a known sequence on the nucleotide database (nt).

```
from Bio import SeqIO
record = SeqIO.read("seq_example.fasta", format="fasta")
results_handle = NCBIWWW.qblast("blastn", "nt", record.format("fasta"))

# You could now save the results in XML format output as default

with open("blastn_result.xml", "w") as out_handle:
```

```
    out_handle.write(result_handle.read())
result_handle.close()
```

For more information and other parameters make sure to check official information doing:

```
from Bio.Blast import NCBIWWW
help(NCBIWWW.qblast)
```

Please note that running a Web BLAST can be slower than a local BLAST, and you can create custom, personal databases to search for sequences against if running locally. You can find more information on the official wiki (and maybe some tutorial here soon ;))

Parsing a BLAST output

Our saved BLAST output can be parsed as follows

```
result_handle = open("blastn_result.xml")
```

As said earlier, our BLAST output is in XML format, so we can parse the result using `NCBIXML`. Please note that we have used one query sequence and so, we are only getting one record. This code can be quickly adapted to multiple query sequences changing

`blast_record=NCBIXML.read(result_handle)` to

`blast_records=NCBIXML.parse(result_handle)`

```
from Bio.Blast import NCBIXML
blast_records = NCBIXML.read(result_handle)
```

You can also parse a BLAST XML output and get other important parameters, such as Length or e-value as follows:

```
from Bio.Blast import NCBIXML
result_handle = open("seq_example.xml")
blast_record = NCBIXML.read(result_handle)
for alignment in blast_record.alignments:
     for hsp in alignment.hsps:
         if hsp.expect < 1e-10:
            print('Seq:', alignment.title)
            print('Length:', alignment.length)
            print('e-value:', hsp.expect)
            print(hsp.query)
            print(hsp.match)
            print(hsp.sbjct)
```

---

5. Multiple Sequence Alignment

Pairwise sequence alignment

Pairwise sequence alignment methods are used to find the best-matching piecewise (local or global) alignments of two query sequences. In other words, a PSA is a tool used to compare two sequences to identify regions of similarity. We will be using the `Bio.pairwise2` module for PSA. Note that also `Bio.Align.PairwiseAligner` is available to use and it's faster for really big sequences.

We will be comparing two example sequences, `seq_example` and `seq_example2` in fasta format.

```
from Bio import pairwise2
from Bio import SeqIO
from Bio.pairwise2 import format_alignment

seq1 = SeqIO.read("seq_example.faa", "fasta")
seq2 = SeqIO.read("seq_example2.faa", "fasta")
```

```
alignments = pairwise2.align.globalxx(seq1.seq, seq2.seq)
print(alignments)
```

As you can see, in `pairwise2.align.globalxx` we find two parameters (written as xx) where you can specify the match score and gap penalties. The match score indicates the compatibility between an alignment of two characters in the sequences, and gap penalties is a negative number to apply when finding gaps between both sequences. You can read more information about this on Biopython's API. Better alignments are usually obtained by penalizing gaps: higher costs for opening a gap and lower costs for extending an existing gap.

Change the scoring scheme as it follows in this example (matching are given +2 points, -1 point for each mismatching character, -0.5 points when opening a gap, and -0.1 points when extending it)

```
pairwise2.align.globalms(seq1.seq, seq2.seq, match=2, mismatch=-1, open=-.5, extend
```

If you want do a local alignment of two sub-regions of a pair of sequences similarly, using `pairwise2.align.localxx`

```
alignments = pairwise2.align.localxx(seq1, seq2)
```

For amino acid sequences match scores are usually encoded in matrices like PAM or BLOSUM. Thus, a more meaningful alignment for our example can be obtained by using the BLOSUM62 matrix, together with a gap open penalty of 10 and a gap extension penalty of 0.5. You can read more about amino acid score matrix on this Nature article

```
from Bio import pairwise2
from Bio import SeqIO
```

```
from Bio.Align import substitution_matrices
blosum62 = substitution_matrices.load("BLOSUM62")
seq1 = SeqIO.read("seq_example.faa", "fasta")
seq2 = SeqIO.read("seq_example2.faa", "fasta")
alignments = pairwise2.align.globalds(seq1.seq, seq2.seq, blosum62, -10, -0.5)
```

Reading a MSA

There are two functions for reading in sequence alignments, `Bio.AlignIO.read()` and `Bio.AlignIO.parse()`, for files containing one or multiple alignments respectively. Usage is quite straightforward, however, you must include filename and alignment format type. This is one example with Stockholm file format:

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

And with fasta file format:

```
from Bio import AlignIO
alignment = AlignIO.read("RHDV_example.faa", "fasta")
print(alignment)
```

Multiple sequence alignment: creating alignment using different algorithms: ClustalW, MUSCLE…

You can create your own multiple sequence alignment (MSA) when you want to compare more than two sequences to find regions with high similarity. The main function for this is `Bio.AlignIO.write()`. However, Biopython provides command-line wrappers for the most common MSA tools: ClustalW, Clustal-O, MUSCLE, T-Coffee… Every algorithm is

different and you should know the most optimal method to use in your case. You can get more information about algorithms on this paper.

For example, if you want to use ClustalW, then first, you have to download its precompiled binaries. Then you can get an executable command as follows.

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

And then you just run `clustalw2` on your terminal. Another example using Clustal-O:

```
from Bio.Align.Applications import ClustalOmegaCommandline
in_file = "seq_example.fasta"
out_file = "seq_alignment.phylip"
clustalomega_cline = ClustalOmegaCommandline(infile=in_file, outfile=out_file, outf
print(clustalomega_cline)
```

And run `clustalomega_cline`. You can convert between sequence alignment file formats using Biopython aswell. Here is an example code, from Stockholm format used earlier to ClustalW format:

```
from Bio import AlignIO
count = AlignIO.convert("RHDV_example.sth", "stockholm", "RHDV_example.aln", "clust
```

---

6. Phylogenetics

Constructing a phylogenetic tree

A phylogenetic tree is a branching diagram that represents evolutionary relationships among organisms or genes. They depict the evolution of a set of taxa from their most recent common ancestor (MRCA). It is one of the most important studies to perform in evolutionary genomics. You can read more about phylogenetic trees here.

In Biopython, we can use `Bio.Phylo` module to create our own tree. As an example we will create a tree from a example file (download here, from Biopython's API) named `simple.dnd` in Newick format.

```
from Bio import Phylo
tree = Phylo.read("simple.dnd", "newick")
```

Using `draw_ascii` we can get a simple plain text dendrogram

```
from Bio import Phylo
tree = Phylo.read("simple.dnd", "newick")
Phylo.draw_ascii(tree)
```

However you can get a prettier tree using matplotlib, a library for creating visualizations in Python

```
tree.rooted = True
Phylo.draw(tree)
```

Exactly as in MSA, there are three main categories of algorithms avaliable to construct a phylogenetic tree: Distance-based methods, Maximum Parsimony (MP) methods and probabilistic methods. You can read more about the differences here. Biopython allows you to choose the model you want to use with `DistanceTreeConstructor` and `ParsimonyTreeConstructor` such as:

```
from Bio import Phylo
from Bio.Phylo.TreeConstruction import DistanceCalculator
from Bio.Phylo.TreeConstruction import DistanceTreeConstructor
from Bio import AlignIO

alignment = AlignIO.read('simple.dnd', 'newick')
constructor = DistanceTreeConstructor()
calculator = DistanceCalculator('identity')
dm = calculator.get_distance(alignment)
tree = constructor.upgma(dm)
Phylo.draw_ascii(tree)
```

The output is an UPGMA tree created from our input file. If you want to get a Neighbour Joining tree (which is a Distance based method) just do:

```
njtree = constructor.nj(dm)
print(njtree)
```

Modifying an existing tree

The simplest way to get an overview of a tree object is to print it as we saw earlier:

```
from Bio import Phylo
tree = Phylo.read("simple.dnd", "newick")
print(tree)
```

If you have your tree in a PhyloXML format, you can color different branches or apply different widths. Please note that you can still save a tree as Newick, but the color and width values will be skipped in the output file. To apply colors just do:

```
tree = tree.as_phyloxml()
tree.root.color = "gray"
Phylo.draw(tree)
```

Colors for a clade are treated as cascading down through the entire clade, so when we colorize the root here, it turns the whole tree gray. We can override that by assigning a different color lower down on the tree using:

```
tree.clade[0, 1].color = "blue"
tree.clade[2, 3].color = "red"
Phylo.draw(tree)
```

This is a wrap for this post! I hope you liked it.

Thank you for reading. Feel free to check out other posts about Biopython and many other things.

Let me know if you have any feedback or issues, I'd love to hear from you!

Related Posts

- Rosalind: Python solutions to common problems in Bioinformatics
- Project Euler: a Python (beginner friendly) approach to complex mathematical challenges
- Biopython Tutorial (I): from beginner to advanced.

David Boo

Where biology, informatics & statistics intersect

Tweet Share