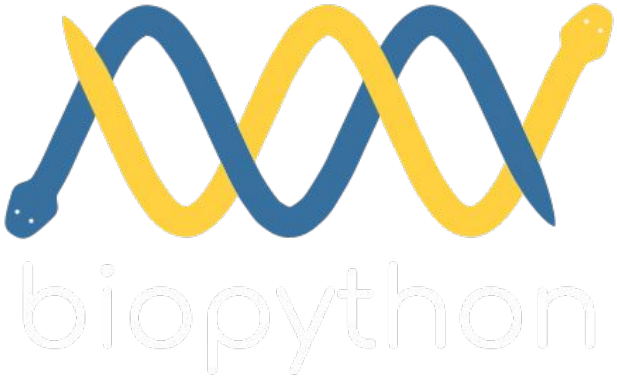
Scientific Programming Practical 10

Introduction

Luca Bianco - Academic Year 2018-19
luca.bianco@fmach.it

Biopython

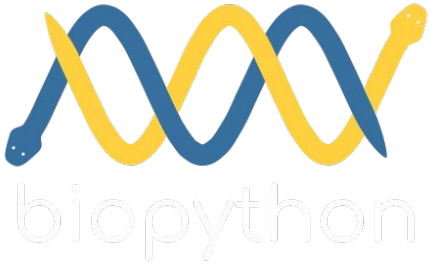


FROM Biopython's website:

The Biopython Project is an international association of developers of freely available **Python tools for computational molecular biology**.

The goal of Biopython is to make it as easy as possible to use **Python for bioinformatics** by creating high-quality, reusable modules and classes.

Biopython



Biopython:

1. Provides tools to **parse several common bioinformatics formats** (e.g. FASTA, FASTQ, BLAST, PDB, Clustalw, Genbank,..).

2. Provides an **interface towards biological data repositories** (e.g. NCBI, Expasy, Swiss-Prot,..)

3. Provides an **interface towards some bioinformatic tools** (e.g. clustalw, MUSCLE, BLAST,…)

4. **Implements some tools** like pairwise alignment **and data structures** to deal with biological data.

Seq objects

Seq objects are more powerful than string to deal with sequences and are defined in the module Bio.Seq.

They have two information:

1. **SEQUENCE**

2. **ALPHABET**
(optional, but useful to check things)

defined in
`Bio.Alphabet.IUPAC`

They are **immutable objects**. The mutable version is **MutableSeq**.

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC

#No alphabet specified
s = Seq("GATTACATAATA")
dna_seq = Seq("GATTATACGTAC", IUPAC.unambiguous_dna)
print("S:", s)
print("S's alphabet:", s.alphabet)
print("dna_seq:", dna_seq)
print("dna_seq's alphabet:", dna_seq.alphabet)

my_prot = Seq("MGNAAAAKKGSEQE", IUPAC.protein)
print("my_prot:", my_prot)
print("my_prot's alphabet:", my_prot.alphabet)
```

```
S: GATTACATAATA
S's alphabet: Alphabet()
dna_seq: GATTATACGTAC
dna_seq's alphabet: IUPACUnambiguousDNA()
my_prot: MGNAAAAKKGSEQE
my_prot's alphabet: IUPACProtein()
```

Seq objects

Seq objects behave like strings, but the consistency of the alphabet is checked too.

For example we cannot concatenate a **unambiguous_dna** with a **IUPAC.protein** sequence.

```
my_mess = dna_seq + my_prot
```

```
S: GATTACATAATA
S's alphabet: Alphabet()
dna_seq: GATTATACGTAC
dna_seq's alphabet: IUPACUnambiguousDNA()
my_prot: MGNAAAAKKGSEQE
my_prot's alphabet: IUPACProtein()

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-4c1f6d65a691> in <module>()
     14 print("my_prot's alphabet:", my_prot.alphabet)
     15
---> 16 my_mess = dna_seq + my_prot

/usr/local/lib/python3.5/dist-packages/Bio/Seq.py in __add__(self, other)
    296                 raise TypeError(
    297                     "Incompatible alphabets {0!r} and {1!r}".format(
--> 298                         self.alphabet, other.alphabet))
    299             # They should be the same sequence type (or one of them is generic)
    300             a = Alphabet._consensus_alphabet([self.alphabet, other.alphabet])

TypeError: Incompatible alphabets IUPACUnambiguousDNA() and IUPACProtein()
```

Seq objects

Seq objects behave like strings, but the consistency of the alphabet is checked too.

For example we cannot concatenate a **unambiguous_dna** with a **IUPAC.protein** sequence.

```
from Bio.Seq import Seq
from Bio.Alphabet import generic_alphabet

dna_seq = Seq("GATTATACGTAC", IUPAC.unambiguous_dna)
my_prot = Seq("MGNAAAAKKGSEQE", IUPAC.protein)

my_prot.alphabet = generic_alphabet

#Does it really make sense though?!?
print(dna_seq + my_prot)
```

```
GATTATACGTACMGNAAAAKKGSEQE
```

Seq objects

Seq objects behave like strings, but the consistency of the alphabet is checked too.

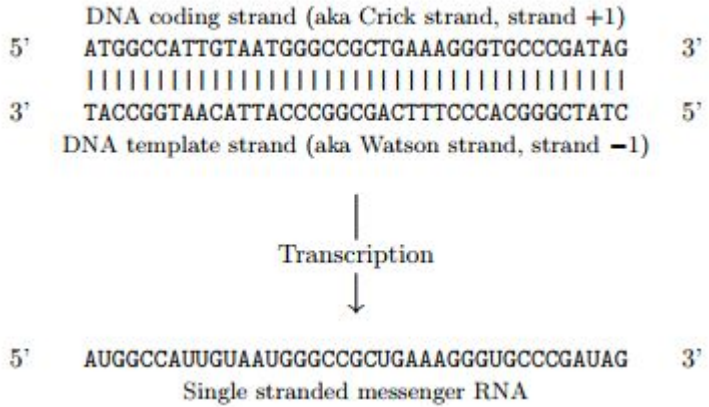
We can loop through the elements of the sequence and perform slicing...

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC

dna_seq = Seq("GATTATACGTACGGCTA", IUPAC.unambiguous_dna)

for base in dna_seq:
    print(base, end = " ")

print("")

sub_seq = dna_seq[4:10]
print(sub_seq)

#Let's reverse the string:

print("Reversed: ", dna_seq[::-1])
#from Seq to string:
dna_str = str(dna_seq)
print("As string:", dna_str)
print(type(dna_str))
```

```
G A T T A T A C G T A C G G C T A
ATACGT
Reversed:  ATCGGCATGCATATTAG
As string: GATTATACGTACGGCTA
<class 'str'>
```

Seq objects

Biopython provides several methods working on Seq objects



General methods (return int and Seq objects):

`Seq.count(s)` : counts the number of times s appears in the sequence;

`Seq.upper()` : makes the sequence of the object Seq upper case

`Seq.lower()` : makes the sequence of the object Seq lower case

Only for DNA/RNA (return Seq objects):

`Seq.complement()` to complement the sequence

`Seq.reverse_complement()` to reverse complement the sequence.

`Seq.transcribe()` transcribes the DNA into mRNA

`Seq.back_transcribe()` back transcribes mRNA into DNA

`Seq.translate()` translates mRNA or DNA into proteins

Other functions are in **SeqUtils:**

`SeqUtils.GC(Seq)` computes GC content

`SeqUtils.MolecularWeight(Seq)` computes the molecular weight of the sequence

….

Seq objects

Biopython provides several methods working on Seq objects

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC

coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous_dna)
print(coding_dna)

mrna = coding_dna.transcribe()
print(mrna)
print("")
print(mrna.alphabet)
print("")
print("... and back")
print(mrna.back_transcribe())
print("")
print("Translation to protein:")
prot = mrna.translate()
print(prot)
print("")
print("Up to first stop:")
print(mrna.translate(to_stop = True))
print("")
print("Mitocondrial translation: (TGA is W!)")
mit_prot = mrna.translate(table=2)
mit_prot
```

```
ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG
AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG

IUPACUnambiguousRNA()

... and back
ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG

Translation to protein:
MAIVMGR*KGAR*

Up to first stop:
MAIVMGR

Mitocondrial translation: (TGA is W!)

Seq('MAIVMGRWKGAR*', HasStopCodon(IUPACProtein(), '*'))
```

Sequence annotations

The **SeqRecord** object is used to store annotations associated to sequences

1. `SeqRecord.seq` : the sequence (the Seq object)
2. `SeqRecord.id` : the identifier of the sequence, typically an accession number
3. `SeqRecord.name` : a "common" name or identifier sometimes identical to the accession number
4. `SeqRecord.description` : a human readable description of the sequence
5. `SeqRecord.letter_annotations` : a per letter annotation using a restricted dictionary (e.g. quality)
6. `SeqRecord.annotations` : a dictionary of unstructured annotation (e.g. organism, publications,...)
7. `SeqRecord.features` : a list of SeqFeature objects with more structured information (e.g. genes pos).
8. `SeqRecord.dbxrefs` : a list of database cross references.

Sequence annotations

Read a fasta file NC005816.fna containing the whole sequence for Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1 and retrieve some information about the sequence.

```
ID: gi|45478711|ref|NC_005816.1|
Name: gi|45478711|ref|NC_005816.1|
Description: gi|45478711|ref|NC_005816.1| Yersinia
pestis biovar Microtus str. 91001 plasmid pPCP1,
complete sequence
Number of features: 0
Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGAT
CCAGG...CTG', SingleLetterAlphabet())

Sequence [first 30 bases]:
TGTAACGAACGGTGCAATAGTGATCCACAC

The id:
gi|45478711|ref|NC_005816.1|

The description:
gi|45478711|ref|NC_005816.1| Yersinia pestis biovar
Microtus str. 91001 plasmid pPCP1, complete sequence

The record is a:  <class 'Bio.SeqRecord.SeqRecord'>
```

```
from Bio import SeqIO

record =
SeqIO.read("file_samples/NC_005816.fna",
"fasta")

print(record)
print("")
print("Sequence [first 30 bases]:")
print(record.seq[0:30])
print("")
print("The id:")
print(record.id)
print("")
print("The description:")
print(record.description)
print("")
print("The record is a: ", type(record))
```

SeqIO.parse

The `Bio.SeqIO` module aims to provide a simple way to work with several different sequence file formats

The method `Bio.SeqIO.parse` is used to parse some sequence data into a **SeqRecord iterator**. In particular, the basic syntax is:

```
SeqRecordIterator = Bio.SeqIO.parse(filename, file_format)
```

where `filename` is typically an open handle to a file and `file_format` is a lower case string describing the file format. Possible options include **fasta, fastq-illumina, abi, ace, clustal...** all the

Note that `Bio.SeqIO.parse` returns an iterator, therefore it is possible to manually fetch one SeqRecord after the other with the `next(iterator)` method.

WARNING: When dealing with very large FASTA or FASTQ files, the overhead of working with all these objects can make scripts too slow. In this case `SimpleFastaParser` and `FastqGeneralIterator` parsers might be better as they which return just a tuple of strings for each record.

SeqIO

Example: Let's get the first 3 entries of the .fasta file contigs82.fasta printing off the length of the sequence and the first 50 bases of each sequence followed by "...".

In [12]:
```
from Bio import SeqIO

seqIterator = SeqIO.parse("file_samples/contigs82.fasta", "fasta")

labels = ["1st","2nd","3rd"]
for l in labels:
    seqRec = next(seqIterator)
    print(l, "entry:")
    print(seqRec.id, " has size ", len(seqRec.seq))
    print(seqRec.seq[:50]+"...")
    print("")
```

```
1st entry:
MDC020656.85  has size  2802
GAGGGGTTTAGTTCCTCATACTCGCAAAGCAAAGATACATAAATTTAGAA...

2nd entry:
MDC001115.177  has size  3118
TGAATGGTGAAAATTAGCCAGAAGATCTTCTCCACACATGACATATGCAT...

3rd entry:
MDC013284.379  has size  5173
TATCGTTTCCTCTGAGTAGAATATCGTTATAACAAGATTTTTTTTTTCCT...
```

SeqIO

The module `Bio.SeqIO` also has three different ways to allow random access to elements:

1. `Bio.SeqIO.to_dict(file_handle/iterator)` : builds a dictionary of all the SeqRecords keeping them in memory and allowing modifications to the records. **This potentially uses a lot of memory but is very fast;**
2. `Bio.SeqIO.index(filename,file_type)` : builds a sort of read-only dictionary, parses the elements into SeqRecords on demand (i.e. it returns an iterator!). **This method is slower, but more memory efficient;**
3. `Bio.SeqIO.index_db(indexName.idx,filenames, file_format)` : builds a read-only dictionary, but stores ids and offsets on a SQLite3 database. **It is slower but uses less memory.**

SeqIO.write

The module `Bio.SeqIO` provides also a way to write sequence records to files in various formats (like fasta, fastq, genbank, pfam…)

SeqRecords can be written out to files by using

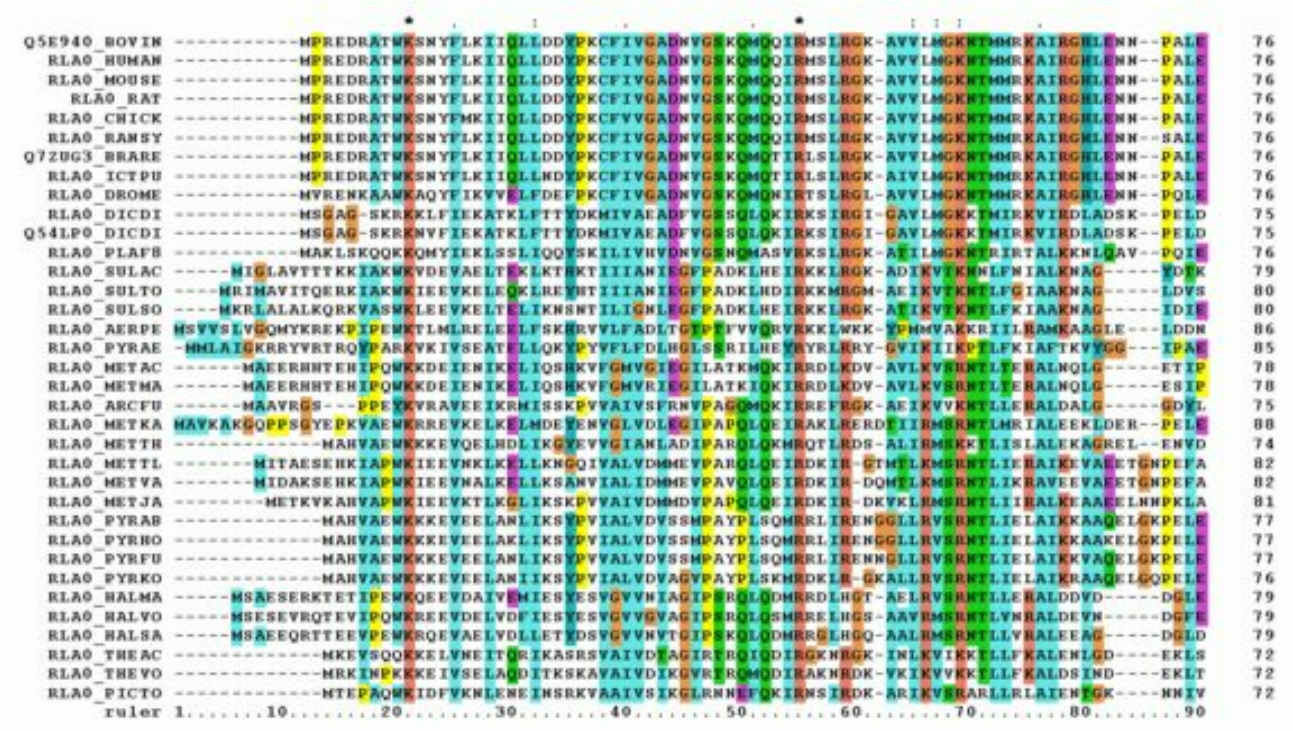
```
N = Bio.SeqIO.write(records,out_filename, file_format)
```

where **records** is a list of the SeqRecords to write, **out_filename** is the string with the filename to write and **file_format** is the format of the file to write. **N** is the number of sequences written.

WARNING: If you write a file that is already present, `SeqIO.write` will just rewrite it without telling you.

Multiple sequence alignment

Multiple Sequence Alignments are a collection of **multiple sequences which have been aligned together** – usually with the insertion of gap characters, and addition of leading or trailing gaps – such that all the sequence strings are the same length.



Parsing MSAs

The function `Bio.AlignIO.parse()` returns an iterator of `MultipleSeqAlignment` objects that is a collection of `SeqRecords`.

In the frequent case that we have to deal with a single multiple alignment we will have to use the `Bio.AlignIO.read()` function.

The basic syntax of the two functions:

```
Bio.AlignIO.parse(file_handle, alignment_format)
Bio.AlignIO.read(file_handle, alignment_format)
```

where `file_handle` is the handler to the opened file, while the `alignment_format` is a lower case string with the alignment format (e.g. fasta, clustal, stockholm, mauve, phylip,...).

```
from Bio import AlignIO

alignments = AlignIO.read("file_samples/PF02171_seed.sth", "stockholm")


for align in alignments:
    start = align.annotations["start"]
    end = align.annotations["end"]
    seq = align.seq
    desc = align.description
    dbref = ",".join([x for x in align.dbxrefs])
    print("{} S:{} E:{}".format(desc, start, end))
    if(len(dbref) > 0):
        print(dbref)
    print("{}".format(seq))
    print("")
```

```
AGO1_SCHPO/500-799 S:500 E:799
YLFFILDK-NSPEP-YGSIKRVCNTMLGVPSQCAISKHILQS---------KPQYCANLGMKINVKVGGIN-CSLIPKSNP----L

AGO6_ARATH/541-851 S:541 E:851
FILCILPERKTSDI-YGPWKKICLTEEGIHTQCICPIKI------------SDQYLTNVLLKINSKLGGIN-SLLGIEYSYNIPLI

AGO4_ARATH/577-885 S:577 E:885
FILCVLPDKKNSDL-YGPWKKKNLTEFGIVTQCMAPTRQPND-----------QYLTNLLLKINAKLGGLN-SMLSVERTPAFTVI

TAG76_CAEEL/660-966 S:660 E:966
CIIVVLQS-KNSDI-YMTVKEQSDIVHGIMSQCVLMKNVSRP---------TPATCANIVLKLNMKMGGIN--SRIVADKITNKYL
```

Writing and converting MSAs

Biopython provides a function `Bio.AlignIO.write()` to write alignments to file
and

`Bio.AlignIO.convert()` to convert one format into the other (provided that all information needed for the second format is available)

```
N = Bio.AlignIO.write(alignments,outfile,file_format)
```

where `alignments` are a MultipleSeqAlignment object with the alignments to write to the output file with name `outfile` that has format `file_format` (a low case string with the file format). N is the number of entries written to the file.

Ex.

```
my_alignments = [align1, align2, align3]
N = AlignIO.write(my_alignments, "file_samples/my_malign.phy", "phylip")
```

```
Bio.AlignIO.convert(input_file, input_file_format, output_file, output_file_format)
```

basically by passing the input file name and format and output file name and format.

Ex:

```
Bio.AlignIO.convert("PF05371_seed.sth", "stockholm", "PF05371_seed.aln", "clustal")
```

Manipulating/writing MSA

It is possible to slice alignments using the [] operator applied on a `SeqRecord`.

Think about it as a matrix

1. `SeqRecord[i,j]` returns the jth character of alignment i as a string;
2. `SeqRecord[:,j]` returns all the jth characters of the multiple alignment as a string;
3. `SeqRecord[:,i:j]` returns a MultipleSeqAlignment with the sub-alignments going for i to j (excluded)
4. `SeqRecord[a:b,i:j]` similar to 3. but for alignments going from a to b (excluded) only

```
YLFFILDK-NSPEP-YGSIKLVPPVYYAHLVSNLARYQDV
FILCILPERKTSDI-YGPWKIVAPVRYAHLAAAQVAQFTK
FILCVLPDKKNSDL-YGPWKVVAPICYAHLAAAQLGTFMK
CIIVVLQS-KNSDI-YMTVKIPTPVYYADLVATRARCHVK
LIVVVLPG--KTPI-YAEVKIPAPAYYAHLVAFRARYHLV
TFVFIITD-DSITT-LHQRYLPTPLYVANEYAKRGRNLWN
DILVGIAR-EKKPD-VHDILVPDVLYAAENLAKRGRNNYK
TIVFGIIA-EKRPD-MHDILIPNVSYAAQNLAKRGHNNYK
MLVVMLAD-DNKTR-YDSLKVPAPCQYAHKLAFLTAQSLH
IVMVVMRS-PNEEK-YSCIKVPAVCHYAHKLAFLVAESIN
LILCLVPN-DNAER-YSSIKVPAVCQYAKKLATLVGTNLH
IVVCLLSS-NRKDK-YDAIKVPAPCQYAHKLAFLVGQSIH
GIMLVLPE-YNTPL-YYKLKLPVTVNYPKLVAGIIANVNR
CFALIIGKEKYKDNDYYEILIPAPIHYADKFVKALGKNWK
LVIVFLEEYPKVDP-YKSFLLPATVHYSDKITKLMLRGIE
LLLAILPD-NNGSL-YGDLKIVPPAYYAHLAAFRARFYLE
```

align[0,0] is Y
align[2,1] is I
align[:,0] is YFFCLTDTMILIGCLL

align[:,0:3] gets first 3 rows (SeqRecords)
align[0:3,0:3] first 3 cols of first 3 rows (SeqRecords):

Pairwise alignment

Biopython has its own module to make pairwise alignment. It provides two algorithms: Smith-Waterman for local alignment and Needleman-Wunsch for global alignment. These methods are implemented in two Biopython functions of the `Bio.pairwise2` module:

`pairwise2.align.globalxx()`
`pairwise2.align.localxx()`

```
aligns = pairwise2.align.globalxx(seq1,seq2)
aligns = pairwise2.align.localxx(seq1,seq2)
```

where seq1 and seq2 are two `str` objects. These methods return a list of alignments (at least one) that have the same **optimal score**. Each alignment is represented as tuples with the following 5 elements in order:

1. The alignment of the first sequence;
2. The alignment of the second sequence;
3. The alignment score;
4. The start of the alignment (for global alignments this is always 0);
5. The end of the alignment (for global alignments this is always the length of the alignment).

Example:

```
alignments = pairwise2.align.globalxx("ACCGTTATATAGGCCA", "ACGTACTAGTATAGGCCA")
for i in range(len(alignments)):
    print(alignments[i])

('ACCGT--TA-TATAGGCCA', 'A-CGTACTAGTATAGGCCA', 15.0, 0, 19)
('ACCGT--TA-TATAGGCCA', 'AC-GTACTAGTATAGGCCA', 15.0, 0, 19)
```

Pairwise alignment

```
OPTIONS FOR MATCHES/MISMATCHES
AND GAP OPENS/EXTENSIONS

pairwise2.align.globalxx
pairwise2.align.globalmx
pairwise2.align.globalms
pairwise2.align.globalmd
pairwise2.align.globalxd
pairwise2.align.globalxs
pairwise2.align.localxx
pairwise2.align.localmx
pairwise2.align.localms
pairwise2.align.localmd
pairwise2.align.localxd
pairwise2.align.localxs
```

Match parameters can be:

- `x` : means that a match scores 1 a mismatch 0;
- `m` : the match and mismatch score are passed as additional params after the sequence (es. `aligns = pairwise2.align.globalmx(seq1,seq2, 1, -1)` to set 1 as match score and -1 as mismatch penalty.

Gap parameters can be:

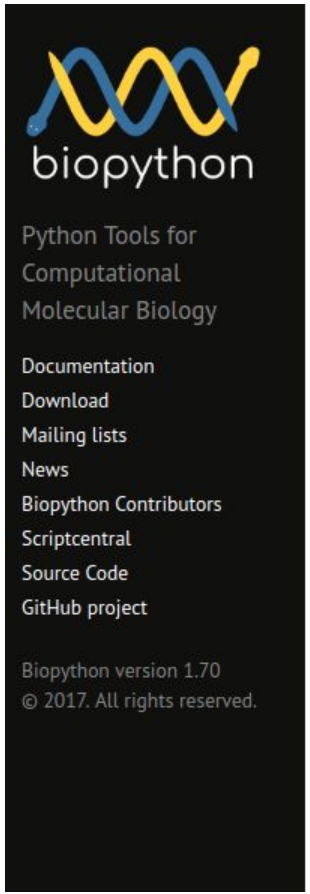
- `x` : gap penalty is 0;
- `s` : same gap open and gap extend penalties for the 2 sequences (passed as additional params after seqs).
- `d` : different gap open and gap extend penalties for the 2 seqs (additional params after the seqs).

Pairwise alignment

```
('ACCGT--TA-TATAGGCCA', 'A-CGTACTAGTATAGGCCA', 15.0, 0, 19)
('ACCGT--TA-TATAGGCCA', 'AC-GTACTAGTATAGGCCA', 15.0, 0, 19)

Looping through aligns
ACCGT--TA-TATAGGCCA
A-CGTACTAGTATAGGCCA
Score: 15.0, Start: 0, End: 19

ACCGT--TA-TATAGGCCA
AC-GTACTAGTATAGGCCA
Score: 15.0, Start: 0, End: 19


Match: 1, Mismatch: -1, Gap open: -0.5, Gap extend: -0.2
ACCGT--TA-TATAGGCCA
A-CGTACTAGTATAGGCCA
Score: 13.3, Start: 0, End: 19

ACCGT--TA-TATAGGCCA
AC-GTACTAGTATAGGCCA
Score: 13.3, Start: 0, End: 19
```

```
from Bio import pairwise2
from Bio import SeqIO


alignments = pairwise2.align.globalxx("ACCGTTATATAGGCCA",
                                      "ACGTACTAGTATAGGCCA")

for i in range(len(alignments)):
    print(alignments[i])

print("")
print("Looping through aligns")
for align in alignments:
        print(align[0])
        print(align[1])
        print("Score: {}, Start: {}, End: {}".format(align[2],
                                                     align[3],
                                                     align[4]))
        print("")

alignments = pairwise2.align.globalms("ACCGTTATATAGGCCA",
                                      "ACGTACTAGTATAGGCCA",
                                      1,-1,-0.5,-0.2)
print("")
print("Match: 1, Mismatch: -1, Gap open: -0.5, Gap extend: -0.2")
for align in alignments:
    print(align[0])
    print(align[1])
    print("Score: {}, Start: {}, End: {}".format(align[2],
                                                 align[3],
                                                 align[4]))
    print("")
```

http://biopython.org



Edit this page on GitHub

Biopython

See also our News feed and Twitter.

Introduction

Biopython is a set of freely available tools for biological computation written in Python by an international team of developers.

It is a distributed collaborative effort to develop Python libraries and applications which address the needs of current and future work in bioinformatics. The source code is made available under the Biopython License, which is extremely liberal and compatible with almost every license in the world.

We are a member project of the Open Bioinformatics Foundation (OBF), who take care of our domain name and hosting for our mailing list etc. The OBF used to host our developement repository, issue tracker and website but these are now on GitHub.

This wiki will help you download and install Biopython, and start using the libraries and tools.

Get Started	Get help	Contribute
Download Biopython	Tutorial (PDF)	What's being worked on
Installation help (PDF)	Documentation on this wiki	Developing on Github
	Cookbook (working examples)	Google Summer of Code
	Discuss and ask questions	Report bugs (older issues)

The latest release is Biopython 1.70, released on 10 July 2017.

http://biopython.org/DIST/docs/api/

Check:

Seq
SeqRecord
MultipleSeqAlignment

Table of Contents

Everything

Modules

Bio
Bio.Affy
Bio.Affy.CelFile
Bio.Align
Bio.Align.AlignInfo
Bio.Align.Applications
Bio.Align.Applications._ClustalOmega
Bio.Align.Applications._Clustalw
Bio.Align.Applications._Dialign
Bio.Align.Applications._MSAProbs
Bio.Align.Applications._Mafft

Everything

All Classes

Bio.Affy.CelFile.ParserError
Bio.Affy.CelFile.Record
Bio.Align.AlignInfo.PSSM
Bio.Align.AlignInfo.SummaryInfo
Bio.Align.Applications._ClustalOmega.ClustalOmegaCom
Bio.Align.Applications._Clustalw.ClustalwCommandline
Bio.Align.Applications._Dialign.DialignCommandline
Bio.Align.Applications._MSAProbs.MSAProbsCommandl
Bio.Align.Applications._Mafft.MafftCommandline
Bio.Align.Applications._Muscle.MuscleCommandline
Bio.Align.Applications._Prank.PrankCommandline
Bio.Align.Applications._Probcons.ProbconsCommandline
Bio.Align.Applications._TCoffee.TCoffeeCommandline
Bio.Align.MultipleSeqAlignment
Bio.AlignIO.ClustalIO.ClustalIterator
Bio.AlignIO.ClustalIO.ClustalWriter
Bio.AlignIO.EmbossIO.EmbossIterator
Bio.AlignIO.EmbossIO.EmbossWriter
Bio.AlignIO.Interfaces.AlignmentIterator
Bio.AlignIO.Interfaces.AlignmentWriter
Bio.AlignIO.Interfaces.SequentialAlignmentWriter
Bio.AlignIO.MafIO.MafIndex
Bio.AlignIO.MafIO.MafWriter
Bio.AlignIO.MauveIO.MauveIterator
Bio.AlignIO.MauveIO.MauveWriter
Bio.AlignIO.NexusIO.NexusWriter
Bio.AlignIO.PhylipIO.PhylipIterator
Bio.AlignIO.PhylipIO.PhylipWriter
Bio.AlignIO.PhylipIO.RelaxedPhylipIterator
Bio.AlignIO.PhylipIO.RelaxedPhylipWriter
Bio.AlignIO.PhylipIO.SequentialPhylipIterator
Bio.AlignIO.PhylipIO.SequentialPhylipWriter

Trees Indices **Help**

[Module Hierarchy | Class Hierarchy]

Module Hierarchy

- **Bio**: *Collection of modules for dealing with biological data in Python.*
  - **Bio.Affy**: *Deal with Affymetrix related data such as cel files.*
    - **Bio.Affy.CelFile**: *Reading information from Affymetrix CEL files version 3 and 4.*
  - **Bio.Align**: *Code for dealing with sequence alignments.*
    - **Bio.Align.AlignInfo**: *Extract information from alignment objects.*
    - **Bio.Align.Applications**: *Alignment command line tool wrappers.*
      - **Bio.Align.Applications._ClustalOmega**: *Command line wrapper for the multiple alignment program Clustal Omega.*
      - **Bio.Align.Applications._Clustalw**: *Command line wrapper for the multiple alignment program Clustal W.*
      - **Bio.Align.Applications._Dialign**: *Command line wrapper for the multiple alignment program DIALIGN2-2.*
      - **Bio.Align.Applications._MSAProbs**: *Command line wrapper for the multiple sequence alignment program MSAProbs.*
      - **Bio.Align.Applications._Mafft**: *Command line wrapper for the multiple alignment programme MAFFT.*
      - **Bio.Align.Applications._Muscle**: *Command line wrapper for the multiple alignment program MUSCLE.*
      - **Bio.Align.Applications._Prank**: *Command line wrapper for the multiple alignment program PRANK.*
      - **Bio.Align.Applications._Probcons**: *Command line wrapper for the multiple alignment program PROBCONS.*
      - **Bio.Align.Applications._TCoffee**: *Command line wrapper for the multiple alignment program TCOFFEE.*
  - **Bio.AlignIO**: *Multiple sequence alignment input/output as alignment objects.*
    - **Bio.AlignIO.ClustalIO**: *Bio.AlignIO support for "clustal" output from CLUSTAL W and other tools.*
    - **Bio.AlignIO.EmbossIO**: *Bio.AlignIO support for "emboss" alignment output from EMBOSS tools.*
    - **Bio.AlignIO.FastaIO**: *Bio.AlignIO support for "fasta-m10" output from Bill Pearson's FASTA tools.*
    - **Bio.AlignIO.Interfaces**: *AlignIO support module (not for general use).*
    - **Bio.AlignIO.MafIO**: *Bio.AlignIO support for the "maf" multiple alignment format.*
    - **Bio.AlignIO.MauveIO**: *Bio.AlignIO support for "xmfa" output from Mauve/ProgressiveMauve.*
    - **Bio.AlignIO.NexusIO**: *Bio.AlignIO support for the "nexus" file format.*
    - **Bio.AlignIO.PhylipIO**: *AlignIO support for "phylip" format from Joe Felsenstein's PHYLIP tools.*
    - **Bio.AlignIO.StockholmIO**: *Bio.AlignIO support for "stockholm" format (used in the PFAM database).*
  - **Bio.Alphabet**: *Alphabets used in Seq objects etc to declare sequence type and letters.*
    - **Bio.Alphabet.IUPAC**: *Standard nucleotide and protein alphabets defined by IUPAC.*
    - **Bio.Alphabet.Reduced**: *Reduced alphabets which lump together several amino-acids into one letter.*
  - **Bio.Application**: *General mechanisms to access applications in Biopython.*
  - **Bio.Blast**: *Code for dealing with BLAST programs and output.*
    - **Bio.Blast.Applications**: *Definitions for interacting with BLAST related applications.*
    - **Bio.Blast.NCBIStandalone**: *Code for calling standalone BLAST and parsing plain text output (DEPRECATED).*
    - **Bio.Blast.NCBIWWW**: *Code to invoke the NCBI BLAST server over the internet.*
    - **Bio.Blast.NCBIXML**: *Code to work with the BLAST XML output.*
    - **Bio.Blast.ParseBlastTable**: *A parser for the NCBI blastpgp version 2.2.5 output format. Currently only supports the '-m 9' option, (table w/ annotations). Returns a BlastTableRec instance*
    - **Bio.Blast.Record**: *Record classes to hold BLAST output.*
  - **Bio.CAPS**: *Cleaved amplified polymorphic sequence (CAPS) markers.*
  - **Bio.Cluster**: *Cluster Analysis.*
    - **Bio.Cluster.cluster**: *C Clustering Library*
  - **Bio.Compass**: *Code to deal with COMPASS output, a program for profile/profile comparison.*
  - **Bio.Crystal**: *Represent the NDB Atlas structure (a minimal subset of PDB format).*
  - **Bio.Data**: *Collections of various bits of useful biological data.*
    - **Bio.Data.CodonTable**: *Codon tables based on those from the NCBI.*
    - **Bio.Data.IUPACData**: *Information about the IUPAC alphabets.*
    - **Bio.Data.SCOPData**: *Additional protein alphabets used in the SCOP database and PDB files.*
  - **Bio.DocSQL**: *Bio.DocSQL: easy access to DB API databases (DEPRECATED).*

Installing biopython

```
import Bio
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-1-f227b1b7f7f3> in <module>()
----> 1 import Bio

ImportError: No module named 'Bio'
```

In windows installing Biopython should be as easy as opening the command prompt as admininstrator (typing `cmd` and then right clicking on the link choosing run as admininstrator) and then `pip3 install biopython`.

In linux `sudo pip3 install biopython` will install biopython for python3 up to python3.5. On python 3.6, the command is: `python3.6 -m pip install biopython`.

http://sciprolab1.readthedocs.io/en/latest/practical10.html

Exercises

1. Write a python function that reads a genebank file given in input and prints off the following information:
   1. Identifier, name and description;
   2. The first 100 characters of the sequence;
   3. Number of external references (dbxrefs) and ids of the external refs.
   4. The name of the organism (hint: check the annotations dictionary at the key "organism")
   5. Retrieve and print all (if any) associated publications (hint: annotation dictionary, key:"references")
   6. Retrieve and print all the locations of "CDS" features of the sequence (hint: check the features)

Hint: go back and check the details of the `SeqRecord` object.

Test the program downloading some files from genebank like this

Show/Hide Solution

2. Write a python program that loads a pfam file (stockholm format .sth) and reports for each record of the alignment:
   1. the description of the entry
   2. the start and end points
   3. the number of gaps and the % of gaps on the total length of the alignment
   4. any external database references (dbxrefs), comma separated (

Print these information to the screen. Finally, write this information in a tab separated file (.tsv) having the following format: `#Description\tstart\tend\tnum_gaps\tpercentage_gaps\tdbxrefs`.

Show/Hide Solution

3. Load the contigs present in the filtered_contigs.fasta file and translate each DNA sequence into the corresponding protein. Write the translated proteins in another .fasta file (e.g. filtered_contigs_translated.fasta).