

# Scientific Programming

## Practical 10

---

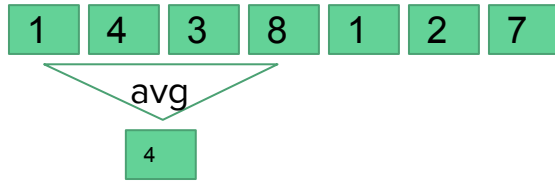
### Introduction

Luca Bianco - Academic Year 2020-21  
luca.bianco@fmach.it

# Ex 6

6. Implement a function `movingAvg(A, n)` where A is a numpy one dimensional array and n is the window size on which computing the average, that outputs the moving average over a numpy one dimensional array. Es. A = [1,2,3,4,5] `movingAvg(A,2) = [1.5, 2.5, 3.5, 4.5]`, while `movingAvg(A,3) = [2, 3, 4]` without using for loops. Hint: use cumsum and clever slicing. Assess the smoothing effect of the moving average by creating a numpy array containing a sinusoidal wave with some additional noise (i.e. use `np.sin` and `np.random.rand`) and testing several values of the window size n.

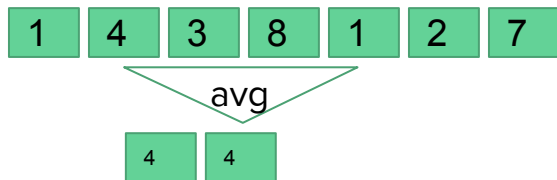
Example (win:4):



# Ex 6

6. Implement a function `movingAvg(A, n)` where A is a numpy one dimensional array and n is the window size on which computing the average, that outputs the moving average over a numpy one dimensional array. Es.  $A = [1, 2, 3, 4, 5]$  `movingAvg(A, 2) = [1.5, 2.5, 3.5, 4.5]`, while `movingAvg(A, 3) = [2, 3, 4]` without using for loops. Hint: use cumsum and clever slicing. Assess the smoothing effect of the moving average by creating a numpy array containing a sinusoidal wave with some additional noise (i.e. use `np.sin` and `np.random.rand`) and testing several values of the window size n.

Example (win:4):

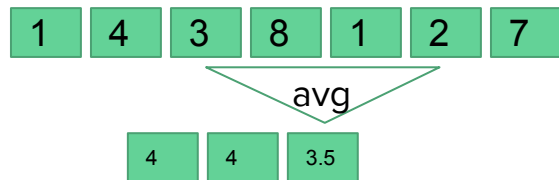


[4. 4. 3.5 4.5]

## Ex 6

6. Implement a function `movingAvg(A, n)` where A is a numpy one dimensional array and n is the window size on which computing the average, that outputs the moving average over a numpy one dimensional array. Es. A = [1,2,3,4,5] `movingAvg(A,2) = [1.5, 2.5, 3.5, 4.5]`, while `movingAvg(A,3) = [2, 3, 4]` without using for loops. Hint: use cumsum and clever slicing. Assess the smoothing effect of the moving average by creating a numpy array containing a sinusoidal wave with some additional noise (i.e. use `np.sin` and `np.random.rand`) and testing several values of the window size n.

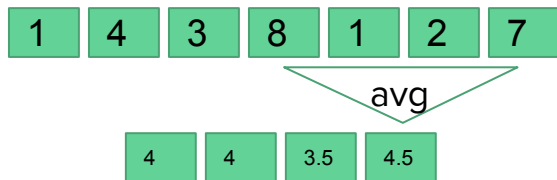
Example (win:4):



# Ex 6

6. Implement a function `movingAvg(A, n)` where A is a numpy one dimensional array and n is the window size on which computing the average, that outputs the moving average over a numpy one dimensional array. Es. A = [1,2,3,4,5] `movingAvg(A,2) = [1.5, 2.5, 3.5, 4.5]`, while `movingAvg(A,3) = [2, 3, 4]` without using for loops. Hint: use cumsum and clever slicing. Assess the smoothing effect of the moving average by creating a numpy array containing a sinusoidal wave with some additional noise (i.e. use `np.sin` and `np.random.rand`) and testing several values of the window size n.

Example (win:4):



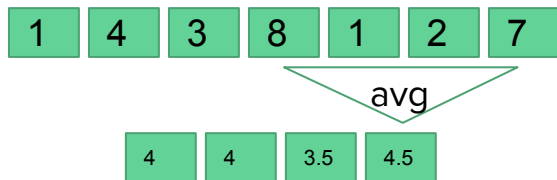
Cumulative sum:



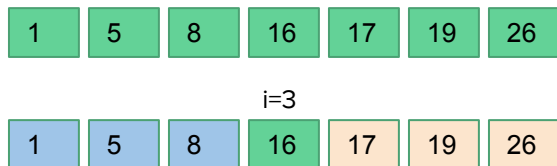
# Ex 6

6. Implement a function `movingAvg(A, n)` where A is a numpy one dimensional array and n is the window size on which computing the average, that outputs the moving average over a numpy one dimensional array. Es.  $A = [1, 2, 3, 4, 5]$  `movingAvg(A, 2) = [1.5, 2.5, 3.5, 4.5]`, while `movingAvg(A, 3) = [2, 3, 4]` without using for loops. Hint: use cumsum and clever slicing. Assess the smoothing effect of the moving average by creating a numpy array containing a sinusoidal wave with some additional noise (i.e. use `np.sin` and `np.random.rand`) and testing several values of the window size n.

Example (win:4):



Cumulative sum:

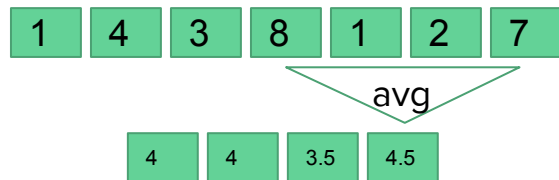


**Clever bit:** when we move to the right with the window we need to disregard (i.e. subtract) the blue elements one after the other

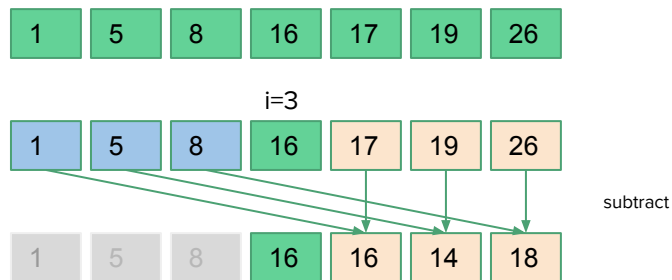
# Ex 6

6. Implement a function `movingAvg(A, n)` where A is a numpy one dimensional array and n is the window size on which computing the average, that outputs the moving average over a numpy one dimensional array. Es. A = [1,2,3,4,5] `movingAvg(A,2) = [1.5, 2.5, 3.5, 4.5]`, while `movingAvg(A,3) = [2, 3, 4]` without using for loops. Hint: use cumsum and clever slicing. Assess the smoothing effect of the moving average by creating a numpy array containing a sinusoidal wave with some additional noise (i.e. use `np.sin` and `np.random.rand`) and testing several values of the window size n.

Example (win:4):



Cumulative sum:



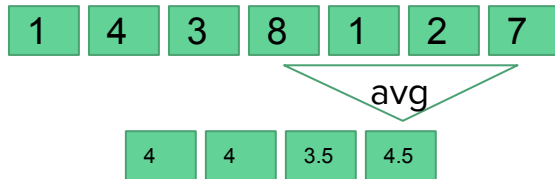
**Clever bit:** when we move to the right with the window we need to disregard (i.e. subtract) the blue elements one after the other

Let's subtract them after the starting point (i=3)

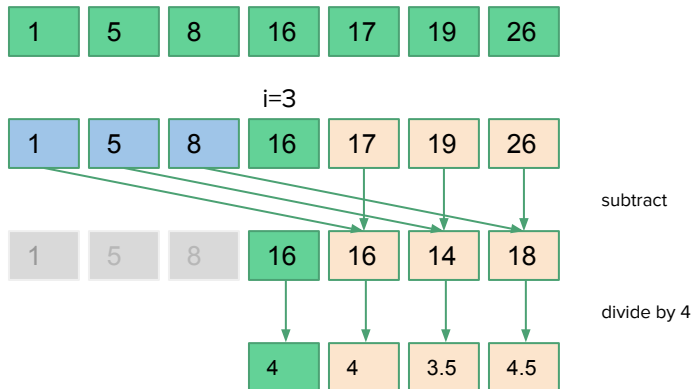
# Ex 6

6. Implement a function `movingAvg(A, n)` where A is a numpy one dimensional array and n is the window size on which computing the average, that outputs the moving average over a numpy one dimensional array. Es. A = [1,2,3,4,5] `movingAvg(A,2) = [1.5, 2.5, 3.5, 4.5]`, while `movingAvg(A,3) = [2, 3, 4]` without using for loops. Hint: use cumsum and clever slicing. Assess the smoothing effect of the moving average by creating a numpy array containing a sinusoidal wave with some additional noise (i.e. use `np.sin` and `np.random.rand`) and testing several values of the window size n.

Example (win:4):



Cumulative sum:



**Clever bit:** when we move to the right with the window we need to disregard (i.e. subtract) the blue elements one after the other

Let's subtract them after the starting point ( $i=3$ )

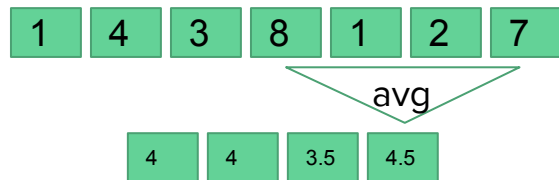
Finally, let's compute the mean value (i.e. divide by 4)



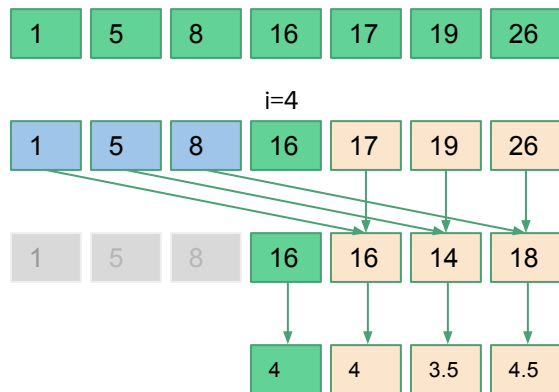
# Ex 6

6. Implement a function `movingAvg(A, n)` where A is a numpy one dimensional array and n is the window size on which computing the average, that outputs the moving average over a numpy one dimensional array. Es. A = [1,2,3,4,5] | `movingAvg(A,2)` = [1.5, 2.5, 3.5, 4.5], while `movingAvg(A,3)` = [2, 3, 4] without using for loops. Hint: use `cumsum` and clever slicing. Assess the smoothing effect of the moving average by creating a numpy array containing a sinusoidal wave with some additional noise (i.e. use `np.sin` and `np.random.rand`) and testing several values of the window size n.

Example:



Cumulative sum:



```
def movingAvg(v, n):
    """computes the moving average of n values in the array v"""
    cs = np.cumsum(v)
    #print(cs)
    #print("\t", cs[n:])
    #print("\t", cs[:-n])
    cs[n:] = cs[n:] - cs[:-n]
    #print("CS:", cs)
    return cs[n - 1:] / n
```

```
X = np.arange(0, 5, 0.005)
#B = np.random(100)
B = np.sin(2*np.pi*X)

#Let's add random numbers uniformly distributed in [0,1)
B += np.random.random_sample(1000)
#B += np.random.rand(1000)
C = movingAvg(B,5)
D = movingAvg(B,10)
E = movingAvg(B,50)
#X = np.arange(0,B.shape[0])
```

```
plt.plot(X,B)
plt.title("No moving average")
plt.show()
plt.close()
plt.plot(np.arange(0,C.shape[0]),C)
plt.title("Window size: 5")
plt.show()
plt.close()
plt.plot(np.arange(0,D.shape[0]),D)
plt.show()
plt.close()
plt.plot(np.arange(0,E.shape[0]),E)
plt.show()
```

subtract

divide by 4

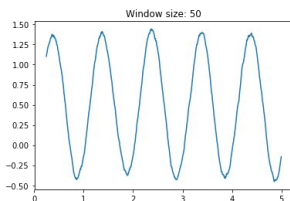
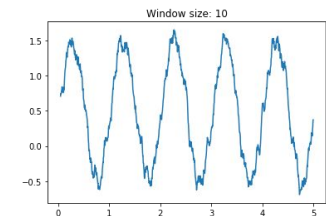
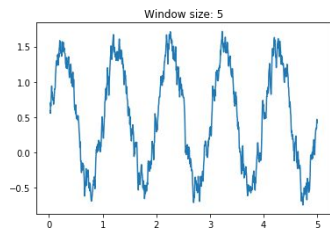
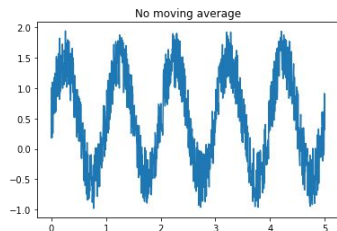
# Ex 6

6. Implement a function `movingAvg(A,n)` where A is a numpy one dimensional array and n is the window size on which computing the average, that outputs the moving average over a numpy one dimensional array. Es. `A = [1,2,3,4,5]` | `movingAvg(A,2) = [1.5, 2.5, 3.5, 4.5]`, while `movingAvg(A,3) = [2, 3, 4]` without using for loops. Hint: use `cumsum` and clever slicing. Assess the smoothing effect of the moving average by creating a numpy array containing a sinusoidal wave with some additional noise (i.e. use `np.sin` and `np.random.rand`) and testing several values of the window size n.

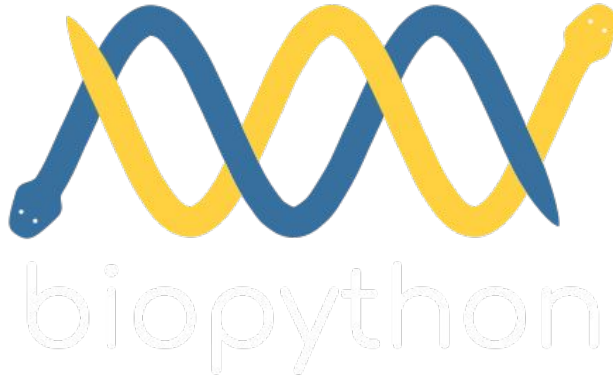
```
def movingAvg(v, n):  
    """computes the moving average of n values in the array v"""  
    cs = np.cumsum(v)  
    #print(cs)  
    #print("\t",cs[n:])  
    #print("\t", cs[:-n])  
    cs[n:] = cs[n:] - cs[:-n]  
    #print("CS:", cs)  
    return cs[n - 1:] / n
```

```
X = np.arange(0, 5, 0.005)  
#B = np.random(100)  
B = np.sin(2*np.pi*X)  
  
#Let's add random numbers uniformly distributed in [0,1)  
B += np.random.random_sample(1000)  
#B += np.random.rand(1000)  
C = movingAvg(B,5)  
D = movingAvg(B,10)  
E = movingAvg(B,50)  
#X = np.arange(0,B.shape[0])
```

```
plt.plot(X,B)  
plt.title("No moving average")  
plt.show()  
plt.close()  
plt.plot(X[4:],C) # X has 1000- 5 -1 elements  
plt.title("Window size: 5")  
plt.show()  
plt.close()  
plt.title("Window size: 10")  
plt.plot(X[9:],D) # X has 1000- 9 -1 elements  
plt.show()  
plt.close()  
plt.title("Window size: 50")  
plt.plot(X[49:],E) # X has 1000- 49 -1 elements  
plt.show()
```



# Biopython

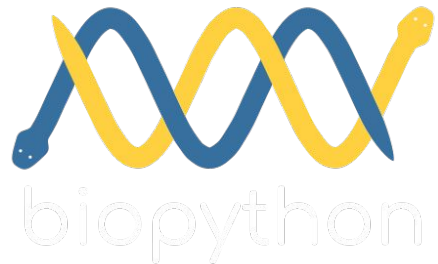


## FROM Biopython's website:

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.

More material at:

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

# Seq objects

Seq objects are more powerful than strings 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**

Some options:

**Bio.Alphabet.generic\_dna**,  
**Bio.Alphabet.generic\_protein**,  
**Bio.Alphabet.ThreeLetterProtein**,  
**Bio.Alphabet.generic\_alphabet**, ...

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:

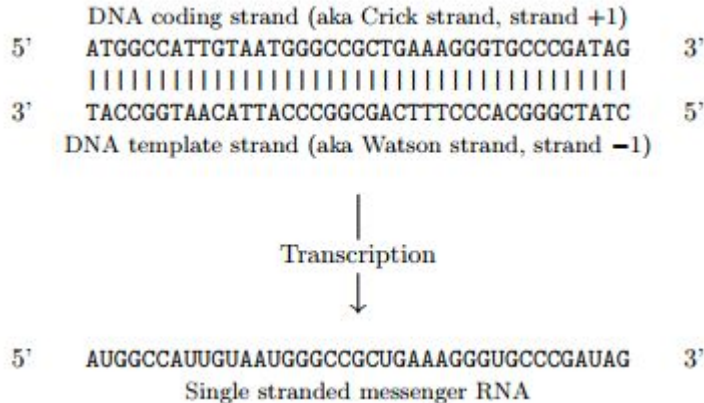
print("Reversed: ", dna_seq[::-1])
#from Seq to string:
dna_str = str(dna_seq)
print("As string:", dna_str)
print(type(dna_str))
```

```
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  
(remember Seq are immutable!)



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 in upper case

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

(**ex. use** from Bio.SeqUtils import molecular\_weight):

`SeqUtils.GC(Seq)` computes GC content

`SeqUtils.molecular_weight(Seq)` computes the molecular weight of the seq

....

Check out: <http://biopython.org/DIST/docs/api/>

# Seq objects

Biopython provides several  
methods working on Seq  
objects  
(remember Seq are immutable!)

```
ATGGCCATTGTAATGGGCGCTGAAAGGGTGCCCGATAG
AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG
```

```
IUPACUnambiguousRNA()
```

```
... and back
ATGGCCATTGTAATGGGCGCTGAAAGGGTGCCCGATAG
```

```
Translation to protein:
MAIVMGR*KGAR*
```

```
Up to first stop:
MAIVMGR
```

```
Mitochondrial translation: (TGA is W!)
MAIVMGRWKGAR*
```

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

coding_dna = Seq("ATGGCCATTGTAATGGGCGCTGAAAGGGTGCCCGATAG",
                  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("Mitochondrial translation: (TGA is W!)")
mit_prot = mrna.translate(table=2)
print(mit_prot)
```

# Sequence annotations

The **SeqRecord** object is used to store annotations associated to sequences. They might provide:

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.

```
>gi|45478711|ref|NC_005816.1| Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence
TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGGGGTAATCTGCTCTCC
TGATTACGAGAGTTTATGGTCACTTTTGAGACAGTTATGGAATTAATCCTGCACAAGCAGGGAATG
AGTAGCCGGGCGATTGCCAGAGAACTGGGGATCTCCCGCAATACCGTTAAACGTTATTTGCAGGCAAAAT
CTGAGCCGCCAAAATATACGCCGCGACCTGCTGTTGCTTCACTCCTGGATGAATACCGGGATTATATTG
TCAACGCATCGCCGATGCTCATCCTTACAAAATCCCGGCAACGGTAATCGCTCGCGAGATCAGAGACCAG
GGATATCGTGGCGGAATGACCATTCTCAGGGCATTCACTCGTTCTCTCGGTTCTCAGGAGCAGGAGC
CTGCCGTTTCGGTTCGAAACTGAACCGGACGACAGATGCAGTTGACTGGGGCACTATGCGTAATGGTCG
CTCACCGCTTCACGTGTTGCTGTTCTCGGATACAGCCGAATGCTGTACATCGAATTCAGTACAAT
ATGCGTTATGACACGCTGGAGACCTGCCATCGTAATGCGTTCGCTTCTTTGGTGGTGTGCCGCGGAAG
TGTTGTATGACAATATGAAAACGTGGG....|
```

<https://www.ncbi.nlm.nih.gov>

← → ↻ [https://www.ncbi.nlm.nih.gov/nucleotide/NC\\_005816](https://www.ncbi.nlm.nih.gov/nucleotide/NC_005816)

NCBI Resources How To

Nucleotide  NC005816 [Advanced](#)

[Learn more](#) about upcoming changes to the Nucleotide, EST, and GSS databases.

GenBank [Send to:](#)

### Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence

NCBI Reference Sequence: NC\_005816.1

[FASTA](#) [Graphics](#)

[Go to:](#) (v)

LOCUS	NC_005816	9609 bp	DNA	circular	CON 11-JAN-2018
DEFINITION	Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.				
ACCESSION	NC_005816				
VERSION	NC_005816.1				
DBLINK	BioProject: <a href="#">PRJNA224116</a> BioSample: <a href="#">SAMN02602970</a> Assembly: <a href="#">GCF_000007885.1</a>				
KEYWORDS	RefSeq.				
SOURCE	Yersinia pestis biovar Microtus str. 91001				
ORGANISM	<a href="#">Yersinia pestis biovar Microtus str. 91001</a> Bacteria; Proteobacteria; Gammaproteobacteria; Enterobacterales; Yersiniaceae; Yersinia.				
REFERENCE	1 (bases 1 to 9609)				
AUTHORS	Zhou,D., Tong,Z., Song,Y., Han,Y., Pei,D., Pang,X., Zhai,J., Li,M., Cui,B., Qi,Z., Jin,L., Dai,R., Du,Z., Wang,J., Guo,Z., Wang,J., Huang,P. and Yang,R.				
TITLE	Genetics of metabolic variations between Yersinia pestis biovars				

# 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.

Formats available:

<https://biopython.org/wiki/SeqIO>

**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 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
GAGGGGTTTAGTTCCTCACTCGCAAAGCAAAGATACATAAATTTAGAA...
```

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

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

# SeqIO

With  
SimpleFastaParser...

```
labels = ["1st", "2nd", "3rd"]

with open("file_samples/contigs82.fasta") as cont_handle:
    for l in labels:
        ID, seq = next(SimpleFastaParser(cont_handle))

        print(l, "entry:")
        print(ID, " has size ", len(seq))
        print(seq[:50]+"...")
        print("")
```

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

2nd entry:  
MDC013284.379 has size 5173  
TATCGTTTCCTCTGAGTAGAATATCGTTATAACAAGATTTTTTTTTTCT...

3rd entry:  
MDC018185.241 has size 23761  
AAAACGAGGAAAATCCATCTTGATGAACAGGAGATGCGGAGGAAAAAAT...



# 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.

```

Q5E940_BOVIN -----MPREDRATWKSNYFLKIIQLDDDPKCFIVGADNVGSKOMQDIHMSLRGK-AVYLMGKNTMMRKAIRGHLNNH--PALE 76
RLA0_HUMAN -----MPREDRATWKSNYFLKIIQLDDDPKCFIVGADNVGSKOMQDIHMSLRGK-AVYLMGKNTMMRKAIRGHLNNH--PALE 76
RLA0_MOUSE -----MPREDRATWKSNYFLKIIQLDDDPKCFIVGADNVGSKOMQDIHMSLRGK-AVYLMGKNTMMRKAIRGHLNNH--PALE 76
RLA0_RAT -----MPREDRATWKSNYFLKIIQLDDDPKCFIVGADNVGSKOMQDIHMSLRGK-AVYLMGKNTMMRKAIRGHLNNH--PALE 76
RLA0_CHICK -----MPREDRATWKSNYFMKIIQLDDDPKCFVVGADNVGSKOMQDIHMSLRGK-AVYLMGKNTMMRKAIRGHLNNH--PALE 76
RLA0_RAMSY -----MPREDRATWKSNYFLKIIQLDDDPKCFIVGADNVGSKOMQDIHMSLRGK-AVYLMGKNTMMRKAIRGHLNNH--PALE 76
Q7ZUG3_BRARE -----MPREDRATWKSNYFLKIIQLDDDPKCFIVGADNVGSKOMQDIHMSLRGK-AVYLMGKNTMMRKAIRGHLNNH--PALE 76
RLA0_ICTPU -----MPREDRATWKSNYFLKIIQLDDDPKCFIVGADNVGSKOMQDIHMSLRGK-AVYLMGKNTMMRKAIRGHLNNH--PALE 76
RLA0_DROME -----HYRENKAAMKAQYFIKVVYLFDEFPPKCFIVGADNVGSKOMQDIHMSLRGK-AVYLMGKNTMMRKAIRGHLNNH--PALE 76
RLA0_DICDI -----MSGAG-SKREKILFIEKATKLFITTDKMIYAEADVFVGSOLQIKRSIRGI-GAVLMGKNTMIRKVVIRDLADSK--PELD 75
Q54LP0_DICDI -----MSGAG-SKRENVFIEKATKLFITTDKMIYAEADVFVGSOLQIKRSIRGI-GAVLMGKNTMIRKVVIRDLADSK--PELD 75
RLA0_PLAFB -----MAKLSKQKKQMYIEKLSSLIQQYSKILIVHVDNVGSKOMQDIHMSLRGK-AVYLMGKNTMIRKVVIRDLADSK--PELD 76
RLA0_SULAC -----MIGLAVTTTKKIAKWKYDEVAELTSLKTKNTIIIANIIGFPPADKLHEIRKKLRGK-ADIKVTKNHLPNIAKLNAG--YDK 79
RLA0_SULTO -----MRIMAVITQERKIAKWKIEEVKELEKLRKNTIIIANIIGFPPADKLHEIRKKLRGK-ADIKVTKNHLPNIAKLNAG--YDK 80
RLA0_SULSO -----MKRLALALQKRYASWKELEVKELETKIKNSNTILGNLEGFPPADKLHEIRKKLRGK-ADIKVTKNHLPNIAKLNAG--YDK 80
RLA0_AERPE MSYVSLVGMQYKREKIPENKTLMLRELELFSKRVVFLADLTGTPFVVRVYKKEKWKKK-YHMMVAKRRIILBAMKAAGLE---LDDN 86
RLA0_PYRAE HMLAIGKRRYVTRQIPARKYKIVSEATLLQKIPYVFLFDLHLSIRILHEVYRLARY-GVIRIKIIDLKFKIAFTKYVGG---IPAE 85
RLA0_METAC -----MAEERNHTEHIPQWKDEIENIKELIQSHKVFCHVRIEGILATKIQIRDLKDV-AVLKVSNTLTERALNQLG---ETIP 78
RLA0_METMA -----MAEERNHTEHIPQWKDEIENIKELIQSHKVFCHVRIEGILATKIQIRDLKDV-AVLKVSNTLTERALNQLG---ETIP 78
RLA0_ARCFU -----MAAVRG-----PPEYVYRAVEEIKRMISSKPYVAIVSFRNVFAGOMKIEREFGK-AEIKVVKNTLTERALDALG---GDYL 75
RLA0_METKA MAYKAKGQPPSYEKKVAEMRREVEKELMDIEYVGLDLEGIAPOLQEIYAKLRERDIIIMRBRKTLMLRAIEEKLDER--PELE 88
RLA0_METTH -----MAHVAEWKKKEVEELAKLIEKSTPYIALVDVSSMPAYPLSQMRRLIRENGLLIVRSNTLLELAIKKAAKELGKPELE 77
RLA0_METTL -----MITAESEHKIAPWKIEEVNKLKELLKNGQIYALVDMMEYPAVQLQEIIRDKIR-GTMTLKMRNTLLELAIKKAAKELGKPELE 82
RLA0_METVA -----MIDAKSEHKIAPWKIEEVNKLKELLKNGQIYALVDMMEYPAVQLQEIIRDKIR-GTMTLKMRNTLLELAIKKAAKELGKPELE 82
RLA0_METJA -----METKVAHVAPWKIEEVNKLKELLKNGQIYALVDMMEYPAVQLQEIIRDKIR-GTMTLKMRNTLLELAIKKAAKELGKPELE 81
RLA0_PYRAB -----MAHVAEWKKKEVEELAKLIEKSTPYIALVDVSSMPAYPLSQMRRLIRENGLLIVRSNTLLELAIKKAAKELGKPELE 77
RLA0_PYRHO -----MAHVAEWKKKEVEELAKLIEKSTPYIALVDVSSMPAYPLSQMRRLIRENGLLIVRSNTLLELAIKKAAKELGKPELE 77
RLA0_PYRFU -----MAHVAEWKKKEVEELAKLIEKSTPYIALVDVSSMPAYPLSQMRRLIRENGLLIVRSNTLLELAIKKAAKELGKPELE 77
RLA0_PYRKO -----MAHVAEWKKKEVEELAKLIEKSTPYIALVDVSSMPAYPLSQMRRLIRENGLLIVRSNTLLELAIKKAAKELGKPELE 76
RLA0_HALMA -----MSAESERKTETIPWKQEEVDIVMIESYSGVYVNIAGIPRGLDMMERDLHGT-AELVRSNTLLELAIKKAAKELGKPELE 79
RLA0_HALVO -----MSESEVRQTEVIPQWKREYVDLVDIESYSGVYVNIAGIPRGLDMMERDLHGT-AELVRSNTLLELAIKKAAKELGKPELE 79
RLA0_HALSA -----MSAEQRTEETEEVPWKQEEVDIVMIESYSGVYVNIAGIPRGLDMMERDLHGT-AELVRSNTLLELAIKKAAKELGKPELE 79
RLA0_THEAC -----MKESYQQKELVNEITQIKASRSVAIVDTAGIRTRQIDIRGKNGRK-INLVKIKKLLFPALNLEGD---EKLS 72
RLA0_THEVO -----MRKINPKKEIVSELAQDITKSKAVAIVDIKGVRTRMODIRAKNHDK-VKIKVVKKLLFPALNLEGD---EKLT 72
RLA0_PICTO -----MTPEAQWKIDFVKHLENEINERKVAIVSISIKELRNNTFKIKNSIDDK-ARIKVRARLRLAILENGK---NHIV 72
ruler 1.....10.....20.....30.....40.....50.....60.....70.....80.....90

```

In Biopython, each row is a `SeqRecord` object and alignments are stored in an object `MultipleSeqAlignment`

# Parsing MSAs: AlignIO

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,...).

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

Each `SeqRecord` contains several information like the **ID**, **Name**, **Description**, **Number of features**, **start**, **end** and **sequence**.

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

```
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("")
```

```
AG01 SCHPO/500-799 S:500 E:799
YLFFILDK-NSPEP-YGSIKRVNNTMLGVP SQCAISKHILQS-----KPQYCANLGMKINVKVGGIN-CSLIPKSNP----L

AG06 ARATH/541-851 S:541 E:851
FILCILPERKTSOI-YGPWKIKCLTEEGIHTQCICPIKI-----SDQYLTNVLLKINSKLGGIN-SLLGIEYSYNIPLI

AG04 ARATH/577-885 S:577 E:885
FILCVLPDKKNSDL-YGPWKKNLTFEGIVTQCMAPTRQPN-----QYLTNLLLKINAKLGLN-SMLSVERTPAFTVI

TAG76 CAEL/660-966 S:660 E:966
CIIIVLQS-KNSDI-YMTVKEQSDIVHGIMSQCVL MNVSRP-----TPATCANIVLKLNMKGGIN--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")
```

**Example:** Convert the seed alignment of the [Piwi \(PF02171\)](#) family stored in the pfam (stockholm) format [PF02171\\_seed.sth](#) into phylip format. Print some stats on the data.

N. of seq: 16  
 Len of seq: 395  
 1 multiple alignments converted to phylip

```
# STOCKHOLM 1.0
#=GS AG01_SCHPO/500-799 AC 074957.1
#=GS AG06_ARATH/541-851 AC 048771.2
#=GS AG04_ARATH/577-885 AC 092VDS.2
#=GS TAG76_CAEL/660-966 AC P34681.2
#=GS 016720_CAEL/566-867 AC 016720.2
#=GS 062275_CAEL/594-924 AC 062275.1
#=GS YQ53_CAEL/650-977 AC 009249.1
#=GS NRDE3_CAEL/673-1001 AC 021691.1
#=GS Q17567_CAEL/397-708 AC 017567.1
#=GS AUB_DROME/555-852 AC 076922.1
#=GS PIWI_DROME/538-829 AC 09VKM1.1
#=GS PIWI_HUMAN/555-847 AC 096394.1
#=GS PIWI_ARCFU/110-406 AC 028951.1
#=GS PIWI_ARCFU/110-406 DR PDB: 2W42 B; 110-406;
#=GS PIWI_ARCFU/110-406 DR PDB: 1YTU B; 110-406;
#=GS PIWI_ARCFU/110-406 DR PDB: 2BGG B; 110-406;
#=GS PIWI_ARCFU/110-406 DR PDB: 1W9H A; 110-406;
#=GS PIWI_ARCFU/110-406 DR PDB: 2BGG A; 110-406;
#=GS PIWI_ARCFU/110-406 DR PDB: 1YTU A; 110-406;
#=GS PIWI_ARCFU/110-406 DR PDB: 2W42 A; 110-406;
#=GS Y1321_METJA/426-699 AC 058717.1
#=GS 067434_AQUAE/419-694 AC 067434.1
#=GS 067434_AQUAE/419-694 DR PDB: 1YVU A; 419-694;
#=GS 067434_AQUAE/419-694 DR PDB: 2F8S A; 419-694;
#=GS 067434_AQUAE/419-694 DR PDB: 2F8T A; 419-694;
#=GS 067434_AQUAE/419-694 DR PDB: 2F8S B; 419-694;
#=GS 067434_AQUAE/419-694 DR PDB: 2NUB A; 419-694;
#=GS 067434_AQUAE/419-694 DR PDB: 2F8T B; 419-694;
#=GS AG010_ARATH/625-946 AC 09XGW1.1
AG01_SCHPO/500-799
YLFFILDK..NSPEP..YGSIKRVCNTMLGVPSQCAISKHILQS.....KPQYCANLGMKINVKGGIN..CSLIPKSNP....LGNVPTL.....ILGGDVYHPG|
AG06_ARATH/541-851
FILCILPERKTSOI..YGPWKKICLTEEGIHTQCICPIKI.....SDQYLTNVLLKINSKLGGIN..SLLGIEVSYNIPLINKIPTL.....TLGMDVSHGPF
AG04_ARATH/577-885
FILCVLPDKKNSDL..YGPWKKNNLTFEGIVTQCMAPTRQPN.....QYLTNLLKINAKLGGN..SMLSVERTPAFTVISKVPTI.....ILGMDVSHGPF
TAG76_CAEL/660-966
CIIIVLQS..KNSDI..YMTVKEQSDIVHGIMSQCVLMKNVSRP.....TPATCANIVLKLNMKGGIN..SRVADKITNKYLVDPQTH.....VVGIDVTHTP|
016720_CAEL/566-867
LIVVVLPG..-KTPI..YAEVKRVGDTVLGIATQCVQAKNAIRT.....
062275_CAEL/594-924
TFVFIITD..DSITT..LHQRKYMIEKDTKMIQVQDMKLSKALSV..-IN---A
DILVGIAR..EKKPD..VHDILKYFEESIGLQTIQLCQQTVDKMMGG---Q
TIVFGIIA..EKRPD..MHDILKYFEELKQQTIQISSSETADKFMRD---H
MLVVMLAD..DNKTR..YDSLKKYLCVCEPIPNQCVNLRRLAGKSKDGENK
IVMVVMRS..PNEEK..YSCIKKRTCVDRPVPSQVVTLLKVIAPRQKQP---T
LILCLVPN..DNAER..YSSIKKRGYVDRAVPTQVVTLLKTTKNRSL-----
IVVCLLSS..NRKDK..YDAIKKYLCTDCPTPSQCVVARTLGKQQT-----
GIMLVLPE..YNTPL..YYKLKSYLINS..-IPSQFMRYDILSNRNL-----
CFALITGKEKYKNDYYEILKKQFLDKLIIISQNILWENWRKDDK-----
LVIVFLIEEYPKVDP..YKSFLLYDFVKRELLKKMIPSVQILNRTLKN---E
LLAILLPD..NNGSL..YGDCLKRICETELGLISQCLTKHVFKI-----
AG010_ARATH/625-946
AG01_SCHPO/500-799
-KPQYCANLGMKINVKGGIN..CSLIPKSNP....LGNVPTL-----
```



```
from Bio import AlignIO

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

out = AlignIO.convert("file_samples/PF02171_seed.sth",
                      "stockholm",
                      "file_samples/PF05371_seed.aln",
                      "clustal")

print("N. of seq: {}\nLen of seq: {}".format(
    len(alignments),
    len(alignments[0])))

print("{} multiple alignments converted to phylip".format(out))
```

CLUSTAL X (1.81) multiple sequence alignment

```
AG01_SCHPO/500-799      YLFFILDK..NSPEP..YGSIKRVCNTMLGVPSQCAISKHILQS-----
AG06_ARATH/541-851      FILCILPERKTSOI..YGPWKKICLTEEGIHTQCICPIKI-----
AG04_ARATH/577-885      FILCVLPDKKNSDL..YGPWKKNNLTFEGIVTQCMAPTRQPN-----
TAG76_CAEL/660-966      CIIIVLQS..KNSDI..YMTVKEQSDIVHGIMSQCVLMKNVSRP-----
016720_CAEL/566-867     LIVVVLPG..-KTPI..YAEVKRVGDTVLGIATQCVQAKNAIRT-----
062275_CAEL/594-924     TFVFIITD..DSITT..LHQRKYMIEKDTKMIQVQDMKLSKALSV..-IN---A
YQ53_CAEL/650-977       DILVGIAR..EKKPD..VHDILKYFEESIGLQTIQLCQQTVDKMMGG---Q
NRDE3_CAEL/673-1001     TIVFGIIA..EKRPD..MHDILKYFEELKQQTIQISSSETADKFMRD---H
Q17567_CAEL/397-708     MLVVMLAD..DNKTR..YDSLKKYLCVCEPIPNQCVNLRRLAGKSKDGENK
AUB_DROME/555-852       IVMVVMRS..PNEEK..YSCIKKRTCVDRPVPSQVVTLLKVIAPRQKQP---T
PIWI_DROME/538-829      LILCLVPN..DNAER..YSSIKKRGYVDRAVPTQVVTLLKTTKNRSL-----
PIWL1_HUMAN/555-847     IVVCLLSS..NRKDK..YDAIKKYLCTDCPTPSQCVVARTLGKQQT-----
PIWI_ARCFU/110-406      GIMLVLPE..YNTPL..YYKLKSYLINS..-IPSQFMRYDILSNRNL-----
Y1321_METJA/426-699     CFALITGKEKYKNDYYEILKKQFLDKLIIISQNILWENWRKDDK-----
067434_AQUAE/419-694    LVIVFLIEEYPKVDP..YKSFLLYDFVKRELLKKMIPSVQILNRTLKN---E
AG010_ARATH/625-946     LLAILLPD..NNGSL..YGDCLKRICETELGLISQCLTKHVFKI-----

AG01_SCHPO/500-799      -KPQYCANLGMKINVKGGIN..CSLIPKSNP....LGNVPTL-----
```

# 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 *j*th character of alignment *i* as a string;
2. `SeqRecord[:, j]` returns all the *j*th 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
CIIIVLQS-KNSDI-YMTVKIPTPVYYADLVATRARCHVK
LIVVVLPG--KTPI-YAEVKIPAPAYYAHLVAFRARYHLV
TFVFIITD-DSITT-LHQRYLPTPLYVANEYAKRGRNLWN
DILVGIAR-EKKPD-VHDILVPDVLAAENLAKRGRNNYK
TIVFGIIA-EKRPD-MHDILIPNVSYAAQNLAQRHNNYK
MLVVMLAD-DNKTR-YDSLKVPAPCQYAHKLAFLTAQSLH
IVMVVMSR-PNEEK-YSCIKVPAVCHYAHKLAFLVAESIN
LILCLVPN-DNAER-YSSIKVPAVCQYAKKLATLVGTNLH
IVVCLLSS-NRKDK-YDAIKVPAVCQYAHKLAFLVGQSIH
GIMLVLPE-YNTPL-YYKLKLPVTVNYPKLVAGIIANVNR
CFALIIGKEKYKDNDDYIEILIPAPIHYADKFVKALGKNWK
LVIVFLEEYPKVDP-YKSFLPATVHYSKITKMLRGIE
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)  
YLFFILDK-N...  
FILCILPERK...  
FILCVLPDK...

`align[0:3,0:3]` first 3 cols of first 3 rows (SeqRecords):  
YLF  
FIL  
FIL

# 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()
```

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)
```

```
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).



# 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
```

The first letter is **the score for a match**  
the second letter is **the penalty for a gap**

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 param 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(alien[0])
```

```
print(align[0])
print(align[1])
```

```
print("Score: {}, Start: {}, End: {}".format(aligned[2],
                                              aligned[3],
                                              aligned[4]))
```

```
print("")
```

# http://biopython.org

[Edit this page on GitHub](#)



Python Tools for  
Computational  
Molecular Biology

[Documentation](#)

[Download](#)

[Mailing lists](#)

[News](#)

[Biopython Contributors](#)

[Scriptcentral](#)

[Source Code](#)

[GitHub project](#)

Biopython version 1.70  
© 2017. All rights reserved.

## 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 development 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.

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

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.ClustalOmegaCommandline](#)
- [Bio.Align.Applications.Clustalw.ClustalwCommandline](#)
- [Bio.Align.Applications.Dialign.DialignCommandline](#)
- [Bio.Align.Applications.MSAProbs.MSAProbsCommandline](#)
- [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.MafftIO.MafftIndex](#)
- [Bio.AlignIO.MafftIO.MafftWriter](#)
- [Bio.AlignIO.MauveIO.MauveIterator](#)
- [Bio.AlignIO.MauveIO.MauveWriter](#)
- [Bio.AlignIO.NexusIO.NexusWriter](#)
- [Bio.AlignIO.PhyliptIO.PhyliptIterator](#)
- [Bio.AlignIO.PhyliptIO.PhyliptWriter](#)
- [Bio.AlignIO.PhyliptIO.RelaxedPhyliptIterator](#)
- [Bio.AlignIO.PhyliptIO.RelaxedPhyliptWriter](#)
- [Bio.AlignIO.PhyliptIO.SequentialPhyliptIterator](#)
- [Bio.AlignIO.PhyliptIO.SequentialPhyliptWriter](#)

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.MafftIO:** Bio.AlignIO support for the "ma" multiple alignment format.
    - Bio.AlignIO.MauveIO:** Bio.AlignIO support for "mafa" output from Mauve/ProgressiveMauve.
    - Bio.AlignIO.NexusIO:** Bio.AlignIO support for the "nexus" file format.
    - Bio.AlignIO.PhyliptIO:** AlignIO support for "phylipt" 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 administrator (typing `cmd` and then right clicking on the link choosing run as administrator) 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`.



## 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 id 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. the number of external database references (dbxrefs), and the first 3 external references comma separated (hint: use join).

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