Python avancé

Sandra Dérozier & Thomas Duigou

Déroulé de la formation

- 2 journées : 9h30 - 17h
- 4 pauses de 15 minutes
- 2 pauses déjeuner d'1 heure
    - départ groupé pour la cantine sur le centre
- 20% théorie / 80% pratique
- N'hésitez pas à nous interrompre et à nous poser des questions, nous sommes là pour ça !

Objectifs pédagogiques

- A l'issue de la formation, vous serez capables de :
  - connaître les éléments avancés du langage de programmation Python,
  - les appliquer sur des cas concrets en bioinformatique,
  - être autonome dans la mise en place de tâches complexes visant à extraire et re-formater des données issues de fichiers textes, dans le cadre de traitement de données via le langage de programmation Python.

Programme de la formation

1. Introduction à la programmation
2. Introduction à Jupyter Notebooks et JupyterLab
3. Fonctions
4. Création de modules
5. Gestion des erreurs
6. Expressions régulières
7. Quelques modules de bioinformatique (numpy, pandas, biopython, matplotlib)

Tour de table

- Comment vous appelez-vous ?
- Quelles sont vos thématiques de recherche ? Sur quoi travaillez-vous ?
- Pourquoi effectuer cette formation ?

Éditeurs de texte

Dans le terminal avec Nano

Nano est un éditeur de texte en ligne de commande pour Unix et Linux.

Ouvrir un fichier

```
nano nom_du_fichier.ext
```

Naviguer dans le fichier

Utiliser les touches directionnelles pour se déplacer dans le fichier.

Éditer du texte

- Pour insérer du texte à l'endroit du curseur, il suffit de taper.
- Pour supprimer du texte, utilisez la touche `Suppr` ou `Retour arrière`.

Sauvegarder et quitter

- Pour sauvegarder le fichier : le raccourci clavier est `Ctrl + O`.
- Pour quitter Nano : le raccourci clavier est `Ctrl + X`.

Éditeurs de texte graphiques

Des éditeurs de texte plus puissants et plus conviviaux existent :

- Visual Studio Code (multi-plateforme)
- Atom (multi-plateforme)
- PyCharm (multi-plateforme)
- Gedit (Linux)

Introduction à Jupyter

Qu'est-ce qu'un Jupyter *notebook* ?

- un **cahier électronique** pouvant rassembler du texte, des images, des formules mathématiques, du code informatique, ...
- **manipulable** dans un **navigateur web**
- initialement développé pour Python, R et Julia (supporte actuellement ~40 langages)
- l'élément de base d'un *notebook* Jupyter est la **cellule** qui peut contenir :
  - du **texte** formatté en **Markdown**
  - du **code informatique exécutable**
- site officiel : **https://jupyter.org/**

Votre environnement de travail

- activation de l'environnement Conda pour la formation

```
$ conda activate formation-python
```

- exécution de JupyterLab

```
$ jupyter lab
```

JupyterLab est un environnement complet d'analyse.

Démonstration et premiers pas tous ensemble

Mon premier Jupyter *notebook*

- **Énoncé** : créer un *notebook* Jupyter nommé « `hello.ipynb` » permettant d'afficher les informations demandées en 5 minutes d'autonomie :

```
Je m'appelle Prénom Nom.
```

Fonctions

Fonctions

- Une fonction permet de répéter une instruction plusieurs fois sans avoir à réécrire la totalité du code.
- Syntaxe

```
def fonction(param1,param2):  # attention à l'indentation
    instruction1
    instruction2
    return resultat
```

Fonctions

- Exemple

```
>>> def pluriel(param):
...     param = param + "s"
...     return param
...
>>> mots=pluriel("sequence")
>>> mots
'sequences'
```

Passage d'arguments

- Vous n'êtes pas tenu de préciser le type d'arguments (entiers, réels, chaîne de caractère,...) lors de l'appel de la fonction.

```
>>> def fois(x, y):
...     return x*y
...
>>> fois(2, 3)
6
>>> fois(3.1415, 5.23)
16.430045000000003
>>> fois(2, 5.23)
10.46
>>> fois('to', 2)
'toto'
```

- L'opérateur « * » reconnait plusieurs types, la fonction 'fois' peut donc effectuer des tâches différentes.

Passage d'arguments

- Les fonctions sont capables de renvoyer plusieurs valeurs à la fois.
- On peut dès lors effectuer des affectations multiples.

```
>>> def carre_cube(x):
...     return (x**2, x**3)  # renvoi d'un tuple
...
>>> carre_cube(2)
(4, 8)
>>> z1, z2=carre_cube(2)  # affectation multiple
>>> z1
4
>>> z2
8
```

Passage d'arguments

- Renvoi de plusieurs valeurs à la fois avec une liste

```
>>> def carre_cube2(x):
...     return [x**2,x**3]
...
>>> carre_cube2(3)
[9, 27]
>>> z1,z2 = carre_cube2(3)
>>> z1
9
>>> z2
27
```

Passage d'arguments

- Les fonctions peuvent avoir un nombre d'arguments variables.

```
>>> str = "Python débutant: 1er mars, Python avancé: 1er avril"
>>> str.replace("Python", "PERL")  # remplace toutes les occurences
'PERL débutant: 1er mars, PERL avancé: 1er avril'
```

```
>>> str = "Python débutant: 1er mars, Python avancé: 1er avril"
>>> str.replace("Python", "PERL", 1)  # remplace une seule occurence
'PERL débutant: 1er mars, Python avancé: 1er avril'
```

- Pour avoir un nombre d'arguments variables, il faut définir des arguments facultatifs.

```
>>> def fct(x, y, z=1):  # un argument facultatif a une valeur par défaut
...     if z != 1:
...        instruction
...     return resultat
```

Portée des variables

L'endroit où est définit une variable détermine l'endroit où elle est accessible !

Allez sur **Python Tutor** (http://pythontutor.com/visualize.html). Écrivez d'abord une fonction carre(x) qui renvoie x au carré. Écrivez en dessous un programme principal qui appelle carre() avec la valeur 2 en paramètre. Suivez l'éxécution pour distinguer les valeurs des variables selon le contexte.

```
>>> # Fonction
>>> def carre(x):
...     z = 10
...     print("Valeur de z dans la fonction :", z)
...     return x**2

>>> # Programme principal
>>> z = 5

>>> print("Valeur de z dans le programme principal :", z)
>>> resultat = carre(2)
>>> print("Valeur de z dans le programme principal :", z)
>>> print(resultat)
```

Travaux pratiques

Créez un programme Python nommé « `revcomp.py` » ou un notebook Jupyter « `revcomp.ipynb` » contenant deux fonctions permettant de "reverse complémenter" une séquence en 25 minutes d'autonomie :

- création d'une **fonction** avec **un paramètre** pour **inverser** la **séquence**,
- création d'une **fonction** avec **un paramètre** pour **complémenter** la **séquence**,
- **retour** du **résultat** et **affichage** de celui-ci.

Travaux pratiques

Créez un programme Python « `traduction.py` » ou un notebook Jupyter « `traduction.ipynb` » contenant une fonction permettant de traduire une séquence en 25 minutes d'autonomie :

- création d'une **fonction** avec **un paramètre** pour **traduire** la **séquence**,
- **retour** du **résultat** et **affichage** de celui-ci.

Code génétique

```
code_genetique = {
"GCT":"A","GCC":"A","GCA":"A","GCG":"A",
"TTA":"L","TTG":"L","CTT":"L","CTC":"L",
"CTA":"L","CTG":"L","CGT":"R","CGC":"R",
"CGA":"R","CGG":"R","AGA":"R","AGG":"R",
"AAA":"K","AAG":"K","AAT":"N","AAC":"N",
"ATG":"M","GAT":"D","GAC":"D","TTT":"F",
"TTC":"F","TGT":"C","TGC":"C","CCT":"P",
"CCC":"P","CCA":"P","CCG":"P","CAA":"Q",
"CAG":"Q","TCT":"S","TCC":"S","TCA":"S",
"TCG":"S","AGT":"S","AGC":"S","GAA":"E",
"GAG":"E","ACT":"T","ACC":"T","ACA":"T",
"ACG":"T","GGT":"G","GGC":"G","GGA":"G",
"GGG":"G","TGG":"W","CAT":"H","CAC":"H",
"TAT":"Y","TAC":"Y","ATT":"I","ATC":"I",
"ATA":"I","GTT":"V","GTC":"V","GTA":"V",
"GTG":"V","TAG":"*","TGA":"*","TAA":"*"}
```

Création de modules

Création de modules

- Ce qui a été vu : création de **Fonctions** et utilisation de **Modules**.
- On peut vouloir ré-utiliser une fonction dans un autre programme.
- La solution consiste à créer un module et y héberger la fonction.
- On groupe les fonctions autour d'un thème précis dans un même module.

Création de modules

Pour créer un module il suffit de :

- créer un fichier avec les fonctions
- enregistrer ce fichier avec l'extension « `.py` »

```
1 """Module inutile qui affiche des messages :-)."""
2
3  def bonjour(nom):
4    """Dit Bonjour."""
5    return "Bonjour " + nom
6
7  def ciao(nom):
8    """Dit Ciao."""
9    return "Ciao " + nom
10
11 def hello(nom):
12    """Dit Hello."""
13    return "Hello " + nom
14
15 DATE = 16092008
```

Remarque : notez les commentaires entre triple guillemets.

Création de modules

L'importation de son propre module se fait comme suit :

```
>>> # Ici on suppose que les fonctions sont dans "message.py"
>>> import message
>>> message.hello("Joe")
'Hello Joe'
>>> message.ciao("Bill")
'Ciao Bill'
>>> message.bonjour("Monsieur")
'Bonjour Monsieur'
>>> message.DATE
16092008
```

Création de modules

- Les `docstrings` sont les commentaires entre triple guillemets.
- Ces commentaires sont appellés avec la commandes `help()`.
- Ces commentaires sont donc la documentation du module.

```
>>> import message
>>> help(message)
Help on module message:

NAME
    message - Module inutile qui affiche des messages :-).

FUNCTIONS
    bonjour(nom)
        Dit Bonjour.

    ciao(nom)
        Dit Ciao.

    hello(nom)
        Dit Hello.

DATA
    DATE = 16092008

FILE
    /home/oinizan/FORMATION-PYTHON-2019/message.py
```

Travaux pratiques

Créer un **module** Python nommé « `sequence.py` » qui reprend les fonctions sur les séquences en *25 minutes d'autonomie* :

- Générer la séquence en majuscule.
- Générer la séquence en minuscule.
- Calculer la longeur de la séquence.
- Extraire le premier codon.
- Extraire le dernier codon.
- Générer la séquence *reverse*.
- Générer la séquence *complement*.

Pensez à tester votre module !

Travaux pratiques

```
Help on module sequence:

NAME
    sequence - Un module pour la manipulation de sequences

FUNCTIONS
    complement(seq)
        Retourne la sequence complémentée

    dernier_codon(seq)
        Retourne le dernier codon

    longueur(seq)
        Retourne la longueur de la séquence

    majuscule(seq)
        Retourne la séquence en majuscule

    minuscule(seq)
        Retourne la séquence en minuscule

    premier_codon(seq)
        Retourne le premier codon

    reverse(seq)
        Retourne la sequence reverse
...
```

Gestion des erreurs

Exceptions

En Python les **exceptions** sont un ensemble de mots-clés dédié à la **gestion des erreurs**. Un exception est un objet qui représente une erreur. On dit qu'une exception est **levée** lorsqu'une erreur survient.

```
>>> 10/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

=> Python a retourné une exception de type **`ZeroDivisionError`**.

D'autres types d'exceptions existent : `NameError`, `TypeError`, `ValueError`, ...

```
>>> x = a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Pour la liste des exceptions en Python, voir
https://docs.python.org/3/library/exceptions.html

Exceptions

Try ... Except

Les mots-clés **`try`** et **`except`** permettent « **d'attraper** » les exceptions. On peut ainsi programmer des comportements adaptés en cas d'erreur.

```
>>> x = 10
>>> y = 0
>>> try:
...     print (x/y)
... except ZeroDivisionError:
...     print ("ERREUR, division par zéro impossible")
>>> print ("Le programme continue ...")
ERREUR, division par zéro impossible
Le programme continue ...
```

Exceptions

Try ... Except

Plusieurs blocs **except** peuvent se suivre pour gérer plusieurs types d'exceptions.

```
>>> x = 10
>>> y = 0
>>> z = "a"
>>> try:
...     print (x/y)
...     print (x+z)
... except ZeroDivisionError:
...     print (f"ERREUR, division par zéro impossible : ({x}/{y})")
... except TypeError:
...     print (f"ERREUR, opération impossible : ({x}+{z})")
... except Exception:
...     print ("Une erreur est survenue")
>>> print ("Le programme continue ...")
ERREUR, division par zéro impossible : (10 / 0)
Le programme continue ...
```

=> Lorsqu'un type d'exception est attrapé, les autres blocs « **except** » ne sont pas évalués.

Exceptions

Try ... Except

```
>>> x = 10
>>> y = 2
>>> z = "a"
>>> try:
...     print (f"Résultat de la division : {x/y}")
...     print (f"Résultat de l'addition : {x+z}")
... except ZeroDivisionError:
...     print (f"ERREUR, division par zéro impossible : ({x}/{y})")
... except TypeError:
...     print (f"ERREUR, opération impossible : ({x}+{z})")
... except Exception:
...     print ("Une erreur est survenue")
>>> print ("Le programme continue ...")
Résultat de la division : 5.0
ERREUR, opération impossible : (10+a)
Le programme continue...
```

=> La ligne « `except Exception` » attrape toutes les exceptions non attrapées par les autres blocs. Cela permet de gérer des exceptions non prévues.

Exceptions

Try ... Except ... Else ... Finally

- Le bloc **`else`** est éxécuté si aucune exception est déclenchée.
- Le bloc **`finally`** est executé quoiqu'il arrive.

```
>>> x = 10
>>> y = 0
>>> try:
...     div = x/y
...
... except ZeroDivisionError:
...     print ("ERREUR, division par zéro impossible")
...
... except Exception:
...     print ("Une erreur est survenue")
...
... else:  # Ici la suite du programme si aucune erreur n'est rencontrée
...     print (f"Le résulat de la division est {div}")
...
... finally:  # Ici les opérations "obligatoires"
...     print ("Fin du programme")
ERREUR, division par zéro impossible
Fin du programme
```

Exceptions

Mots-clés `as` et `raise`

Le mot clé **`as`** permet de **stocker** une exception dans une variable.

```
>>> x = 10
>>> y = 0
>>> try:
...     div = x/y
...
... except ZeroDivisionError as erreur:
...     print (f"ERREUR : {erreur}")
ERREUR : division by zero
```

Exceptions

Mots-clés `as` et `raise`

Le mot clé **`raise`** permet de **déclencher** une exception.

```
>>> x = 10
>>> y = 0
>>> if y == 0:
...     raise ZeroDivisionError("La division par zéro n'est pas possible.")
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[26], line 2
      1 if y == 0:
----> 2     raise ZeroDivisionError("La division par zéro n'est pas possible.")

ZeroDivisionError: La division par zéro n'est pas possible.
```

Exceptions

Mots-clé `assert`

Le mot clé **`assert`** permet de **vérifier** une condition.

```
>>> x = 10
>>> y = 0
>>> try:
>>>     assert y != 0  # Déclence une exception "AssertionError" si faux
>>> except AssertionError:
>>>     print ("La division par zéro n'est pas possible.")
La division par zéro n'est pas possible.
```

Ci-dessous, un bloc d'instructions équivalent qui utilise le mot-clé **`if`** :

```
>>> x = 10
>>> y = 0
>>> if y == 0:
...     raise ZeroDivisionError("La division par zéro n'est pas possible.")
La division par zéro n'est pas possible.
```

=> L'un et l'autre sont valables et ils produisent le même résultat, mais l'utilisation de `assert` aide à clarifier le code.

Travaux Pratiques (1/2)

Écrivez un script Python **`capteur.py`** ou un notebook **`capteur.ipynb`** qui calcule la moyenne d'une série de températures. Le programme doit demander à l'utilisateur d'entrer les températures un par une, et afficher la moyenne des températures à chaque fois (*25 minutes d'autonomie*).

Instructions :

- Utilisez la fonction **`input()`** pour demander à l'utilisateur d'entrer les nombres.
- Utiliser une boucle **`while`** pour demander à l'utilisateur d'entrer des températures jusqu'à ce qu'il entre le mot clé « **STOP** ».
- Assurez-vous de gérer avec **`try / except`** les cas où l'utilisateur entre une valeur non numérique.

Conseils :

- Utilisez la méthode **`float()`** pour convertir les entrées de l'utilisateur en nombres décimaux afin de permettre les opérations avec des nombres à virgule.
- Utilisez la méthode **`round()`** pour arrondir le résultat de la moyenne à deux décimales.
- Vous pouvez stocker les températures dans une liste pour faciliter le calcul de la moyenne.

Bonus :

- Utilisez une fonction pour calculer la moyenne des températures.
- Utilisez une fonction pour vérifier si l'entrée de l'utilisateur est un nombre.
- Ne calculer la calculer la moyenne que sur les 5 dernières températures.

Travaux Pratiques (1/2)

Exemple d'affichage lors de l'exécution :

```
Capteur de température

Entrez une température : 10
Moynenne : 10.0

Entrez une température : 13
Moynenne : 11.5

Entrez une température : 11
Moynenne : 11.34

Entrez une température : STOP
Fin du programme
```

Travaux Pratiques (2/2)

Écrivez un script Python « `codon.py` » ou un notebook « `codon.ipynb` » qui prends en entrée une séquence d'ADN et retourne son dernier codon, ou déclenche une exception si la séquence n'est pas complète (*25 minutes d'autonomie*).

Instructions :

- Utilisez la méthode **`input()`** pour demander à l'utilisateur d'entrer la séquence d'ADN.
- Utiliser la méthode **`dernier_codon()`** du module **`sequence.py`**. Ne modifiez pas cette méthode, mais utilisez-la dans votre programme.
- La programme doit déclencher une exception lorsque la séquence n'est pas complète.

Conseils :

- Exécuter la méthode **`dernier_codon()`** dans un bloc `try / except` pour gérer les exceptions.

Travaux Pratiques (2/2)

Exemple d'affichage lors de l'exécution :

```
Dernier codon

Entrez une séquence d'ADN : ATGCGTACG
Le dernier codon est : ACG
```

```
Dernier codon

Entrez une séquence d'ADN : ATGCGTAC
ERREUR : La séquence n'est pas complète.
```

Expressions régulières

Définition

Les expressions régulières constituent un système très puissant et très rapide pour faire des **recherches dans des chaînes de caractères** (fonctionnalité rechercher/remplacer très poussée).

Le **module re** permet de manipuler les expressions regulières en Python.

```
>>> import re
```

Exemples d'utilisation :

- On va rechercher le motif **ATG** dans la chaine de caractères **ATGCAGTCGACTAGCTAG**.
- On va chercher un motif de 3 lettres commençant par A et se terminant par G dans la chaine de caractères **ATGCAGTCGACTAGCTAG**.

Motifs

- [a-z] et [A-Z] : tout caractère alphabétique en minuscule et en majuscule
- [a-zA-Z0-9] : tout caractère alphanumérique
- [aeiouy] : un caractère de type voyelle (1 seul caractère)
- . : n'importe quel caractère

- \n : retour chariot
- \t : tabulation
- \s : un espace, une tabulation, un saut de ligne ([\t\n])

- ^ : début de ligne
- $: fin de ligne

- \d : [0-9]
- \w : [0-9A-Za-z_]

Exemples :

- un motif qui commence par **P9e** ou par **M8a**
- un motif qui se termine par **u7B 0** ou par **p9A** 7

Opérateurs

- * : 0 à n fois le caractère précédent ou l'expression entre parenthèses précédente
- + : 1 à n fois le caractère précédent ou l'expression entre parenthèses précédente
- ? : 0 à 1 fois le caractère précédent ou l'expression entre parenthèses précédente

- {n} : n fois le caractère précédent ou l'expression entre parenthèses précédente
- {n,m} : n à m fois le caractère précédent ou l'expression entre parenthèses précédente

- (AT|CG) : les chaînes de caractères **AT** ou **CG**

Exemples :

- un motif qui couvre **AT3G15245** et **OS4G02786**
- un motif qui couvre **ATG** puis **TTG** séparés de 7 caractères

Caractères spéciaux et points d'attention

- ^ | ( ) [ ] { } \ / $ + * ? . - : caractères spéciaux
- \ : caractère d'échappement

- [^0-9] : tout sauf un caractère numérique
- ^[0-9] : commence par un caractère numérique

- () : référencer une partie du motif afin de le récupérer

Quelques outils en ligne

- https://regexone.com/
- https://regexr.com/
- https://extendsclass.com/regex-tester.html#python
- https://pythex.org/

Fonction search()

La fonction **search()** du module re permet de **chercher un motif** au sein d'une chaîne de caractères.

```
>>> import re
>>>
>>> sequence = 'ATGATATATATATA'
>>>
>>> if re.search('ATG', sequence):
...     print("Motif trouvé !")
...
Motif trouvé !
```

Fonction match()

La fonction **match()** du module re permet de **chercher un motif au début** d'une chaîne de caractères.

```
>>> import re
>>>
>>> sequence = 'ATGATATATATATA'
>>>
>>> if re.match('ATG', sequence):
...     print("Motif trouvé au début !")
...
'Motif trouvé au début !'
>>>
>>> if re.match('TAG', sequence):
...      print("Motif trouvé au début !")
...
>>>
```

Fonction compile()

La fonction **compile()** du module re permet de **compiler l'expression régulière** avant de l'utiliser. Ceci est pratique lorsque l'on teste la même expression régulière sur un grand nombre de chaînes de caractères.

```
>>> import re
>>> regex = re.compile('^ATG')
>>>
>>> sequence = 'ATGATATATATATA'
>>> if regex.search(sequence):
...     print("Motif trouvé au début !")
...
'Motif trouvé au début !'
>>>
>>> sequence2 = 'TGATAGCATCGATCGATGC'
>>> if regex.search(sequence2):
...     print("Motif trouvé au début !")
...
>>>
```

Les groupes

```
>>> import re
>>> regex = re.compile('([A-Z]{2})([0-9]{1,2})G([0-9]{5})')
>>> resultat = regex.search('AT5G25476')
>>> resultat.group(0)
'AT5G25476'
>>> resultat.group(1)
'AT'
>>> resultat.group(2)
'5'
>>> resultat.group(3)
'25476'
>>>
>>> resultat.start()
0
>>> resultat.end()
9
```

Remarques :

- .group(0) retourne toute la correspondance.
- .group(1) retourne le premier élément, .group(2) le 2ème et ainsi de suite.
- .start() et .end() retournent la position de début et de fin de la zone qui correspond à l'expression régulière.

Les groupes nommés

Afin de faciliter la récupération des groupes, il est possible de les nommer.

```
>>> import re
>>> regex = re.compile('(?P<sp>[A-Z]{2})(?P<chr>[0-9]{1,2})G(?P<pos>[0-9]{5})')
>>>
>>> resultat = regex.search('AT5G25476')
>>>
>>> resultat.group('sp')
'AT'
>>> resultat.group('chr')
'5'
>>> resultat.group('pos')
'25476'
```

Méthode .sub()

La méthode **.sub()** du module re permet de **substituer une expression par une autre** au sein d'une chaîne de caractères.

```
>>> import re
>>>
>>> seq = 'ATGGTAGATAG'
>>>
>>> seq = re.sub('AT', 'TOTO', seq)
>>> seq
'TOTOGGTAGTOTOAG'
```

Travaux Pratiques

Créer un programme Python nommé **regex.py** ou un *notebook* Jupyter nommé **regex.ipynb** permettant d'isoler les *features* présents dans un fichier au format GenBank afin de générer un fichier au format tabulé contenant les informations suivantes :

- le nom du *feature*
- la position de début
- la position de fin
- le sens du brin
- le type de *feature*
- la longueur du *feature*

Quelques modules de bioinformatique

Modules de bioinformatique

- Quelques modules utiles :
  - `biopython` : manipulation de séquences et requête de bases de données
  - `numpy` : opérations mathématiques sur des tableaux et des matrices
  - `pandas` : manipulation de tableaux
  - `matplotlib` : génération de graphiques
- Et d'autres qui ne seront pas abordés :
  - `argparse` : gestion des arguments
  - `csv` : lire et écrire des fichiers au format csv
  - `gffpandas` : lecture d'annotations GFF3
  - `networkx` : gérer des graphes
  - `random` : générer des nombres aléatoires
  - `scikit-learn` : faire de l'apprentissage automatique
  - `sqlite3` : gérer une base de données SQLite
  - `tensorflow`, `keras`, `pytorch` : constuire des réseaux de neurones
  - `...`

Installation d'un module

Trois situations possibles :

- le module est un module de base (argparse, csv) : rien à faire !
- le module peut être installé avec l'outil « `pip` » (https://pypi.org/)

```
pip install pandas
```

  - remarque : l'outil « `conda` » est un autre outil de plus en plus répandu pour gérer les modules. Lien vers Anaconda.

- le module est un fichier « fait maison » (ex. : votre module `sequence.py`)
  - positioner le ficher à la racine du programme

Dans tous les cas, pour utiliser le module :

```
# mon_script.py
import argparse
import pandas
import sequence
```

Numpy

Module Numpy

- L'objet principal de numpy : un tableau à (n) dimensions: **ndarray**.
- WNG ! Il y a une classe array.array en python, ne pas confondre.
- https://docs.scipy.org/doc/numpy/user/quickstart.html

Module Numpy

- Exemple :

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.size
15
```

Module Numpy : création de tableaux

- array prend une *liste* en argument :

```
>>> a = np.array([1,2,3,4]) # une dimension
>>> a
array([1, 2, 3, 4])
>>> b = np.array([(1.5,2,3),(4,5,6)]) # deux dimensions
>>> b
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

- Préciser le *type* des valeurs :

```
>>> b.dtype.name
'float64'
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

Module Numpy : création de tableaux

- Initialiser avec des valeurs :

```
>>> np.zeros((3,4)) # tuple pour initialiser
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> np.ones((3,4))
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
>>> np.empty((2,3)) # valeurs aléatoires
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

- Initialiser avec des ranges :

```
>>> np.arange( 10, 30, 5)
array([10, 15, 20, 25])
```

Module Numpy : afficher les tableaux

- numpy présente les tableaux comme suit :
    - la dernière dimension (axe) est affichée de gauche à droite.
    - l'avant dernière dimension est affichée de haut en bas.
    - les dimensions restantes sont également affichées de haut en bas séparées par une ligne blanche.

```
>>> c = np.arange(24).reshape(2,3,4)
>>> print (c) # fonction print
[[[ 0  1  2  3] # dernière dimension: 4 colonnes
 [ 4  5  6  7]
 [ 8  9 10 11]] # avant dernière dimension: 3 lignes

 [[12 13 14 15]
  [16 17 18 19]
 [20 21 22 23]]] # première dimension: 2 tableaux de 3 lignes et 4 colonnes
```

Module Numpy : Indexes

- L'indexation fonctionne comme pour les listes.
- Le premier index correspond à la première dimension, ...

```
>>> print (c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

```
>>> print (c[1]) # second tableau
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

```
>>> print (c[1][1]) # seconde ligne du second tableau
[16 17 18 19]
```

Module Numpy : opérations mathématiques

- Moyenne

```
>>> a = np.array([[1, 2, 3][4, 5, 6][7, 8, 9]])

# du tableau complet
>>> np.mean(a)
5.0

# par dimension
>>> np.mean (a, axis=1)
array([2., 5., 8.])
```

- Maximum

```
>>> np.max(a)
9
```

- Somme

```
>>> np.sum(a, axis=0)
array([12, 15, 18])
```

Travaux pratiques

Le tableau suivant représente 4 relevés de températures pour chaque jour de la semaine :

```
Lun 12 11 14 12
Mar 12 10 14 11
Mer 11 11 14 13
Jeu 18 23 23 17
Ven 17 22 21 17
Sam 16 20 22 16
Dim 18 25 22 17
```

Créez un programme python « `temp.py` » ou un notebook Jupyter « `temp.ipynb` » qui indique le jour le plus chaud de la semaine. Remarques :

- il est possible d'itérer sur un `np.array` comme sur une liste,
- `np.mean()` : méthode `mean()` de `numpy`,
- `max()` et `index()` de la classe `list` peuvent être utile…

Pandas

Module Pandas

- Pandas pour la manipulation de données de type :
    - tabulées (SQL, Excel, ...),
    - séries temporelles ,
    - matrices,
    - données soumises aux traitements statistiques ...
- À utiliser avec `numpy`.
    - https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html

Module Pandas : création d'objets

- Import de `numpy` & `pandas` :

```
>>> import numpy as np
>>> import pandas as pd
```

- Création d'une *série* :

```
>>> s = pd.Series([1, 3, 5, np.nan, 6, 8]) #  autorise les nan
>>> s
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

Module Pandas : création d'objets

- À l'aide d'un *DataFrame* convertion des données d'un dictionnaire en données tabulées :

```
>>> df2 = pd.DataFrame({'A': 1.,
...                     'B': pd.Timestamp('20130102'),
...                     'C': pd.Series(1, index=list(range(4)), dtype='float32'),
...                     'D': np.array([3] * 4, dtype='int32'),
...                     'E': pd.Categorical(["test", "train", "test", "train"]),
...                     'F': 'foo'})
>>> df2
     A          B    C  D      E    F
0  1.0 2013-01-02  1.0  3   test  foo
1  1.0 2013-01-02  1.0  3  train  foo
2  1.0 2013-01-02  1.0  3   test  foo
3  1.0 2013-01-02  1.0  3  train  foo
```

```
>>> pd.Series(1, index=list(range(4)), dtype='float32')
>>> # index permet de numéroter les lignes
0    1.0
1    1.0
2    1.0
3    1.0
dtype: float32
```

Module Pandas : création d'objets

- Les colonnes du *DataFrame* ont des types différents :

```
>>> df2.dtypes
A           float64
B    datetime64[ns]
C           float32
D             int32
E          category
F            object
dtype: object
```

Module Pandas : création d'objets

```
>>> dates = pd.date_range('20130101', periods=6)
>>> dates
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

```
df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list('ABCD'))
>>> df
                   A         B         C         D
2013-01-01  0.716143  0.812185  1.360765  0.479679
2013-01-02 -0.004160 -0.619901  0.730602 -1.537268
2013-01-03 -1.162811 -0.319938 -0.953694 -1.895890
2013-01-04  0.174067  0.556952 -0.219398  0.403335
2013-01-05  0.367846 -0.390192  0.250073  0.310983
2013-01-06  0.089971  0.766604 -0.185695  0.158733
```

Module Pandas : visualisation

- Haut du tableau

```
>>> df.head()
                   A         B         C         D
2013-01-01  0.716143  0.812185  1.360765  0.479679
2013-01-02 -0.004160 -0.619901  0.730602 -1.537268
2013-01-03 -1.162811 -0.319938 -0.953694 -1.895890
2013-01-04  0.174067  0.556952 -0.219398  0.403335
2013-01-05  0.367846 -0.390192  0.250073  0.310983
```

- Bas du tableau

```
>>> df.tail()
                   A         B         C         D
2013-01-02 -0.004160 -0.619901  0.730602 -1.537268
2013-01-03 -1.162811 -0.319938 -0.953694 -1.895890
2013-01-04  0.174067  0.556952 -0.219398  0.403335
2013-01-05  0.367846 -0.390192  0.250073  0.310983
2013-01-06  0.089971  0.766604 -0.185695  0.158733
```

Module Pandas : visualisation

- Quelques statistiques descriptives

```
>>> df.describe()
              A         B         C         D
count  6.000000  6.000000  6.000000  6.000000
mean   0.030176  0.134285  0.163775 -0.346738
std    0.637691  0.646253  0.809841  1.072465
min   -1.162811 -0.619901 -0.953694 -1.895890
25%    0.019373 -0.372628 -0.210972 -1.113268
50%    0.132019  0.118507  0.032189  0.234858
75%    0.319401  0.714191  0.610469  0.380247
max    0.716143  0.812185  1.360765  0.479679
```

Module Pandas : visualisation

- Transposition :

```
>>> df.T
   2013-01-01  2013-01-02  2013-01-03  2013-01-04  2013-01-05  2013-01-06
A    0.716143   -0.004160   -1.162811    0.174067    0.367846    0.089971
B    0.812185   -0.619901   -0.319938    0.556952   -0.390192    0.766604
C    1.360765    0.730602   -0.953694   -0.219398    0.250073   -0.185695
D    0.479679   -1.537268   -1.895890    0.403335    0.310983    0.158733
```

- Tri par colonne :

```
>>> df.sort_values('B')
2013-01-02 -0.004160 -0.619901  0.730602 -1.537268
2013-01-05  0.367846 -0.390192  0.250073  0.310983
2013-01-03 -1.162811 -0.319938 -0.953694 -1.895890
2013-01-04  0.174067  0.556952 -0.219398  0.403335
2013-01-06  0.089971  0.766604 -0.185695  0.158733
2013-01-01  0.716143  0.812185  1.360765  0.479679
```

Module Pandas : accès aux valeurs

```
>>> df
                   A         B         C         D
2013-01-01  0.716143  0.812185  1.360765  0.479679
2013-01-02 -0.004160 -0.619901  0.730602 -1.537268
2013-01-03 -1.162811 -0.319938 -0.953694 -1.895890
2013-01-04  0.174067  0.556952 -0.219398  0.403335
2013-01-05  0.367846 -0.390192  0.250073  0.310983
2013-01-06  0.089971  0.766604 -0.185695  0.158733
```

- d'une colonne (sous forme d'une *Series*) :

```
>>> df['A']
2013-01-01    0.716143
2013-01-02    -0.004160
2013-01-03    -1.162811
2013-01-04    0.174067
2013-01-05    0.367846
2013-01-06    0.089971
Freq: D, Name: A, dtype: float64
```

Module Pandas : accès aux valeurs

```
>>> df
                   A         B         C         D
2013-01-01  0.716143  0.812185  1.360765  0.479679
2013-01-02 -0.004160 -0.619901  0.730602 -1.537268
2013-01-03 -1.162811 -0.319938 -0.953694 -1.895890
2013-01-04  0.174067  0.556952 -0.219398  0.403335
2013-01-05  0.367846 -0.390192  0.250073  0.310983
2013-01-06  0.089971  0.766604 -0.185695  0.158733
```

- d'une ligne à partir de son nom (sous forme d'une *Series*) :

```
>>> df.loc['2013-01-01']
A    0.716143
B    0.812185
C    1.360765
D    0.479679
```

- d'une ligne à partir de son numéro (sous forme d'une *Series*) :

```
>>> df.iloc[0]
A    0.716143
B    0.812185
C    1.360765
D    0.479679
```

Module Pandas : accès aux valeurs

```
>>> df
                   A         B         C         D
2013-01-01  0.716143  0.812185  1.360765  0.479679
2013-01-02 -0.004160 -0.619901  0.730602 -1.537268
2013-01-03 -1.162811 -0.319938 -0.953694 -1.895890
2013-01-04  0.174067  0.556952 -0.219398  0.403335
2013-01-05  0.367846 -0.390192  0.250073  0.310983
2013-01-06  0.089971  0.766604 -0.185695  0.158733
```

- d'une valeur en particulier

```
>>> df['B']['2013-01-03']
-0.319938
```

```
>>> df.at['2013-01-03','B']
-0.319938
```

Modules Pandas : itérations

```
>>> df
                   A         B         C         D
2013-01-01  0.716143  0.812185  1.360765  0.479679
2013-01-02 -0.004160 -0.619901  0.730602 -1.537268
2013-01-03 -1.162811 -0.319938 -0.953694 -1.895890
```

- sur les colonnes :

```
>>> for x in df:
...     print(x)
...
A
B
C
D
```

- sur les lignes :

```
>>> for x in df.itertuples():
...     print(x.A)
...
0.716143
-0.004160
-1.162811
```

Module Pandas : ajout d'une colonne

```
>>> df
                   A         B         C         D
2013-01-01  0.716143  0.812185  1.360765  0.479679
2013-01-02 -0.004160 -0.619901  0.730602 -1.537268
2013-01-03 -1.162811 -0.319938 -0.953694 -1.895890
```

```
>>> df.assign(E=[-0.575142, 0.853726, -0.651291])
                   A         B         C         D         E
2013-01-01  0.716143  0.812185  1.360765  0.479679 -0.575142
2013-01-02 -0.004160 -0.619901  0.730602 -1.537268  0.853726
2013-01-03 -1.162811 -0.319938 -0.953694 -1.895890 -0.651291
```

Remarque : la colonne `E` avec la méthode `assign` n'est pas persistante.

```
>>> df.insert(4, 'E', [-0.575142, 0.853726, -0.651291])
                   A         B         C         D         E
2013-01-01  0.716143  0.812185  1.360765  0.479679 -0.575142
2013-01-02 -0.004160 -0.619901  0.730602 -1.537268  0.853726
2013-01-03 -1.162811 -0.319938 -0.953694 -1.895890 -0.651291
```

Remarque : le 1er argument d'`insert` est la localisation de la nouvelle colonne.

Module Pandas : suppression d'une colonne

```
>>> df
                   A         B         C         D         E
2013-01-01  0.716143  0.812185  1.360765  0.479679 -0.575142
2013-01-02 -0.004160 -0.619901  0.730602 -1.537268  0.853726
2013-01-03 -1.162811 -0.319938 -0.953694 -1.895890 -0.651291
```

```
>>> del df['E']
>>> df
                   A         B         C         D
2013-01-01  0.716143  0.812185  1.360765  0.479679
2013-01-02 -0.004160 -0.619901  0.730602 -1.537268
2013-01-03 -1.162811 -0.319938 -0.953694 -1.895890
```

Module Pandas : combinaison de *DataFrames*

Combinaison de deux tableaux de chiffres à partir d'une colonne commune.

```
>>> df1
            Sandra  Thomas
maths           19      15
francais        13      17
géographie      16      20

>>> df2
          Franck  Olivier
maths         14       12
francais      11       18
histoire      17       14
```

```
>>> pd.concat([df1, df2])
            Sandra  Thomas  Franck  Olivier
maths         19.0    15.0     NaN      NaN
francais      13.0    17.0     NaN      NaN
géographie    16.0    20.0     NaN      NaN
maths          NaN     NaN    14.0     12.0
francais       NaN     NaN    11.0     18.0
histoire       NaN     NaN    17.0     14.0
```

Remarques : `NaN` indique des valeurs manquantes ; lignes des deux *DataFrames* recopiées !

Module Pandas : combinaison de *DataFrames*

Ajout du paramètre `axis=1` pour ne pas recopier les lignes des deux *DataFrames*.

```
>>> pd.concat([df1, df2], axis=1)
            Sandra  Thomas  Franck  Olivier
maths         19.0    15.0    14.0     12.0
francais      13.0    17.0    11.0     18.0
géographie    16.0    20.0     NaN      NaN
histoire       NaN     NaN    17.0     14.0
```

Import / Export de fichiers CSV

- Importer des données depuis un fichier CSV

```
df = pandas.read_csv("nom_fichier.csv", options...)
```

- Écrire un tableau dans un fichier CSV

```
df.to_csv("nom_fichier.csv", options...)
```

Travaux pratiques

Dans un script python nommé « `dataframe.py` » ou un notebook Jupyter « `dataframe.ipynb` », aidez-vous de la documentation en ligne sur Pandas pour créer un dataframe comme suit :

```
    R1 R2  R3  R4
Lun  12  11  14  12
Mar  12  10  14  11
Mer  11  11  14  13
Jeu  18  23  23  17
Ven  17  22  21  17
Sam  16  20  22  16
Dim  18  25  22  17
```

Affichez les informations suivantes :

```
              Lun        Mar    Mer        Jeu        Ven   Sam        Dim
count    4.000000   4.000000   4.00   4.000000   4.000000   4.0   4.000000
mean    12.250000  11.750000  12.25  20.250000  19.250000  18.5  20.500000
std      1.258306   1.707825   1.50   3.201562   2.629956   3.0   3.696846
min     11.000000  10.000000  11.00  17.000000  17.000000  16.0  17.000000
25%     11.750000  10.750000  11.00  17.750000  17.000000  16.0  17.750000
50%     12.000000  11.500000  12.00  20.500000  19.000000  18.0  20.000000
75%     12.500000  12.500000  13.25  23.000000  21.250000  20.5  22.750000
max     14.000000  14.000000  14.00  23.000000  22.000000  22.0  25.000000
```

Module Biopython

Qu'est-ce que Biopython ?

Biopython est un projet gratuit et *Open Source* fournissant des fonctions développées pour le traitement et l'analyse de données biologiques en Python.

Page officielle : https://biopython.org/.

Tutoriel : http://biopython.org/DIST/docs/tutorial/Tutorial.html.

Utilisation

```
>>> import Bio
```

Aperçu des fonctionnalités

- manipulation de séquences (FASTA, GenBank…)
- alignement de séquences (ClustalW, MUSCLE, BLAST…)
- accès aux données du NCBI (Entrez, ESearch, EFetch…)
- Swiss-Prot et ExPASy
- PDB
- génétique des populations
- phylogénie
- analyse de motifs / de clusters
- …

Manipulation de séquences

Création d'une séquence

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq
Seq('AGTACACTGGT')
>>> print(my_seq)
AGTACACTGGT
```

Séquence complémentaire

```
>>> my_seq.complement()
Seq('TCATGTGACCA')
```

Séquence complémentaire inverse

```
>>> my_seq.reverse_complement()
Seq('ACCAGTGTACT')
```

Travaux Pratiques

Créer un programme Python nommé **biopython_revcomp.py** ou un *notebook* Jupyter nommé **biopython_revcomp.ipynb** permettant de répondre à l'exercice sur les fonctions de *reverse complement* mais en utilisant cette fois-ci Biopython.

Parsing de données

Fichier *ls_orchid.fasta* contenant 94 séquences

```
>gi|2765658|emb|Z78533.1|CIZ78533 C.irapeanum 5.8S rRNA gene and ITS1 and ITS2 DNA
CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGGAATAAACGATCGAGTG
AATCCGGAGGACCGGTGTACTCAGCTCACCGGGGGCATTGCTCCCGTGGTGACCCTGATTTGTTGTTGGG
...
```

```
>>> from Bio import SeqIO
>>> for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta"):
...     print(seq_record.id)
...     print(repr(seq_record.seq))
...     print(len(seq_record))
...
```

Sortie standard

```
gi|2765658|emb|Z78533.1|CIZ78533
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', SingleLetterAlphabet())
740
...
gi|2765564|emb|Z78439.1|PBZ78439
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC', SingleLetterAlphabet())
592
```

Travaux Pratiques

Créer un programme Python nommé **biopython_gb.py** ou un *notebook* Jupyter nommé **biopython_gb.ipynb** permettant, à partir du fichier au format GenBank *ls_orchid.gbk*, d'afficher les informations suivantes :

- l'identifiant de la séquence
- la séquence
- la longueur de la séquence

Module matplotlib

Module matplotlib

Le module **matplotlib** permet de générer des graphiques.

Différents types de représentation sont possibles :

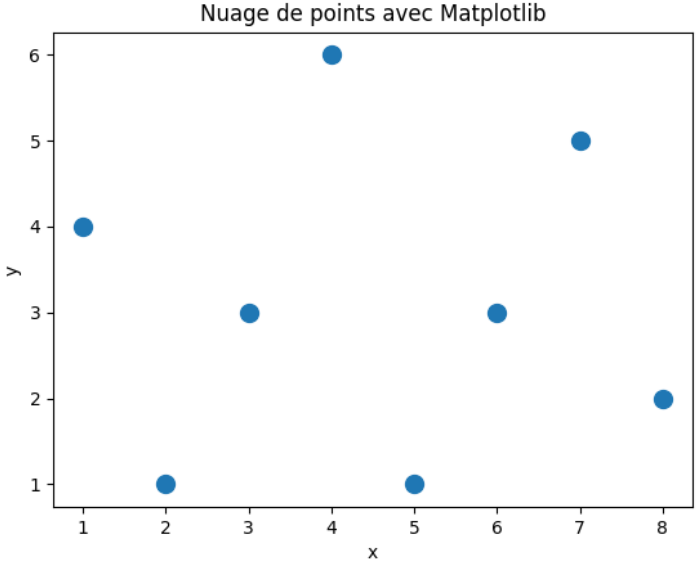
- sous forme de **points**
- sous forme de **courbe**
- sous forme d'**histogrammes**

Site Web dédié au module : https://matplotlib.org/.

Site intéressant : https://www.python-graph-gallery.com/.

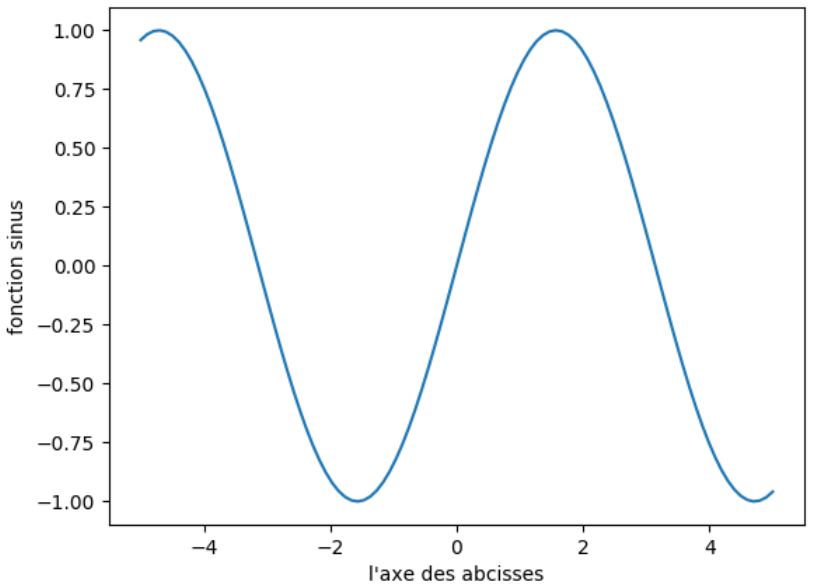
Nuage de points

```
>>> import matplotlib.pyplot as plt
>>> x = [1,2,3,4,5,6,7,8]
>>> y = [4,1,3,6,1,3,5,2]
>>> plt.scatter(x,y,s=100)
>>> plt.title('Nuage de points avec Matplotlib')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.savefig('ScatterPlot.png')
>>> plt.show()
```



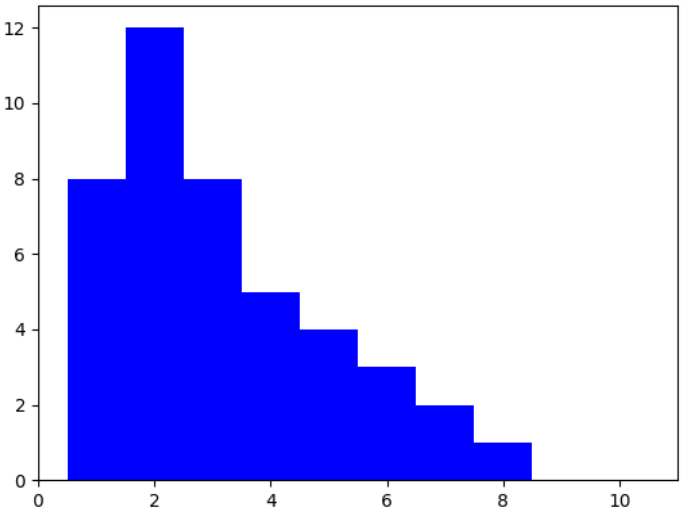
Courbe

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> x=np.linspace(-5,5,100)
>>> plt.plot(x,np.sin(x))  # on utilise la fonction sinus de Numpy
>>> plt.ylabel('fonction sinus')
>>> plt.xlabel("l'axe des abcisses")
>>> plt.show()
```



Histogrammes

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> x = [1,2,3,4,5,6,7,8,9,10]
>>> height = [8,12,8,5,4,3,2,1,0,0]
>>> width = 1.0
>>> plt.bar(x, height, width, color='b' )
>>> plt.savefig('SimpleBar.png')
>>> plt.show()
```



Travaux Pratiques

Dans un notebook Jupyter nommé « `graph.ipynb` », générer un **graphique** qui représente l'évolution des températures prévues pour la semaine à venir, et donner un **titre** au graphique et des **labels** aux axes des abscisses et des ordonnées.

Les prévisions pour les 7 jours à venir à Jouy-en-Josas sont les suivantes : `30, 34, 35, 28, 25, 26, 24`

Exercice complet

Enoncé

Objectif : à partir des chromosomes au format FASTA et de la description des annotations au format GFF, récupérer les séquences nucléotidiques des CDS (**S_cerevisiae_chromosomes.fna** / **S_cerevisiae_annotations.gff**)

Conseils :

- Bien regarder la structure des fichiers
- Procéder étape par étape

Bonus :

- Formater la sortie au format FASTA
- Proposer un histogramme horizontal avec en abscisse la taille des séquences du fichier FASTA et en ordonnée leurs noms
- Proposer un *pie chart* composé du nombre des différents types d'éléments présents dans le fichier GFF
- Développer un *script* prenant en argument les deux fichiers (FASTA et GFF)

Données

Extrait du fichier GFF

```
BK006935.2    tpg    CDS    65778    67520    .    -    0
ID=cds28;Parent=rna28;Dbxref=SGD:S000000038,NCBI_GP:DAA06946.1;Name=DAA06946.1;
Note=G1 cyclin involved in cell cycle progression%3B activates Cdc28p kinase to
promote the G1 to S phase transition%3B plays a role in regulating transcription
of the other G1 cyclins%2C CLN1 and CLN2%3B regulated by phosphorylation and
proteolysis%3B acetly-CoA induces CLN3 transcription in response to nutrient
repletion to promote cell-cycle entry.;gbkey=CDS;gene=CLN3;product=cyclin CLN3;
protein_id=DAA06946.1
```

Extrait du fichier multiFASTA

```
>BK006935.2 TPA_inf: Saccharomyces cerevisiae S288c chromosome I, complete sequenc
ccacaccacacccacacacccacacaccacaccacacaccacaccacacccacacacacacatCCTAACACTACCCTAACAC
CTAACCCTGGCCAACCTGTCTCTCAACTTACCCTCCATTACCCTGCCTCCACTCGTTACCCTGTCCCATTCAACCATACCA
CTCCGAACCACCATCCATCCCTCTACTTACTACCACTCACCCACCGTTACCCTCCAATTACCCATATCCAACCCACTGCCA
CTTACCCTACCATTACCCTACCATCCACCATGACCTACTCACCATACTGTTCTTCTACCCACCATATTGAAACGCTAACAA
ATGATCGTAAATAACACACACGTGCTTACCCTACCACTTTATACCACCACCACATGCCATACTCACCCTCACTTGTATACT
GATTTTACGTACGCACACGGATGCTACAGTATATACCATCTCAAACTTACCCTACTCTCAGATTCCACTTCACTCCATGG
CCCATCTCTCACTGAATCAGTACCAAATGCACTCACATCATTATGCACGGCACTTGCCTCAGCGG
```

Une ou deux pistes

- Comment peut-on croiser les données issues des deux fichiers ?
- Quels sont les champs d'intérêt ?
- De quelles informations a-t-on besoin pour générer la sortie ?

Différentes étapes

- lien entre les deux fichiers via la séquence de référence
- filtrage sur le type d'élément (CDS)
- informations nécessaires : positions et orientation
- sortie : un nom de séquence + séquence