**Liste des commandes utiles en Python**

*Le document listes des fonctions utiles dans le cadre de l’UE 4B033, Python for physiology modelling (pypm). Les commandes sont valables pour la version 3.7 de Python.*

Sommaire

[1. Les bases 2](#_Toc3221947)

[1.1. Fonctions de base 2](#_Toc3221948)

[1.2. Les listes 3](#_Toc3221949)

[1.3. Les tuples 4](#_Toc3221950)

[1.4. Les dictionnaires 4](#_Toc3221951)

[1.5. Les conditions 5](#_Toc3221952)

[1.6. Les fonctions 5](#_Toc3221953)

[1.7. Les fichiers 5](#_Toc3221954)

[1.8. Les classes 6](#_Toc3221955)

[1.9. Les Exceptions 6](#_Toc3221956)

[1.10. Objet muable et immuables 6](#_Toc3221957)

[2. Les modules 7](#_Toc3221958)

[1.11. Random 7](#_Toc3221959)

[1.12. Numpy 7](#_Toc3221960)

[1.13. Matplotlib 7](#_Toc3221961)

[1.14. Math 8](#_Toc3221962)

[1.15. Scipy 8](#_Toc3221963)

[1.16. Itertools 8](#_Toc3221964)

[1.17. BioPython 8](#_Toc3221965)

[1.18. pip 8](#_Toc3221966)

[1.19. os 8](#_Toc3221967)

[3. Truc et astuces 9](#_Toc3221968)

# Les bases

## Fonctions de base

* float() Force à avoir un chiffre à virgule
* int() Force à avoir un chiffre entier
* str() Indique qu'il s'agit d'une liste de caractères
* list()
* type() Type de la variable
* dir(" ")
* help(module.fonction) Donne l’aide d’une fonction
* id(variable) Donne l’id d’une variable. Permet de voir si deux objets ont le même id (donc si on modifie l’un, l’autre est aussi modifié). Permet aussi de voir si notre variable a changé d’id après modification.
* print() Affiche à l'écran une ou plusieurs variable
  + .format()
  + ('\n') Permet un retour à la ligne
* range()
* len(variable) Renvoie la taille de la variable
* imput()
* max(variable) Renvoie la valeur maximale d’une liste ou d’un tuple
* min(variable) Renvoie la valeur minimale d’une liste ou d’un tuple
* map(fonction, variable)
* reversed()
* zip()
* enumerate()
* str.upper() Passe toutes les données en majuscules
* str.count('A')
* str.replace('caractère à remplacer', 'par ce caractère') Permet de remplacer un ou plusieurs caractère par un ou plusieurs autres mais il faut réattribuer la valeur sinon la modification s’affiche mais la variable n’est pas modifier
  + Exemple : phrase = 'J’aime le chocolat'
  + phrase = phrase.replace('chocolat, 'beure')
* str.upper()
* str.translate()
* .search()
* .create()
* .strip()
* .rstrip()
* .split()
* "".join()
* str1.find(str2, beg, end) Renvoie l’index de l’endroit où les chaines de caractères sont similaires. S’il y a plusieurs endroit où c’est similaire, find() ne revoit que la premier index (boucle avec beg variable et une liste qui récupère les index si on veut tous les récupérer). Renvoie -1 si la valeur n’est pas trouvée. str2 est la chaine de caractère recherché dans str1. Par défaut, beg est à 0 et end est à len(str1) (remplacé par beg et end par des int).
  + str1.index(str2, beg, end =) est similaire à find() mais renvoie une erreur si la valeur de str2 n’est pas dans str1
* .endswith()

## Les listes

Les index d’une liste : liste = ['p', 'y', 'p', 'm'] les index sont 0 1 2 3 ou -4 -3 -2 -1

liste[0:2] => ['p', 'y']

liste[:2] => ['p', 'y'] Va du début à la position 2 (exclu)

liste[2:] => ['p', 'm'] Va de la position 2 à la fin

liste[:-2] => ['p', 'y'] Va du début à la position -2 (exclu)

On peut trouver un élément dans une liste avec if, si l’élément est dans la liste (si c’est vrai) alors on effectue la suite sinon on saute le if.

* if 'p' in liste:

On peut boucler/itérer dans une liste.

* for lettre in liste:

On peut appliquer des opérations mathématiques à une liste (ou d’autres modification type .upper()) boucler/itérer dans une liste :

* liste = [1, 2, 3, 4]
* liste2 = [i\*2 for i in liste]
* .append() Rajoute à la fin de la liste la ou les nouvelles valeurs
  + liste + 'valeur' fonctionne aussi mais la liste n’est pas mise à jour à moins de faire : liste += 'valeur'
* .insert(1, 'valeur') Rajoute à un index précis de la liste, la ou les nouvelles valeurs. Cela décale les autres valeurs à index +1 (ou +n valeurs ajoutés).
* .extend() Ajoute une liste à une autre liste (fusion de liste). Si on passait par append, on aurait une liste dans une liste (= liste imbriqué).
* .remove('valeur') Retire une valeur de la liste en lui donnant la valeur à retirer.
* .pop(index) Retire une valeur de la liste en lui donnant l’index de la valeur et renvoie la valeur supprimé
  + liste\_2 = liste.pop(1) La valeur qui à l’index 1 est supprimé de la liste mais on peut attribuer cette valeur supprimé à une autre variable
* del liste[index] Supprime une valeur de la liste en lui donnant l’index de la valeur
* .sort() Range la liste dans l’ordre croissant, si c’est une liste est composé d’int ou de float
  + liste.sort(reverse = True) Range la liste mais dans le sens inverse (décroissant), c’est l’équivalent de .reverse()
* sorted() Si une liste est composé d’int ou de float, affiche la liste dans l’ordre croissant. La liste n’est pas modifié, sauf si on écrit comme cela :
  + liste = sorted(liste)
* .index('valeur') Renvoie l’index de la valeur recherché
* .reverse() Inverse la liste, ne fonctionne pas avec un string

## Les tuples

Les tuples sont des listes que l’on ne peut pas modifier. Ils se présentent sous format.

mon\_tuple = (1, 2, 3) Attention le mot tuple est déjà utilisé, on ne peut donc pas l’utiliser comme nom de variable

Si on veut vraiment la modifier, il faut enregistré ce tuple dans une liste. Ou alors il faut redéclarer / écraser le tuple en le redéclarant.

mon\_tuple[index] Permet de récupérer la valeur lié à cet index

Fonctionne globalement comme une liste, la majorité des fonctions des listes marchent pour les tuples.

## Les dictionnaires

On a des pairs de clés et de valeurs sous le format :

dictionnaire = {'clé':'valeur', 'clé1':'valeur1', 'clé2':'valeur2'}

On peut impriquer des dictionnaires les uns dans les autres :

dictionnaire = {

'Pierre': {'age': 40, 'profession': 'banquier'},

'Paul': {'age': 25, 'profession': 'ingénieur'}

}

* .get(clé) Renvoie la valeur si on donne la clé
  + dictionnaire[clé] une autre méthode
  + dictionnaire['Pierre'][ 'age'] Renvoie la valeur quand un dictionnaire est inclus dans un autre
* .keys() Renvoie une liste de toutes les clés du dictionnaire
* .values() Renvoie une liste de toutes les valeurs du dictionnaire
* .items()  Renvoie une liste de tuples => [(clé 1, valeur 1), (clé 2, valeur 2)]
  + On peut récupérer les clés et les valeurs avec 2 boucles for
    - base = 'C:/Users/Paul/Desktop'
    - dossiers = {'Musique': ['Rock', 'Jazz'], 'Photos': ['Vacances', 'Famille']}
    - for cle, valeurs in dossiers.items():
      * for valeur in valeurs:
        + dossier = '{0}/{1}/{2}'.format(base, cle, valeur)
        + os.makedirs(dossier) Créer un nouveau dossier dont on aura spécifier le chemin et le nom du dossier à la fin (ici c’est la cle et la valeur font office de dossiers à créer).
* .update({'clé':'valeur'}) Si on veut modifier une valeur associé à une clé
  + dictionnaire['clé'] = valeur une autre méthode
* del dictionnaire['clé'] Supprime le couple clé : valeur du dictionnaire
* .pop('clé') Supprime le couple clé : valeur du dictionnaire et renvoie la valeur supprimé
* .clear() Efface tout le contenu du dictionnaires
  + dictionnaire = {} Méthode équivalente
* if dictionnaire.has\_key('clé') : Permet de vérifier si une clé est contenue dans le dictionnaire
  + if 'clé' in dictionnaire : Méthode équivalente

## Les conditions

Boucle For

Exemple de raccourcie :

print('A' + ' '.join(str(e) for e in profile[0]))

break

continue

Boucle While

If / Else / Elif

Les comparateurs : != / == / < / > / =< / =>

## Les fonctions

## Les fichiers

* f = open(''Chemin'', ''r'') Ouvre un fichier suivant une façon précise (lecture, écriture, etc.) et on attribue une variable à ce fichier. Sachant que le chemin vers le fichier, on peut aussi le mettre dans une variable pour éviter de taper à chaque fois tout le chemin (dans ce cas, plus besoin des '' '').
  + ''r'' Ouvre un fichier pour lecture
  + ''w'' Ouvre un fichier pour écrire dedans, il écrase tout ce qu’il y avait avant. Créer un nouveau fichier s’il n’existe pas.
  + ''a'' Ouvre un fichier pour écrire dedans en rajoutant à la suite de ce qu’il y a déjà. N’écrase pas contraire à ''w''. Créer un nouveau fichier s’il n’existe pas. (a pour append).
* if f.mode == ''r'' Applique une condition suivant le type d’ouverture du fichier
* .read()  Cette méthode renvoie tout ce qui contenait le fichier dans une chaine de caractère.. S’il y avait plusieurs lignes, la chaine de caractère contient des \n.
* .readline() Cette méthode lit une seule ligne du fichier. Néanmoins il y a un curseur « invisible » donc si on relance cette fonction, on a la ligne d’après.
  + while line : line est une chaine de caractère prédéfinit avant et qui n’est pas vide. Ce qui se passe ici c’est qu’il y a une boucle tant que line est vrai, c’est-à-dire tant qu’elle n’est pas vide (false).
* .readlines() Cette méthode lit ligne par ligne le fichier. On a une liste où chaque élément est une ligne du fichier sous forme de chaine de caractères.
  + for line in f.readlines() : Si on veut récupérer ligne par ligne des éléments.
* .write() Ajoute des informations dans un fichier.
* .writelines ()
* .close() Tant que l’on ne ferme pas le fichier, il reste ouvert. Ainsi si on a écrit dedans avec .write, tant que l’on ne ferme pas le fichier, la modification n’est pas sauvegardé.
* .closed() Une variable de la variable qui contient le fichier. Permet de savoir si le fichier est ouvert ou non. False si le fichier est ouvert, True s’il est fermé.

## Les classes

Permet de réutiliser du code, un peu comme une function. C’est un moyen de définir des attributs, des méthodes, etc. En fait, c’est surtout un conteneur qui permet d’avoir des propriétés que l’on va pouvoir instancier pour créer des objets. Une classe peut contenir des fonctions, que l’on appelle des méthodes.

Sa structure de base est :

* class NomDeLaClasse():
  + pass

Principe du constructeur, définit par :

* + def \_\_init\_\_(self): self est un argument de la fonction. C’est une variable qui réfère à la class elle-même
    - print('Construction de la classe')

## Les Exceptions

Permet de prévoir des cas si jamais notre script renvoie une erreur pour un bout de code.

* try: Exécute se code, s’il y a une erreur c’est le except qui sera renvoyé
* except:

## Objet muable et immuables

Muable : les listes, les dictionnaires et les sets / Immuable : str, int, float, bouléen et les tuples

C’est très important de connaitre cette nuance car il va influer sur le temps d’exécution d’un script. Imaginons, on fait une boucle qui ajoute lettre à un chaine de caractère. La chaine de caractère doit être recrée à chaque fois. Par contre, si on utilise une liste où l’on ajoute les lettres et à la fin d’utiliser la fonction .join() pour passer de la liste à une chaine de caractère. Le temps de calcul peut être divisé par 55 !

# Les modules

Il y a plusieurs façons d’importer des fonctions venant d’autres script python (= modules)

* import x Importe tout le module, les fonctions inclus dans le modules devront du coup être appelées comme cela : module.fonction() donc ici x.fonction()
* from x import fonction1 Cela permet d’importer qu’une seule fonction d’un module. Le problème c’est que si on a déjà créé une fonction1 dans notre script, le script plante car il ne sait plus à quelle fonction se référer (celle du script ou celle du module).
  + On peut aussi faire from x import \* cela importe toutes les fonctions du module. Il est déconseillé de faire cela car si on a une variable qui a le même nom qu’une fonction importé, le script plante. Néanmoins, on peut utiliser directement la fonction, on a plus besoin de spécifié le module en l’utilisant.
* import module as x Permet de modifier le nom du module quand on importe une fonction venant de celui-ci. module.fonction() devient x.fonction()
* module.\_\_file\_\_ Renvoie le chemin où ce situe le module
* dir(module) Donne la liste de toutes les fonctions du module
* help(module.fonction) Donne l’aide d’une fonction

## Random

* import random
* random.randint()

## Numpy

Numpy permet l’utilisation de matrice à n dimensions.

* import numpy as np
* np.linspace()
* np.zeros()
* np.asarray()
* np.empty()
* np.cumsum()
* np.expand\_dims()
* np.random.rand()

## Matplotlib

* import matplotlib as plt
* plt.plot()
* plt.show()

## Math

* import math
* math.sqrt()
* math.factorial()

## Scipy

* import scipy
* scipy.stats.rvs()

## Itertools

* import itertools
* itertools.permutations()

## BioPython

* import Bio
* Bio.SeqIO.parse()
* Bio.SeqUtils.GC() Compte le % de GC dans une séquence
* Bio.Seq()
  + Bio.Alphabet.IUPAC.unambiguous\_rna
  + Bio.Alphabet.IUPAC.unambiguous\_dna
* Bio.SeqIO.parse()

## pip

## os

Permet de créer des dossiers, des fichiers, vérifier s’ils existent, etc.

* import os
* os.makedirs('Chemin') Créer un nouveau dossier dont on aura spécifier le chemin et le nom du dossier à la fin => 'C:\Users\Paul\Desktop\miam' Créer un dossier miam. Si le dossier parent (ici Desktop) n’existe pas, il est aussi créé.

# Truc et astuces

if \_\_name\_\_ == '\_\_main\_\_':

print(x)

Si ce code est exécuté en tant que script principal (appelé directement avec Python et pas importé comme un module), alors la partie en dessous est exécuter (ici la fonction print() ).

b = b + a ou b += a

a = b

Cette attribution est fausse car a et b auront le même id ce qui fait que si on change après la valeur de l’un, la valeur de l’autre changera aussi. Si on veut conserver une valeur dans une autre variable pour ensuite pour changer la première variable il faut faite comme cela :

c = b + a

a = b

b = c

ou alors : a, b = b, a + b

Attention au chemin dans windows. Si jamais notre chemin contient \n par exemple Python l’interprète comme un saut de ligne et non comme faisait partie du chemin. On peut soit remplacer les \ par des \\ ou par des /.

* base = r'C:\Users\Paul\Desktop' Ici r va remplacer le \ par des \\
* base = base.replace('\\', '/')