

TP1 – Bases de Python 1

1 Pour commencer

Python est un langage de programmation interprété haut niveau. Une implémentation Python peut-être téléchargée, par exemple avec Miniconda3 (<https://conda.io/miniconda.html>). Une fois Python installé, vous pouvez installer un IDE (un environnement de développement) pour travailler avec une interface graphique. On pourra par exemple penser à Pyzo ou Spyder, ou encore Jupyter pour une version notebook. À l'agrégation, les interpréteurs disponibles sont Jupyter (accessible en tapant `jupyter notebook` dans un terminal) et Pyzo (<https://pyzo.org/start.html>).

Se débrouiller avec Pyzo - L'aide du logiciel :

Python et ses packages constituent un langage extrêmement riche et il est hors de question d'en connaître toutes les commandes qui seront abordées en TP et leurs syntaxes. L'aide de Pyzo est très utile pour retrouver ce genre d'informations, et il est essentiel de savoir l'utiliser :

- La première fois, dans l'onglet "Outils" sélectionnez l'aide interactive.
- Lorsque vous tapez une commande dans l'interpréteur Python (fenêtre en haut à droite), l'aide s'affichera automatiquement.
- Vous pouvez aussi entrer directement la commande dans la fenêtre d'aide.
- En cas de doute sur l'orthographe de la fonction, la touche tabulation <Tab> à la suite d'un début de mot montre quelles sont les commandes commençant par ces lettres.

À noter que l'aide en ligne de Pyzo ne sera pas disponible le jour de l'agrégation. En revanche, en plus de l'aide de Pyzo, le tutoriel officiel de Python sera disponible le jour de l'oral de l'agrégation. Ce tutoriel est disponible à l'adresse suivante : <https://docs.python.org/fr/3.8/tutorial/>.

Présentation de l'interface :

L'interface de Pyzo est divisé en plusieurs fenêtres :

- Un éditeur de texte pour taper son script Python directement dans un fichier et le sauvegarder.
- Un interpréteur Python pour taper et exécuter directement des commandes Python (la fenêtre en haut à droite avec le "prompt" `>>>`).
- Un navigateur pour se déplacer dans l'arborescence des dossiers.
- Bien d'autres fenêtres que l'on affiche ou pas en utilisant le menu "Outils".

Remarques générales :

- Les commandes peuvent se taper directement après le prompt (`>>>`) de Python : l'opération est alors immédiatement effectuée et le résultat retourné.
- On ne peut pas modifier une commande entrée précédemment. On peut ré-exécuter une commande se trouvant dans la fenêtre "Historique des commandes" en double-cliquant dessus, ou en faisant défiler les commandes entrées précédemment avec la flèche vers le haut du clavier.
- On peut entrer les commandes les unes à la suite des autres dans un fichier *script* dont l'extension doit être `.py`. Pour créer un nouveau fichier `.py`, aller dans le menu "Fichier, Nouveau..."
- Pour exécuter un script, s'il est ouvert dans Pyzo, il suffit de l'exécuter en cliquant sur le menu "Run, Run file as script".

Packages et modules :

Une des forces du langage Python réside dans le nombre important de bibliothèques logicielles externes (packages et modules) disponibles. Il s'agit d'ensembles de fonctions regroupées et mises à disposition afin de pouvoir être utilisées sans avoir à les réécrire.

Les packages principaux que vous serez amenés à utiliser sont les suivants.

- **NumPy**, qui permet d'effectuer des calculs numériques avec Python. Elle introduit une gestion facilitée des tableaux de nombres.
- **matplotlib**, bibliothèque très riche permettant de tracer des courbes 2D.
- **SciPy** qui sera la plus utilisée, bibliothèque de calcul scientifique, qui contient NumPy, matplotlib et bien d'autres modules (résolution de système linéaire, recherche de valeur propre, transformée de Fourier, etc.).

Pour utiliser un module ou un package, il faut d'abord l'importer. Par exemple, pour avoir accès à la valeur approchée de π qui se situe dans le package NumPy, on entre les commandes suivantes :

```
import numpy as np
np.pi
```

2 Manipulation des objets de base

Pour tous les exercices suivants, écrire les commandes suivantes dans l'interpréteur Python et observer les réponses afin d'identifier l'utilisation de chacune des commandes.

Exercice 1 - Calculatrice, variables scalaires, premiers tableaux

```
5
2+5
2345/34578
2**3
pow(2,3)
5**(1/2)
pow(5,1/2)
```

Attention! Que renvoient les commandes 2^3 et $5^{(1/3)}$?

```
a=10
a
b=a+5
b
c=1,2,3
print(a,b,c)
a,b=b,a
print(a,b)
```

```
s="Hello!"
print(s)
```

```
c=[1,-2,7,0,10]
d=[3, 4]
c+d
2*c
10*[0]
print(c[0], c[1])
print(c[-1], c[-2])
c[0]+c[1]+c[2]+c[3]+c[4]
```

```
e=range(10);
print(e)
```

```
e=list(e)
print(e)
```

```
e=range(5, 11)
print(list(e))
```

```
e=[i**2 for i in range(11)]
print(e)
max(e)
min(e)
sum(e)
```

On va charger le package sous le nom `np` pour toute la suite du TP grâce à la commande suivante.

```
import numpy as np
```

Les fonctions usuelles sont alors disponibles via NumPy.

```
x=-1
y=2
```

```
np.sqrt(y)
np.exp(x)
np.log(y)
np.sin(y)
np.maximum(x, y)
np.minimum(x, y)
```

Ce package permet également d'avoir accès à plus d'objets mathématiques.

```
np.pi
np.e
np.cos(2*np.pi)
np.complex(2, 3)
```

Exercice 3 - Tableaux NumPy

Les méthodes disponibles avec les tableaux Python sont limitées. On a accès à beaucoup plus de méthodes avec les tableaux NumPy.

```
c=np.array([1,-2,7,0,10])
2*c
c[3]
c[0]+c[1]+c[2]+c[3]+c[4]
c[0]=0
np.size(c)
```

```
a=np.array([[0, 1], [0, 0]])
a[1,1]=1
a[0,0]=a[1,1]
np.shape(a)
```

Il existe de nombreuses manières de construire des tableaux NumPy de manière plus ou moins automatiques. Pour les commandes suivantes, identifier comment les constructions.

```
np.zeros((2, 3))
3*np.ones((2, 3))
np.identity(5)
```

```
np.linspace(0, 2*np.pi, 10)
np.linspace(20, 1, 10)
x,dx=np.linspace(0, 2*np.pi, 10, retstep=True)
print(x, dx)
```

Exercice 2 - Le calcul avec Numpy

Le package Numpy est LE package de référence pour le calcul numérique en Python. On l'utilisera en permanence.

```
np.arange(0, 10, 0.5)
dx=0.5
np.arange(0, 10+dx/2, dx)
np.arange(10, -dx/2, -dx)
```

Exercice 4 - Opérations sur les tableaux

Il existe une multitude d'opérations sur les tableaux NumPy. Pour les commandes suivantes, identifier l'action de chacune des fonctions.

```
a=np.array([[0, 1], [0, 0]])

np.hstack((a, a))
np.concatenate((a, a), axis=1)

np.vstack((a, a))
np.concatenate((a, a), axis=0)

x=np.hstack((0*a, a, 2*a, 3*a))
print(x)
np.amax(x)
np.amin(x)
np.sin(x)

E=np.empty((3, 4))
np.empty_like(E)
np.zeros_like(E)
```

Il est possible d'extraire un sous tableau d'un tableau plus grand. Pour cela, il suffit de taper : `tableau[début:fin+1, début:fin+1]`.

```
A=np.array([[1, 2, 3],
            [11, 12, 13],
            [21, 22, 23]])
A.shape
A[2, 1]
A[0:2, 1:3]
A[:, 1]
A[0, :]
A[-1, :]
A[:, -3]
A[[0, 2], [0, -1]]
A>11
A[A>11]
```

Exercice 5 - Opérations matricielles

Il est possible de créer une matrice soit avec le type `array` soit avec le type `matrix`.

🔧 Avec le type `array`.

```
A=np.array([[4, 0, 1], [1, 4, 1], [1, 0, 4]])
B=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
2*A-B
```

Comme souvent dans Python, il existe plusieurs commandes ayant la même fonction.

- Multiplication matricielle : `A*B`, `A@B`, `np.matmul(A, B)`.
- Transposition : `A.T`, `A.transpose()`, `np.transpose(A)`.
- Puissance matricielle : `A**3`, `A@A@A`, `np.linalg.matrix_power(A, 3)`.

Dans ce dernier cas, on utilise le module `linalg` (pour Linear Algebra) de NumPy, que l'on reverra plus tard.

🔧 Avec le type `matrix`

```
C=np.matrix(A)
C**3

np.trace(A)
np.linalg.det(A)
np.linalg.inv(A)
```

⚠ Il y a une différence entre le type `matrix` et le type `array` pour certaines opérations. C'est en particulier le cas pour la multiplication matrice/vecteur :

🔧 Avec le type `array`.

```
A=np.array([[4, 0, 1], [1, 4, 1], [1, 0, 4]])
b=np.array([1, 1, 1])
A*b
np.matmul(A, b)
np.dot(A, b)
```

🔧 Avec le type `matrix`.

```
C=np.matrix(A)
x=C*b
x=C*b.T
print(b.shape, np.transpose(b).shape)
x=C*b[:, np.newaxis]
x=C*b.reshape((b.size, 1))
```

Exercice 6 - Attentions aux surprises

Qu'observez-vous pour chacun des deux cas suivants ?

```
B=A
B[0,0]=0
print(B)
print(A)
```

```
C=A.copy()
C[0,0]=100
print(C)
print(A)
```

Attention à la copie de tableaux!

Exercice 7 - Broadcasting

La suite de commande suivante donne un résultat qui peut sembler surprenant.

```
c=np.array([1, -2, 7, 0, 10])
d=np.array([1, 2, 3, 4, 5]).reshape((5, 1))
c+d
```

Numpy utilise du **broadcasting**. On somme un vecteur ligne avec un vecteur colonne, les dimensions ne sont donc pas les mêmes et on pourrait s'attendre à avoir un message d'erreur. Ici, Numpy retourne quand même un résultat qui pourrait sembler bizarre à première vue. Le broadcasting fonctionne de la manière suivante :

- On a un tableau *a* et un tableau *b*.
- Pour chaque dimension de *a* et *b*,
 - si elles sont égales, c'est bon, on passe à la dimension suivante,
 - si elles sont différentes et que l'une d'elle est égale à 1, le tableau en question est dupliqué le long de cette dimension égale à 1 pour être égale à l'autre,
 - sinon, un message d'erreur est retourné disant que le broadcasting ne peut pas être fait.
- L'opération demandée est ensuite effectuée sur les tableaux potentiellement "redimensionnés"

Dans notre cas :

```
print(c.shape)
print(d.shape)
```

c a pour dimensions (5,) et *d* a pour dimensions (5,1). Le broadcasting fonctionne donc puisque la première dimension de *c* est 5 et celle de *d* est 1. Numpy va donc dupliquer *d* le long de sa première dimension pour atteindre une taille de 5 comme *c*. La deuxième dimension de *d* est 5 et *c* n'en a pas. Numpy considère que c'est 1 et duplique donc *c* le long de sa "deuxième" dimension pour atteindre 5 comme *d*. Au final, *c* et *d* sont vues comme des matrices 5×5 et Numpy effectue la somme de ces matrices. Le résultat est donc bien une matrice 5×5.

Cela peut être perturbant mais le broadcasting est une fonctionnalité extrêmement performante. Il est utilisé dans de nombreuses fonctions de Numpy.

Exercice 8 - Graphiques

Il existe beaucoup de packages en Python pour tracer des graphiques. Le plus utilisé est **matplotlib**.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x=np.linspace(0, 2*np.pi, 100)
plt.plot(x, np.sin(x))
plt.show()
```

```
plt.plot(np.cos(x), np.sin(x))
plt.show()
```

```
plt.plot(x, np.cos(x), x, np.sin(x))
plt.show()
```

```
plt.plot(x, np.cos(x))
plt.plot(x, np.sin(x))
plt.show()
```

Que fait la commande **show** ?

Pour afficher plusieurs fenêtre :

```
x=np.linspace(0, 2*np.pi, 100)
y=np.linspace(0., 1., 100)
```

```
plt.figure(1)
plt.plot(x, np.sin(x))
```

```
plt.figure(2)
plt.plot(y, y * np.sin(y))
```

```
plt.show()
```

Pour continuer sans devoir fermer toutes les figures :

```
x=np.linspace(0, 4 * np.pi, 30)
plt.plot(x, np.sin(x), 'r*--')
plt.show(block=False)
```

Le troisième argument est une option permettant de spécifier la couleur de la courbe (**r** rouge), la représentation des coordonnées (***** étoile) et le type du trait (**--** discontinu). Voir l'aide de matplotlib pour avoir la liste des options possibles.

On peut également mettre une légende, donner des titres aux axes et à la figure.

```
plt.plot([0, 1], [1, 0], 'b')
plt.plot(0.1, 0.9, 'r+')
plt.xlabel('x')
plt.ylabel('y')
plt.title('joli graph')
plt.legend(['segment', 'point'])
plt.show()
```

La commande **subplot(m, n, i)** permet de tracer *m* × *n* graphiques sur une même figure (voir l'aide).

```
x=np.linspace(0, 2*np.pi, 100)
plt.subplot(2, 2, 1)
plt.plot(x, np.sin(x))
```

```
plt.subplot(2, 2, 2)
plt.plot(np.cos(x), np.sin(x))
```

```
plt.subplot(2, 2, 3)
plt.plot(x, np.cos(x), x, np.sin(x))
```

```
plt.subplot(2, 2, 4)
plt.plot(x, x * np.sin(x))
```

```
plt.show()
```

3 Programmation

Exercice 1 - Boucles

Comme bien d'autres langages informatiques, Python permet à l'utilisateur de définir des sous-programmes sur mesure, avec un ou plusieurs arguments.

⚠ Python utilise les tabulations pour définir si les instructions font parties des blocs ou non. Il n'y a, en effet, pas de mot-clé pour marquer la fin des blocs en Python. Les symboles ":" après les mots-clefs marquent le début des blocs et sont donc obligatoires.

🔗 Boucle for

```
for i in range(1, 10):  
    print(i)  
  
for i in [0, 1, 2, 3]:  
    print("i a pour valeur", i)
```

🔗 Boucle while

```
x = 1  
while x < 10:  
    print("x a pour valeur", x)
```

🔗 Boucle if

```
if n % 3 == 0:  
    print(n, "=3k")  
elif n % 3 == 1:  
    print(n, "=3k+1")  
else:  
    print(n, "=3k+2")
```

Les mot-clés `elif` et `else` ne sont pas obligatoires.

Exercice 2 - Fonctions

🔗 Fonction

```
def somme(n):  
    S=0  
    for i in range(n):  
        S=S+i+1  
    return S
```

Une fonction peut ne rien retourner, il suffit de ne pas mettre le mot-clé `return`. Ici aussi il faut respecter les tabulations.

Une même fonction peut souvent être écrite de façon *itérative* (i.e. en utilisant une boucle et des tests) ou *réursive* (i.e. sans utilisation de boucle, la fonction faisant appel à elle-même pour le calcul de ses valeurs, analogue informatique de la récurrence mathématique).

```
def somme_rec(n):  
    if n==0:  
        return 0  
    else:  
        return n+somme_rec(n-1)
```

⚠ La récursion coûte cher, il faut donc la réserver aux cas où on ne peut pas faire de version itérative!

Exercice 3 - Premiers programmes

1. Sans utiliser de fonction prédéfinie, écrire une fonction `maximum` calculant le maximum entre deux réels a et b .
→ Utiliser une boucle `if`.
 2. Écrire une fonction `prod_first_odds` qui calcule le produit des n premiers entiers impairs.
→ Utiliser une boucle `for`.
 3. Sans utiliser de fonction prédéfinie, écrire deux fonctions calculant $n!$ pour $n \in \mathbb{N}$:
 - une fonction itérative `factorielle_it`
 - une fonction réursive `factorielle_rec`
-