Université Lyon 1 Année 2018-2019
Master Mathématiques Générales 1ère année
Logiciels Scientifiques

**TP1 – Python**

# 1 Pour commencer

Python est un langage de programmation interprété haut niveau. Une implémentation Python peut-être téléchargée, pr exemple avec Miniconda3 (`https://conda.io/miniconda.html`). Une fois Python installé, vous pouvez installer un IDE (un environement de développement) pour travailler avec une interface graphique. On pourra par exemple penser à Pyzo ou Spyder.

**Sur les machines des salles de TP, Python et Pyzo devraient déjà être installés, vous n'avez donc pas à faire tout ça**.

**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 connaitre 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 intéractive.
- 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.

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/3.5/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 fenêtre que l'on afficher 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 précedemment 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'éxécuter 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 courbe 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 $\pi$ qui se situe dans le package NumPy, on entre les commandes suivantes :

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

## 2 Exercices préliminaires

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
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)

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

---

**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. On va donc 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 limités. En créant des tableaux NumPy, on a accès à beaucoup plus de méthodes.

```
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)

d=np.array([[0, 0, 0], [0, 0, 0]])
print(d)

d=np.zeros((2, 3))
print(d)
np.shape(d)

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

---

**Exercice 4 - Opérations sur les tableaux**

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

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

```
b=np.hstack((a, a))
print(b)
b=np.concatenate((a, a), axis=1)
print(b)

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

c=np.hstack((0*a, a, 2*a, 3*a))
print(c)

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

x=np.arange(0, 10, 0.5)
print(x)

dx=0.5
x=np.arange(0, 10+dx/2, dx)
print(x)

y=np.arange(10, -dx/2, -dx)
print(y)

np.amax(x)
np.amin(x)
np.sin(x)
```

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]])
A-B
A*B
A@B
np.matmul(A, B)

A.T
A.transpose()
np.transpose(A)

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.

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])
x=A*b
x=np.matmul(A, b)
x=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))
```

Résolution simple de système linéaire :

```
x=np.linalg.solve(A, b)
```

Création de matrices de différentes formes

```
C=np.identity(5)
D=np.zeros((3, 4))
E=3*np.ones((3, 4))
F=np.empty((3, 4))
F=np.empty_like(E)
F=np.zeros_like(E)
```

---

### Exercice 6 - Attentions aux surprises

Qu'obervez-vous dans le cas suivant ?

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

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 - Graphiques

Il existe beaucoup de package 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 $m \times n$ 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()
```

---

### Exercice 8 - Programmation

La structure des boucles et des fonctions de Pyhon et de Sage sont les mêmes.

1. En utilisant une boucle `for`, programmer une fonction itérative qui calcule la somme des $n$ premiers entiers et qui affiche les sommes intermédiaires.
2. En utilisant une boucle `if`, programmer une fonction qui calcule le $n$-ième terme de la suite de Fibonacci.
3. Le tri par insertion consiste à prendre les éléments d'un tableau un à un et de les insérer à la bonne place dans un tableau déjà trié. Écrire une fonction `tri_insertion(T)` qui implémente le tri par insertion. Tester la fonction sur un tableau $T$ de taille $n$ contenant des entiers aléatoires entre 0 et $n$ (commande `np.random.randint`).