Université Lyon 1 Année 2019-2020
Licence Mathématiques 2ème année
Introduction à l'Analyse Numérique

TP0 : Introduction à Python

1 Pour commencer

Python est un langage de programmation interprété haut niveau. De nombreux packages sont disponibles pour enrichir ses fonctionnalités. Une implémentation Python peut-être téléchargée avec Miniconda3 par exemple

`https://conda.io/miniconda.html`

Une fois Python installé, il y a plusieurs possibilités :

1. utilisation d'un *Notebook*,
2. utilisation d'un *IDE* (environnement de développement), par exemple *Pyzo* ou *Spyder*.

Il n'y a pas d'installation à effectuer sur les machines de salles de TP. Si vous souhaitez installer ces logiciels sur une machine personnelle, n'hésitez pas à demander aux chargés de TP.

Cette année, on utilisera en priorité les notebooks.

Pour les TPs suivants, du code à compléter sera mis en ligne.

1.1 Notebook

Utiliser cet outil permet de combiner texte et code. La mise en forme se fait à l'aide de *cellules*.

Chaque cellule s'exécute avec la commande "Play", ou en utilisant "Maj+Entrée".

La sauvegarde se fait sous la forme d'un fichier avec l'extension `ipynb`, on peut aussi exporter sous forme de script Python (extension `.py`), ou de fichier pdf par exemple.

Pour utiliser l'aide (par exemple pour la fonction print), on peut écrire `help(print)` ou `? print` dans une cellule, puis l'exécuter.

1.2 IDE

L'interface d'un IDE est divisée en plusieurs fenêtres :

— Un éditeur de texte pour taper son *script* Python directement dans un fichier et le sauvegarder (sous la forme de fichiers `.py`).

— 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 peut afficher (ou pas) en utilisant le menu "Outils" : par exemple l'aide interactive.

On écrit le code dans l'éditeur de texte. On peut alors sélectionner une partie du code et l'exécuter.

1.3 Conseils généraux

— Ne pas hésiter à consulter l'aide (ou l'aide en ligne).

— Bien respecter la syntaxe, la casse (majuscules/minuscules), et les espaces (en particulier pour les boucles et les fonctions).

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 - Un peu de Python

Effectuer des opérations :

```
5
2+5
2345/34578

2**3
5**(1/3)
```

Définition de variables :

```
a = 10
a

b = a + 5
b

a, b, c = 1, 2, 3
print(a, b, c)

a, b = b, a
print(a, b)

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

Les listes : il y aurait beaucoup à dire dessus, voici quelques opérations que l'on sera amené à utiliser plus tard.

```
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])

# créer l'intervalle des entiers entre
# 0 et 9
e = range(10)
print(e)
# pour en créer la liste
e = list(e)
print(e)

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

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

Les fonctions mathématiques usuelles sont accessibles dans le package `math` :

```
sqrt(2)
import math
math.sqrt(2)
```

Nous allons maintenant voir comment effectuer des tests et des boucles. Pour ces commandes, le plus simple est de les écrire dans un fichier externe, que vous pouvez appeler `tuto.py` par exemple, et de l'exécuter ensuite. Pour cela, on ouvre le menu "Fichier", on créé un nouveau fichier et l'on l'enregistre avec le nom voulu (avec "Fichier, Enregistrer" ou `Ctrl-S`). Pour exécuter toutes les commandes dans le fichier, vous pouvez cliquer sur "Démarrer le script" dans le menu "Exécuter".

— Les tests avec le mot-clé `if`. La syntaxe est la suivante :

```
if une_condition:
  instructions
elif une_autre_condition:
  instructions
else:
  instructions
```

Remarques :

— Les symboles `:` après le `if`, `elif` et `else` sont obligatoires. Ils marquent le début des blocs.

— Les tabulations sont importantes ! Elles définissent 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 mot-clés `elif` et `else` ne sont pas obligatoires.

Par exemple :

```
x = 42
if x < 0:
```

```
    x = 0
    print('Négatif, changé en 0')
elif x == 0 or x == 1:
    print('Zéro ou Un')
else:
    print('Plus grand que 1')
```

— Les boucles `for`. La syntaxe est :

```
for variable_boucle in une_variable:
    instructions
```

Remarques :

— la variable `variable_boucle` va, une par une, prendre les valeurs dans la variable `une_variable`. Pour chacune de ces valeurs, les instructions dans le bloc `for` seront exécutées.

— `une_variable` peut être, par exemple, une liste Python ou un tableau (on le verra plus tard).

— Comme pour les `if`, les `:` et les tabulations sont nécessaires.

Par exemple :

```
for i in range(5):
    print(i*i)
```

— Les boucles `while`. La syntaxe est :

```
while condition:
    instructions
```

Les instructions dans le bloc `while` sont exécutées tant que `condition` est vraie. Exemple (suite de Fibonacci) :

```
a, b = 0, 1
while b < 1000:
    a, b = b, a + b
    print(round(b/a, 3), end=", ")
```

Exercice 2 - Le calcul avec Numpy et les premiers tableaux

Le package **Numpy** est **LE package de référence** pour le calcul numérique en Python. Il doit être importé avec la commande `import numpy`.

```
# Les listes Python
c = [1, -2, 7, 0, 10]
2 * c

# Avec un tableau Numpy
import numpy as np
c = np.array([1, -2, 7, 0, 10])
2 * c
print(2 * c)

c.size
np.size(c)

c[3]
c[0] + c[1] + c[2] + c[3] + c[4]
c[0] = 0

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

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

np.cos(2 * np.pi)
np.sqrt(-1)

2 + 3j
np.complex(2, 3)
```

Exercice 3 - Tableaux et fonctions associées

Quelques opérations sur les tableaux numpy. Bien observer le résultat de chaque commande pour comprendre son fonctionnement.

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

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)

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]
```

La dernière commande montre qu'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]`.

Il est aussi possible d'extraire un ensemble d'éléments qui ne forment pas un sous bloc :

```
print(A[[0, 2], :])
print(A[[0, 2], [0, -1]])
print(A > 11)
print(A[A > 11])
```

Attention à la copie de tableaux :

```
B = A
B[0, 0] = -1
print(B)
print(A)

C = A.copy()
C[0, 0] = 1
print(C)
print(A)
```

Exercice 4 - Fonctions usuelles

Ces fonctions existent à la fois dans le package math et dans le package Numpy. Comme on utilisera essentiellement des tableaux Numpy, il est préférable d'utiliser les versions Numpy.

```
x = -1
y = 2

np.abs(x) # avec Numpy
math.abs(x) # avec math

np.maximum(x, y)
np.minimum(x, y)

np.sqrt(y)
np.exp(x)
np.log(y)
np.sin(y)
```

Ces fonctions marchent aussi sur des tableaux en opérant élément par élément. Essayer !

Exercice 5 - Opérations matricielles

Il est possible de créer une matrice soit avec le type `array` soit avec le type `matrix`. Observer bien les résultats des commandes suivantes :

```
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)

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 :

```
b = np.array([1, 1, 2])

# avec le type array
x = A * b
x = np.matmul(A, b)
x = np.dot(A, b)

# avec le type matrix
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))
x = C * np.asmatrix(b).T
```

Résolution simple de système linéaire :

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

Pour plus de méthodes de résolution de systèmes linéaires, le package `scipy` est la référence. C'est le package Python contenant des méthodes numériques haut-niveau :

- Format sparse de matrices
- Résolution de systèmes linéaires
- Recherche de valeurs propres
- Transformée de Fourier
- Interpolation
- Méthodes d'intégration numériques
- Résolution de systèmes d'équations différentielles
- ...

On l'utilisera plus tard dans d'autres TP.

```
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 - Opérations sur les tableaux

Comme on l'a déjà un peu vu, les opérations arithmétiques sur les `array` se font élément par élément.

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

# tableau 2d, 3 lignes, 1 colonne
b = np.ones((3, 1))
c = np.array([1, 2, 3]).reshape((3, 1))

B**2
A * B
b / c
A**np.array([[2, 1, 2], [2, 0, 1], [1, 1,
    1]])
```

Exercice 7 - 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()
```

Les graphiques sont tracés uniquement lorsque la commande `show()` est lancée. Pour afficher plusieurs fenêtres :

```
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 graphe')
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 - Messages d'erreur et autres problèmes

Dans les commandes suivantes, identifier le message d'erreur ou le comportement inattendu et corriger la commande lorsque cela est possible.

```
toto

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