Python : installation et prise en main

L'objet de ce document est de donner quelques éléments d'information sur l'installation et la prise en main du logiciel Python. Il est largement inspiré du document analogue réalisé par Émeline Luirard pour la préparation à l'agrégation.

- Python est un logiciel libre qu'on peut télécharger sur https://www.python.org/. Nous travaillerons avec Python 3. De nombreux environnements de travail sont disponibles, parmi eux Pyzo est téléchargeable sur https://pyzo.org/. Pour l'installation, suivez les indications du site Pyzo, ou si cela vous parle, saisissez les commandes ci-dessous dans un terminal :

```
$ sudo apt-get update
$ sudo apt-get install python3.6
$ sudo apt-get install python3-pip python3-pyqt5
$ sudo python3 -m pip install pyzo --upgrade
```

  Ensuite, il suffit d'ouvrir Pyzo.
- Python possède une aide consultable en tapant `help()` dans la console. Plus généralement pour obtenir une aide concernant une fonction `bidule` du module `machin`, tapez `help(machin.bidule)`. Vous pouvez aussi ouvrir la fenêtre de l'aide interactive, dans le menu Outils.
- Pour vous familiariser avec les fonctions de base de Python, vous trouverez de nombreux tutoriels sur la toile, par exemple ici.

1 Prise en main

1.1 Fenêtre de commande, éditeur, aide en ligne

Lorsqu'on lance le logiciel Pyzo, la fenêtre est découpée en divers cadres avec notamment :

- une fenêtre de commande (interpréteur) : où l'on peut exécuter des commandes : >>> indique que le logiciel attend vos instructions. L'interpréteur peut être utilisé comme une calculatrice : vous entrez un calcul, vous tapez sur la touche "entrée" et Python vous donne le résultat.
- une fenêtre d'éditeur de texte : où l'on peut regrouper les commandes et les sauvegarder dans un `fichier.py`.

On peut, dans la fenêtre de Python copier et coller des lignes à l'aide de la souris (y compris les exemples donnés par l'aide) ; on peut aussi naviguer dans l'historique des commandes au moyen des flèches haut et bas, cela évite de retaper les commandes si l'on exécute une fonction plusieurs fois de suite par exemple.

Python est un logiciel de calcul numérique et non de calcul formel : il effectue des calculs numériques approchés et non des calculs exacts. Par défaut, les réels sont affichés avec 15 chiffres significatifs.

```
-->pi
pi  =
3.141592653589793
```

1.2 Programmation via l'éditeur

Dès que l'on souhaite exécuter plus de deux ou trois tâches consécutives, i.e. dès que l'on sort du mode d'utilisation "supercalculatrice" de Python, on utilise l'éditeur de texte. On enregistre le fichier obtenu avec le suffixe `.py` puis on exécute le fichier dans Pyzo, au moyen du menu `Exécuter...` ou par un raccourci clavier. Pour exécuter seulement un bout de code, vous pouvez le séléctionner et cliquer sur `Exécuter la sélection`.

Une ligne de commentaires commence par `#`. De manière générale, Python ignore tout ce qui, sur une ligne, suit la commande `#`.

```
Debut programme
[...]
# on definit maintenant les abscisses et ordonnées
x=np.arange(10) ; # le vecteur des abscisses  entier de 0 à 9
y=np.sin(x) ; # le vecteur des ordonnées
[...]
Fin du programme
```

Si vous voulez afficher une variable, vous devrez utiliser `print(variable)`.

1.3 Importation et chargement de modules

Pour exploiter tout le potentiel de Python dans un contexte mathématique, il est nécessaire d'installer ou d'importer un certain nombre de paquets, modules et sous modules dédiés, contenant par exemple des fonctions spéciales ou des outils graphiques.

On installera ainsi "**une fois pour toutes**" les modules suivants :

```
install numpy
install matplotlib
install scipy
```

Ensuite, à **chaque début de session** (i.e. quand vous ouvrez Pizo), il faudra importer un certain nombre de modules, par exemple :

- `math` : pour importer les fonctions mathématiques usuelles et certaines constantes usuelles comme π (`pi`).
- `cmath` : pour les complexes : (`1j` pour i).
- `numpy` : pour utiliser le type `array` (tableau dont les éléments sont tous du même type, pratique pour les vecteurs, les matrices). On conseille
- `scipy` : outils nécessaires au calcul matriciel. Il contient un sous-module qui nous servira pour la partie aléatoire.
- `random` : Ce module implémente des générateurs de nombres pseudo-aléatoires.
- `matplotlib` : pour générer des graphiques.

Voici un exemple typique de début de session :

```
from random import *
import math
import numpy as np
import numpy.random as npr
from matplotlib import pyplot as plt
```

Comme vous le constatez, plusieurs syntaxes sont possibles.

```
import bidule
#importe tous les éléments du module bidule
from nom_du_module1 import *
#importe tous les éléments du module1
import nom_du_module2 as mod2
#importe le module 2 avec le nom raccourci mod2
from module3 import sous_module as smod
```

Dans le cas du module2, les fonctions du module2 devront être appelées par `mod2.fonction()`, et pour le sous module dans le dernier exemple, on écrira de manière analogue `smod.fonction()`. Par exemple, si l'on importe le module `numpy` sous le diminutif `np`, i.e. si on execute la commande `import numpy as np`, alors la fonction racine carré (qui est dans le module `numpy`) sera appelée comme via `np.sqrt(.)`, autrement dit `np.sqrt(36)=6`.

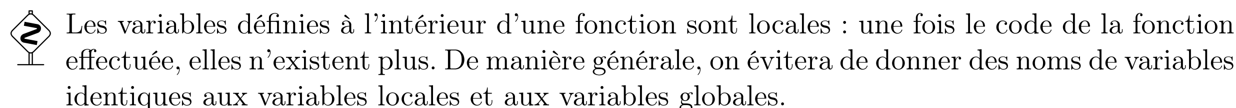
2 Syntaxe et programmation de base

2.1 Fonctions

Comme mentionné plus haut, dès que l'on sort du mode calculatrice de la fenêtre de commande de Python et que l'on souhaite réaliser des tâches quelque peu complexes, il est raisonnable d'utiliser l'éditeur et d'écrire des programmes ou fonctions, auxquels on peut faire appel de façon répétée. La syntaxe est la suivante

```
def nom_de_la_fonction(parametres):
        """ Details de ce que fait la fonction"""
        calculs
        return valeur_sortie
```

Dès que Python trouve un `return truc`, il renvoie `truc` et abandonne ensuite l'exécution de la fonction.


Les variables définies à l'intérieur d'une fonction sont locales : une fois le code de la fonction effectuée, elles n'existent plus. De manière générale, on évitera de donner des noms de variables identiques aux variables locales et aux variables globales.

On donnera des noms longs et explicites aux variables globales.

N.B. *Vous pouvez mettre des paramètres pas défaut, un nombre de paramètres arbitraire, etc. On s'en sortira sans mais pour ceux que ça intéresse, voir [?, p. 42] et/ou [?, p. 14].*

2.2 Opérateurs logiques et booléens

Les opérations logiques de bases sont le "et" logique (`and`), le "ou" logique (`or`) et la négation (`not`). Les valeurs logiques sont `True` et `False`. Voici divers opérateurs de comparaison :

`x==y`	x est égal à y
`x!=y`	x est différent de y
`x<=y`	$x \leq y$
`x in y`	x appartient à y (pour les listes)

Pour demander des conditions plus complexes, on peut combiner ces opérateurs de comparaison avec les booléens, par exemple, `(x==y) or (x>5)`.

2.3 Boucles

Même si on cherche en général à l'éviter pour la rapidité d'exécution des programmes, l'usage des boucles est parfois inévitable. Les syntaxes des boucles `for`, `while` et des instructions conditionnelles sont les suivantes :

```
i, aux = 0,2
while (i<5)and(aux<10):
        aux=aux**2
        i+=1
```

```
n=5
for i in range(10):
        a[i]=1/(i+1)
        a[i]=1/(i+1)
```

```
if i == j:
        a[i,j] = 2
else:
        a[i,j] = 0
```

Vérifiez impérativement que vos boucles `while` vont s'arrêter !

On veillera à bien indenter les codes dans l'éditeur : en Python, il n'y a pas de `end`, c'est l'indentation qui indique lorsque la boucle est fermée.

N.B. *Lorsqu'il y a plus de deux cas, on utilisera* `elif` *:*

```
if i == j:
        a[i,j] = 2
elif i == j+1:
        a[i,j] = 0
else:
        a[i,j] = 1
```

2.4 Opérations usuelles

`x+=1`	remplace x par $x+1$ (valable aussi pour -,*,/)
`x**a`	x^a
`x,y=17,42`	pour affecter deux variables en même temps, toutes les expressions sont évaluées *avant* la première affectation, ex : `x,y=x+2,x`
`x,y=y,x`	du même type que le précédent, pour échanger deux variables

2.5 Listes

Une liste est une séquence d'éléments rangés dans un certain ordre, elle peut contenir des objets de types différents, par exemple, `liste=[42,'reponse']`, et sa taille peut varier au cours du temps.

En Python, on commence à compter à 0.

Pour créer une liste, connaître sa longueur et accéder à ses éléments :

`liste=[]`	crée une liste vide
`len(liste)`	renvoie la longueur de la liste
`liste[i]`	extrait l'élément à l'indice i de la liste (en comptant à partir de 0)
`liste[-1]`	extrait le dernier élément
`liste[2:4]`	extrait une sous liste

Les "intervalles" dans Python sont fermés à gauche et ouverts à droite : `liste[2:4]` extrait les éléments de l'indice 2 à 3 de la liste.
Pour créer des listes, Python est sympa, grâce aux listes en compréhension :
`[x**2 for x in range(10)]` retourne la liste `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`.
Enfin, vous pouvez transformer une liste en ensemble (i.e. pas de doublons) par `set(liste)`.

Modification de listes :

`ma_liste[1]=4`	remplace l'élèment à l'indice 1 par 4
`liste1+liste2`	concatène deux listes
`ma_liste.append(5)`	ajoute un élément à la fin de la liste
`ma_liste.extend(L)`	ajoute en fin de liste les éléments de L
`ma_liste.insert(i,x)`	insère un élément x en position i
`ma_liste.remove(x)`	supprime la première occurence de x
`ma_liste.sort()`	trie la liste
...	

Taper `help(list)` pour obtenir toutes les "méthodes" associées. (Une méthode est une fonction qui ne s'applique qu'à un type d'objet, ici les `list`.)

On utilise assez souvent la fonction `range(start, stop[, step])` qui renvoie les entiers de start à stop. Attention, cependant, ce n'est pas une liste, c'est ce qu'on appelle un "itérable" : un curseur qui se déplace, tous ses entiers ne sont pas stockés. Si on veut une liste, on pourra utiliser `list(range(...))` ou `[range(...)]`.

Attention quand vous copiez des listes :

```
>>> liste=[42, 'reponse']
>>> copie=liste
>>> liste[1]=17
>>> liste, copie
```

```
[42,17], [42,17]
>>> copie[0]=0
>>> liste, copie
[0,17], [0,17]
```

Les variables `liste` et `copie` pointent vers le même objet. On dit qu'elles font "référence" au même objet. Pour faire une copie, on peut utiliser `copie=liste[:]`, mais attention si la liste contient des sous-listes !

2.6 Algèbre linéaire : vecteurs, matrices

Un objet de type `array` est un tableau dont tous les éléments sont du même type. La différence avec les `list` : sa taille n'est pas modifiable une fois créé. Il servira à représenter les vecteurs et les matrices. Il est dans le module `numpy`. Dans la suite, je noterai M pour une matrice, V pour un vecteur, A pour un array quelconque.

```
M=np.array([[0,1],[2,3]]) # matrice
V=np.array([0,1,4,9])# vecteur ligne
```

Dans le module `scipy`, vous trouverez le sous-module `linalg` permettant de faire du calcul matriciel. Voici quelques commandes utiles pour manipuler les `array` :

Pour créer un `array`, connaître sa longueur et accéder à ses éléments :

`np.eye(n)`
`np.ones((n,m))`
`np.zeros((n,m))`
`np.empty((n,m))`
	matrice identité de taille n
	matrice 1 de taille $n \times m$
	matrice nulle de taille $n \times m$
	matrice vide de taille $n \times m$

`np.arange(debut, fin(non inclus), pas)`	le tableau des entiers de debut à fin avec un pas
`np.linspace(min, max, nb de points)`	points réguliers $[min, ..., max]$
`np.diag(V)`	matrice avec en diagonale V
`V[-1]`	dernier élément du vecteur V
`V[start:end:step]`	pour extraire une sous-liste
`M[i,j]`	élément à l'indice $(i_{ligne}, j_{colonne})$
`A.shape`	renvoie (nb de lignes, nb de colonnes)
`len(V)`	renvoie la longueur du vecteur
`A.reshape([n,m])`	redimensionne le tableau A en n lignes et m colonnes. Si l'une des valeurs vaut -1, `numpy` la trouve tout seul.
`V.reshape([nbLign,1])` ou `V[:,np.newaxis]`	transforme un vecteur ligne en colonne

Opérations matricielles usuelles :

`A.dot(B)`	multiplication matricielle : matrice/matrice, matrice/vecteur, vecteur/vecteur (produit scalaire)
`np.linalg.inv(M)`	inverse matriciel
`M.T`	transposée
`np.linalg.det(M)`	déterminant
`V[0:3]=5`	remplace les 3 premiers éléments de V par des 5.
`A>0`	renvoie un `array` de même taille que A avec des booléens.
`A[A>0]`	renvoie un `array` avec les éléments de A qui vérifient la condition Ici ça fait une copie et non une référence. Pour les conditions, on utilise `&` pour "et" et `|` pour "ou"

`np.sum(A)`
`np.sum(A,axis=0)`
`np.sum(A,axis=1)`
	somme des éléments de A
	pour sommer selon les colonnes,
	selon les lignes

`np.cumsum(A)`	somme cumulative des éléments de A, on peut encore utiliser `axis`
`np.mean(A)`	moyenne des éléments de A, on peut encore utiliser `axis`
`np.std(A)`	écart-type des éléments de A, on peut encore utiliser `axis`

`eigVal,eigVec=np.linalg.eig(M)`
 	
`eigVal,eigVec=np.linalg.eigh(M)`
	valeurs propres (avec multiplicité) et vecteurs propres à droite,
	en colonne, normalisés et dans le même ordre.
	pour les matrices symétriques, hermitiennes

Si vous cherchez quelque chose en particulier vous pouvez aussi utiliser `np.lookfor('create array')`.
Deux images assez explicites, pour l'extraction de listes, tirées de https://scipy-lectures.org :

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

```
>>> a[(0,1,2,3,4), (1,2,3,4,5)]
array([1, 12, 23, 34, 45])

>>> a[3:, [0,2,5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = np.array([1,0,1,0,0,1], dtype=bool)
>>> a[mask, 2]
array([2, 22, 52])
```

2.7 Tracer des graphes

Pour tracer des graphes, nous faut préalablement avoir chargé le sous module `pyplot` de `matplotlib`, par exemple comme décrit plus haut :

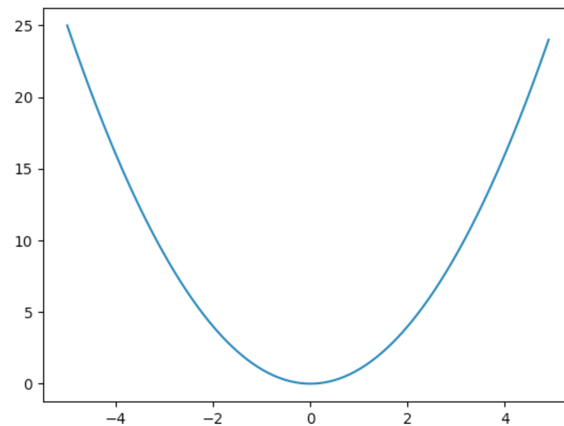
```
from matplotlib import pyplot as plt
```

Ensuite, si l'on veut le graphe de $x \mapsto x^2$ sur $[-5, 5]$, on peut procéder comme suit en discrétisant avec un pas $h = 0.1$:

```
x=np.arange(-5,5,0.1)
y=x**2
plt.plot(x,y)
```

Ce code génère le graphe mais ne l'affiche pas. Pour afficher le graphe à l'écran il faut ajouter

```
plt.show()
```



Si l'on veut effacer la fenêtre graphique, on écrit enfin `plt.clf()`. On peut bien sûr décorer la fenêtre graphique en ajoutant légende, nom des axes, etc.

```
fig, axs=plt.subplots(n,m) # on aura des sous-figures sur n lignes et m colonnes
#sans argument quand on n'en veut qu'un.
axs[i,j].plot(x,y,"r", label="nom fonction") # trace dans le graphique (i,j),
#en couleur rouge, et associe une legende
fig.suptitle("titre principal")
axs[0,0].set_title("titre graphe 1")
axs[0,1].set_title(r"$math latex$") # le 'r' dit à Python de ne pas interpreter
#les caracteres speciaux
#Ne pas oublier:
axs.legend() # affiche les légendes
```

N.B. *Le troisième argument de* `plot` *est le style du trait. On mettra "o" pour avoir des points non reliés.*

D'autres commandes :

`axs[..].set_ylabel("titre ordonnees")`	donne un titre aux ordonnées
`axs[..].set_ylim([a,b])`	limite le graphe aux ordonnées $[a, b]$
`ax[..].set_xticks(rray)`	graduation axe des abscisses
`ax[..].set_sticklabels(["0", "1",...])`	noms des graduations de l'axe des abscisses

Diagramme en bâtons : histogrammes de lois discrètes

`ax.bar(x,y,width=...)` à la place de `plot`.

Histogramme

`plt.hist(X,bins=n)` répartit les données de X dans n sous intervalles de $[min(X), max(X)]$. `hist` prend la place de `plot`, on garde la même syntaxe que précédemment pour faire différents graphes par exemple. Mais pour mettre plusieurs histogrammes sur le même on fait `plt.hist([X1,X2,X3], bins=n, label=...)`.

N.B. *Avec `a=plt.hist(X,bins=n)`, on récupère en premier la hauteur des bâtons et en deuxième le découpage des sous-intervalles fait par Python.*

N.B. *On peut donner à `bins` ses propres sous-intervalles dans un `array`.*

Pour superposer avec une densité, il faut demander à Python de normaliser les histogrammes, ça se fait avec l'option de `hist` : `density=True`. Puis on trace la densité.

```
a=[]
for k in range(10000):
    a=a+[-np.log(random.random())]
plt.hist(a,100,density=True)
x=np.arange(0,8,0.1)
y=np.exp(-x)
plt.plot(x,y,'r')
plt.show()
```