

MÉMO - PYTHON

Emeline LUIRARD

- Pourquoi Python ? C'est un langage lisible avec une syntaxe orientée présentation. Il est proche de la langue parlée, c'est ce qu'on appelle un langage de haut niveau.
N.B. la philosophie de ce langage est résumé dans le Zen de Python. Vous pouvez y accéder en important le module `this`.
- 💡 Votre devise doit être : **coder compact et simple !** C'est comme cela que vos codes seront les plus lisibles et que vous ferez le moins d'erreurs.

TP 1-3

🏹 Comment ça marche ?

Thonny, votre environnement de développement Python pour cette année, est constitué

- d'un interpréteur(ou console) : c'est votre boîte de dialogue avec Python. `>>>` 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. C'est pratique pour des instructions simples mais dès qu'on veut écrire du code plus long, pouvoir commenter et partager son code, il vaut mieux utiliser l'environnement de développement.
 - d'une fenêtre d'éditeur de texte : où l'on peut regrouper les commandes et les sauvegarder dans un `fichier.py`.
- 💡 On peut naviguer dans l'historique des commandes de l'interpréteur 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.

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.

Pour faciliter la lecture de vos codes pour plus tard, il est possible de les commenter. Une ligne de commentaires commence par `#`. De manière générale, Python ignore tout ce qui, sur une ligne, suit la commande `#`.

1 Modules

Il faut voir les modules comme des boîtes à outils. On commence par les "acheter" : les télécharger dans le menu Outils. Un module peut contenir de nouveaux types d'objets, de nouvelles fonctions. On ne les importe ("ouvrir sa boîte à outils à côté de soi") que lorsqu'on en a besoin dans la suite du code. Pour les importer, on écrira en haut de son fichier de code

```
import module1 as mod1 # pour importer module1 en le renommant mod1
```

ou

```
from module2 import sous_module as smod # pour importer un sous-module de module2
```

Les fonctions du module `module1` devront être appelées par `mod1.nom_de_la_fonction()`. On l'utilise pour ne pas qu'il y ait d'écrasement d'autres fonctions, pour ne pas encombrer l'espace des noms. Pour les sous modules, on fera `smod.fonction()`.

Voici les principaux modules :

- `math` : pour importer les fonctions mathématiques usuelles et certaines constantes usuelles comme π (`pi`).
- `numpy` : pour utiliser le type `array`. cf plus loin.
- `matplotlib` : pour générer des graphiques.

Le début de chaque script pourra commencer comme ceci :

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

💡 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)`.

2 Objets et types

En Python, tout est un objet. Pour manipuler ces objets, on leur donne un nom. On dit qu'on affecte une variable à une valeur. Par exemple,

```
y=42
mon_prenom="emeline"
```

N.B. En général, on n'utilise que des minuscules et des `"_"` pour les noms de variables. Et on essaie de les choisir pour qu'ils aient du sens.

Référencer une variable permet d'appeler plus facilement un objet et de garder sa valeur en mémoire. Il est possible d'afficher la valeur d'une variable avec `print(votre variable)`.

A chaque type d'objets est associé un ensemble de mécanismes. C'est ce qu'on appelle des *méthodes*. Voici une liste non-exhaustive de différents types :

- les entiers *int*, ils ont une précision illimitée. Par exemple, 5, 17, 42.
- Les flottants *float*, leur précision est limitée à 15 chiffres significatifs en général. Cela peut amener à des problèmes.



⌞ Pour les nombres décimaux, on utilise un point et non une virgule. Par exemple, 1.2

- Les chaînes de caractères *str*. On les note entre guillemets.

Par exemple, `"Bonjour, il fait beau aujourd'hui"`. Le guillemet indique à Python d'y lire comme du texte et non du code. Ainsi `"x"` sera lu comme du texte et non comme la variable `x`.

- Les listes *list*. On les note entre crochets, chaque élément étant séparé par une virgule. 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.

- 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 tableaux. Il est introduit avec le module `numpy`.

2.1 Opérations sur les nombres

Les quatres opérations de base sont `+`, `*`, `-`, `/`. Pour mettre à une puissance, on utilise `**`.

<code>x**a</code>	x^a
<code>a%b</code>	reste de la division euclidienne de a par b
<code>math.sqrt(x)</code>	\sqrt{x} en pensant à importer le module <code>math</code>
<code>min(a,b)</code>	renvoie le minimum entre a et b

2.2 Opérations sur les chaînes de caractères

<code>"Bonjour"+"Pierre"</code>	concatène les deux chaînes de caractères
<code>"A"*20</code>	concatène 20 fois la chaîne de caractères "A"
<code>len(chaine)</code>	donne la longueur d'une chaîne de caractères i.e. le nombre de caractères.

2.3 Opérations sur les listes

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

"Il faut penser les indices comme repérant non pas les tranches de saucisson, mais les coups d'opinel qui ont permis de couper les tranches."

Les "intervalles" dans Python sont fermés à gauche et ouverts à droite : `L[2:4]` extrait les éléments de l'indice 2 à 3 de la liste `L`.

Une image assez explicite, pour l'extraction de sous-listes/sous-array, tirée 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]])

```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

Modification de listes :

<code>ma_liste[1]=4</code>	remplace l'élément à l'indice 1 par 4
<code>liste1+liste2</code>	concatène deux listes
<code>ma_liste.append(5)</code>	ajoute un élément à la fin de la liste
<code>ma_liste.sort()</code>	trie la liste

Taper `help(list)` pour obtenir toutes les méthodes associées.

Pour aller plus loin :



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(...)]`.


2.4 Opérations sur les "array"

Voici quelques commandes utiles pour manipuler les `array` :

<code>np.arange(debut, fin, pas)</code>	le tableau des entiers de <i>debut</i> à <i>fin</i> (non incluse) avec un <i>pas</i>
<code>len(V)</code>	renvoie la longueur du vecteur
<code>V[-1]</code>	dernier élément du vecteur <i>V</i>
<code>V[start:end:step]</code>	pour extraire un sous-array, cf le paragraphe sur les listes
<code>np.asarray(L)</code>	créé un array à partir de la liste <i>L</i>

<code>A+10</code>	ajoute terme à terme
<code>A*A</code>	multiplication terme à terme
<code>1/A</code>	inverse terme à terme
<code>A**n</code>	puissance terme à terme
<code>np.exp(V)</code>	renvoie un objet de type array contenant l'image par l'exponentielle de chaque élément de <i>V</i> fonctionne également avec <code>np.log</code> et <code>np.sqrt</code> .

3 Quelques commandes usuelles

<code>x+=1</code>	remplace x par $x + 1$ (valable aussi pour $-$, $*$, $/$)
<code>x,y=17,42</code>	pour affecter deux variables en même temps, toutes les expressions sont évaluées <i>avant</i> la première affectation, ex : <code>x,y=x+2,x</code>
<code>x,y=y,x</code>	du même type que le précédent, pour échanger deux variables
<code>x=input("Donner une valeur")</code>	demande à l'utilisateur une valeur.  La variable <code>x</code> contient une chaîne de caractères.
<code>int(x), float(x), str(x)</code>	change le type de <code>x</code> respectivement en entier, flottant, chaîne de caractères.

4 Bloc d'instructions if/elif/else et for

4.1 Opérateurs logiques et booléens

Les opérations logiques de base sont le “et” logique (**and**), le “ou” logique (**or**) et la négation (**not**). Les valeurs logiques (ou booléens) sont **True** et **False**. Voici divers opérateurs de comparaison :

<code>x==y</code>	x est égal à y
<code>x!=y</code>	x est différent de y
<code>x<=y</code>	$x \leq y$

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

4.2 Tests conditionnels

Un bloc d'instruction est un ensemble d'instructions indentées du même nombre de caractères. On met **toujours** un `" :` avant un bloc d'instructions. La syntaxe est la suivante :

```
if condition1:
    instructions si condition1 realisee
elif condition 2:
    instructions si condition2 realisee
else:
    instructions si ni condition1 ni condition2 ne sont realisees
```

S'il n'y a qu'une condition à tester, il suffit d'enlever la ligne avec **elif**. Inversement, on peut rajouter autant de **elif** que nécessaire. Voici un exemple dans chaque cas :

```

if nb<30:
    x=0.10*nb
elif nb<50:
    x=0.05*nb
else:
    x=0.01*nb

if reponse=='oui':
    print("Super !")
else:
    print("Pourquoi?")

if note<10:
    print("Travail encore !")
elif note<12:
    print("Reçu")
elif note<14:
    print("mention AB")
elif note<16:
    print("mention B")
else:
    print("mention TB")

```

Les conditions sont testées dans l'ordre jusqu'à ce que l'une d'elles soient vérifiées (les suivantes ne sont donc pas testées). Donc attention à bien choisir l'ordre des tests ! Seul le bloc d'instructions ayant la condition réalisée sera lu et exécuté.



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.

4.3 Boucles for

Comme pour les tests conditionnels, on indente le bloc d'instructions du même nombre de caractères et on met **toujours** un " : " avant un bloc d'instructions. La syntaxe des boucles **for** est la suivante :

```

for nom_de_variable in iterateur:
    instructions repetitives

```

par exemple,

```

ma_liste=["soleil","bretagne","pluie"]
for i in range(0,len(ma_liste)):
    print(ma_liste[i])

```

Une boucle **for** définit une variable, ici nommée *i*, qui vit dans ce qu'on appelle un itérateur, ici `range(0,len(ma_liste))`. Elle prend comme valeur le premier élément de l'itérateur et exécute les instructions indentées, puis passe au deuxième élément de l'itérateur etc. Ici le code affiche les 3 éléments de la liste. On aurait aussi pu écrire cela :

```

ma_liste=["soleil","bretagne","pluie"]
for mot in ma_liste:
    print(mot)

```

💡 Donner des noms de variables explicites permet de savoir avec quel objet on travaille. Dans le code ci-dessus, la variable `mot` contient des chaînes de caractères. Dans la boucle précédente, la variable `i` contenait des entiers.

Si on veut avoir accès à la fois à la position et la valeur de l'objet dans l'itérateur, on pourra utiliser `enumerate(iterateur)`. Par exemple,

```
ma_liste=["soleil","bretagne","pluie"]
for i, mot in enumerate(ma_liste):
    print(mot, "est à la position", i)
```



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.

Il est tout à fait possible de mixer des tests conditionnels et des boucles `for`. Par exemple, pour trouver le nombre de "A" dans une chaîne de caractères, on peut procéder comme ceci :

```
ma_chaine='AOODJHFAEPZAFJRNFOZ'*
nb=0 #nombre de A
for lettre in ma_chaine:
    if lettre=="A":
        nb=nb+1
print(nb)
```

N.B. En fait, il existe une fonction toute faite qui vous permet de compter le nombre de "A" : `ma_chaine.count("A")`.

5 Illustrations graphiques

Pour tracer des graphes, nous avons besoin du sous-module `pyplot` de `matplotlib`. Une fois le vecteur contenant les abscisses et celui contenant les ordonnées créés, on peut tracer la fonction.

Par exemple :

```
x=np.arange(0,100,0.1)
y=1+np.exp(x)
plt.plot(x,y,"r+", label="nom de la courbe")
# le troisième argument détermine la couleur et le type de point de la courbe
# ici rouge, avec des croix
plt.xlabel("titre des abscisses")
plt.ylabel("titre des ordonnées")
plt.title("titre du graphe")
plt.legend() # pour afficher les légendes de type "label". Il ne prend pas d'argument.
plt.show() #pour ouvrir la fenêtre graphique
```

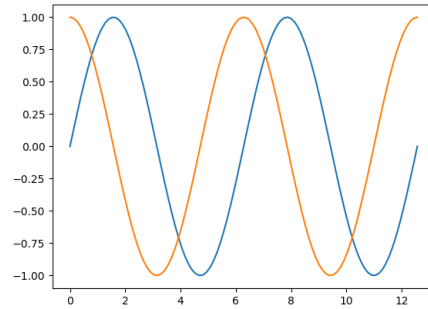
💡 Plus il y a de points dans un intervalle donné, i.e. plus le pas est petit, plus le tracé sera proche de la fonction voulue.

Si on veut représenter plusieurs courbes sur le même graphique, on fait des appels à la fonction `plot` les uns à la suite des autres, ou on lui donne les différentes courbes en arguments :

```

x=np.linspace(0,4*pi,100)
# 100 points répartis entre 0 et 4pi
y=np.sin(x)
z=np.cos(x)
plt.plot(x,y)
plt.plot(x,z)
#ou
plt.plot(x,y, x,z)
plt.show()

```



D'autres méthodes utiles du sous-module pyplot :

```

plt.clf() #efface la fenêtre graphique
plt.close() #pour fermer la fenêtre tout de suite quand le code est long
#et qu'on ne veut afficher que le dernier tracé.
plt.semilogy(x,y) # remplace plt.plot
# pour tracer avec une échelle logarithmique en axe des ordonnées.
plt.grid() # trace une grille

```

Pour aller plus loin : créer des sous-graphes

On définit la figure, qu'on appelle en général `fig`. Puis dans la figure on définit les sous-figures, selon les axes. On va utiliser la fonction `plot` qui prend en arguments des `array`.

```

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'une.
axs[i,j].plot(x,y) # trace dans le graphique (i,j),
#à répéter autant de fois que de sous-figures.
fig.suptitle("titre principal")
axs[0,0].set_title("titre graphe 1")
axs[0,1].set_title("titre graphe 2")

```

Références

- [CBCC16] Alexandre Casamayou-Boucau, Pascal Chauvin, and Guillaume Connan. *Programmation en Python pour les mathématiques - 2e éd.* Dunod, Paris, 2e édition edition, January 2016.
- [PL] Thierry Parmentelat and Arnaud Legout. Python 3 : des fondamentaux aux concepts avancés du langage FUN-MOOC.
- [Vig18] Vincent Vigon. *python proba stat*. Independently published, October 2018.