Introduction à Python

September 5, 2021

1 Introduction à la programmation en Python des méthodes mathématiques

Ceci est une très brève introduction au langage python (et son notebook) que nous allons utiliser en vue de coder des méthodes numériques, déterministes ou probabilistes.

1.0.1 IL NE S'AGIT PAS RÉELLEMENT D'APPRENDRE UN LANGAGE DE PROGRAMMATION AVEC SES SUBTILITÉS MAIS SIMPLEMENT D'ÊTRE CAPABLE DE L'UTILISER POUR PROGRAMMER DES MÉTHODES MATHÉMATIQUES.

Donc le but n'est pas de vous faire devenir un geek de Python… Pour info : Python est un véritable langage de programmation développé depuis 1990 dont l'intérêt dépasse le calcul scientifique. C'est un langage dit de haut niveau qui est multi-plateforme (Windows, macOS, Unix) et multi-paradigme (prog impérative, fonctionnelle, orientée objet). Il est distribué sous licence libre et gratuite, ce qui ne gâche rien :) Vous pourrez installer python et divers paquets supplémentaires par exemple via la distribution Anaconda.

1.1 Pour créer ou utiliser un notebook existant, ouvrir un terminal et taper :

jupyter notebook

Une fenêtre de navigateur s'ouvre et on peut créer un nouveau notebook en allant dans "Nouveau" –> "Python 3".

2 Remarques

Chaque cellule ("cell" ou "cellule" dans le menu du notebook) est soit une cellule de "code" (donc de Python), soit une cellule "markdown" où l'on peut insérer du texte (la cellule où est écrit ce paragraphe, par exemple). Pour ce qui concerne Python, chaque cellule peut être exécutée indépendamment ("Run" dans le menu ou "Shift-Entrée") mais on peut aussi découper un petit programme en plusieurs cellules pour le rendre plus lisible (une cellule peut, par exemple, contenir une fonction à réutiliser) : Python tient compte de TOUTES les cellules de code précédentes SI elles ont été exécutées.

3 Pour ce premier TP : vous devez recopier les cellules de code et regarder ce qu'elles donnent, tout en vous familiarisant avec la synthaxe Python. Des exercices sont faits pour vous permettre de vérifier que vous avez bien compris.

3.1 Les variables simples

Comme en mathématiques, tout langage de programmation manipule des variables dont voici quelques exemples :

[ ]:
```
A = 2
B = "Ceci est une chaine de caractères"
C = 3/4
D = 9,2333
E = 9.2333
```

[ ]:
```
print(A, B, C, D, E)
```

Il est important de connaître la nature des variables qu'on manipule, c'est-à-dire leur type.

[ ]:
```
print(type(A), type(B), type(C), type(D), type(E))
```

En Python 3, la variable C est de type "réel"; jusqu'en Python 2, c'était un entier, car l'opérateur / réalisait en fait la division euclidienne entre deux entiers. Notez aussi la synthaxe du "print" avec des (). En ce qui concerne D et E, attention à la différence entre la virgule et le point : avec une "," pour D, c'est un tuple, une variable composée, qui est en réalité le couple (9, 2333). Pour écrire des nombres à virgule, il faut donc utiliser la notation ".", par exemple : 2.127

3.2 Utilisation de fonctions usuelles

Essayons de calculer un cosinus :

[ ]:
```
a = 1.0
print(cos(a))
```

3.3 Vous devez bien lire les messages d'erreur qui doivent vous permettre de comprendre ce qui est problématique dans votre code.

En général, Python pointe la ligne qui pose problème et précise l'erreur du message d'erreur.

La fonction "cos" n'est pas définie par défaut, il faut donc importer une bibliothèque de fonctions. Pour le calcul scientifique, nous utiliserons essentiellement deux bibliothèques numpy pour les fonctions, variables scientifiques; et matplotlib pour les représentations graphiques.

Voici comment importer l'ensemble de toutes les fonctions d'une bibliothèque :

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

4 Remarque : la cellule ci-dessus devra figurer comme la première cellule de vos futurs notebooks.

[ ]:
```
E = 2.5
print(np.cos(E))
print(2*E)
print(E/3)
print(E**3)
print(np.e)
print(np.pi)
```

On utilise “∗” pour multiplier, “/” pour diviser et “∗∗” pour calculer la puissance; enfin, les majuscules ont leur importance : e est la base de l'exponentielle, qui est définie comme variable dans la bibliothèque !

[ ]:
```
print(np.e**(1.0/2))
print(np.exp(1.0/2))
```

5 Variables composites

5.1 Les tuples

Nous avons rencontré plus haut un premier type de variable composite, le tuple, qui est une généralisation du couple, qui est une collection de variables séparées par des virgules. Les parenthèses ne sont pas obligatoires, mais c'est sans doute préférable pour la lecture au moins au début :

[ ]:
```
A = (2, 9, "test", np.e, 3.0/2)
B = 3, 12
print(A, B)
```

On le voit, un tuple peut contenir n'importe quel type de variable, y compris un autre tuple :

[ ]:
```
C = (A, B)
print(C)
```

On peut accéder à un élément spécifique d'un tuple par son indice, sa position dans le tuple. Attention: en Python, par défaut tous les indices commencent à zéro !

[ ]:
```
print(A[1], "hop", A[0])
```

Si on peut accéder à n'importe quel élément d'un tuple, on ne peut le modifier. C'est d'ailleurs un intérêt de ce type de données, qui est ainsi protégée :

[ ]:
```
A[2] = 2.0
```

5.2 Les listes

Les listes ressemblent beaucoup aux tuples, mais elles sont modifiables. On les déclare avec des crochets.

[ ]:
```
AA = [2, "toto", A, 3.2]
print(AA)
```

[ ]:
```
print(AA[3])
```

[ ]:
```
AA[2] = 0
print(AA)
```

Une commande très utile pour les listes (par exemple, pour les suites récurrentes) : “append”. Elle sert à rajouter un élément à la fin de la liste :

[ ]:
```
AA.append("j'ajoute quelque chose")
print(AA)
AA.append(10.0)
print(AA)
```

5.2.1 Exercice

Ajouter un élément de votre choix à la liste ci-dessus et changer ‘toto’ en ‘titi’. Afficher chaque élément de la liste et demander leurs types.

5.3 Les vecteurs et matrices (tableaux Python)

Même pour l’Analyse, nous aurons besoin de manipuler des vecteurs et des matrices : ce sont des tableaux Python. Exemple : nous voulons représenter graphiquement la fonction cosinus sur $[0, \pi]$.

On commence par définir les abscisees :

[ ]:
```
N = 10
x = np.linspace(0, np.pi, N)
print(x)
```

L’opération linspace a créé un vecteur de taille N, dont le premier élément est 0.0, le dernier est π et les autres composantes sont réparties régulièrement sur l’intervalle $[0, \pi]$. Voyez plutôt :

[ ]:
```
i = 2
print(x[i+1] - x[i])
print(x[i+2] - x[i+1])
```

Si on tape :

[ ]:
```
y = np.cos(x)
print(y)
```

on voit que Python a pris le cosinus de chaque composante. Pour avoir la représentation graphique, il suffit alors de taper :

[ ]:
```
plt.plot(x, y)
```

Attention à l'échantillonnage : ici $N = 10$ et la courbe ne semble pas très "lisse". Pour la lisser, il suffit d'augmenter la valeur de N :

[ ]:
```
N = 200
x = np.linspace(0, np.pi, N)
y = np.cos(x)
```

[ ]:
```
plt.plot(x, y)
```

5.3.1 Exercice

Dessiner sur un même graphique les fonctions sinus et cosinus sur l'intervalle $[-1, 10]$ avec un échantillonage à 300 points.

Le cas de matrices quelconques sera vu plus en détails dans un TP spécifique : se rappeler qu'une matrice a un certain nombre de lignes et de colonnes et que ces informations doivent être clairement indiquées en Python.

[ ]:
```
M = np.matrix(np.zeros((3, 4)))
M[2,1] = 4.0
M[0,1] = 3.0

print(M)
```

[ ]:
```
N = np.matrix([[1, 2, 3], [3, 4, 5], [1, 2, 6], [3, 4, 7]])
print(N)
```

[ ]:
```
print(M*N)
print(M+N)
```

5.3.2 Exercice

Créer une matrice nulle 3×3 puis la transformer en matrice identité en affectant les valeurs 1 sur la diagonale. Créer ensuite la matrice

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 1 & 2 & 1 & 0 \\ 0 & 1 & 2 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

5.3.3 Exercice

Créer deux matrices 3×3 sans coefficient nul, les multiplier et vérifier le résultat en faisant le calcul à la main. Faire de même la multiplication d'une matrice 3×3 par un vecteur colonne.

5.4 Techniques de “slicing”

Il s’agit d’une technique très utile pour accéder à plusieurs éléments d’un vecteur (ou d’une matrice). Elle consiste à utiliser les deux points “:”

[ ]:
```
x=np.linspace(0.0,10.0,11)
print(x)
```

[ ]:
```
print(x[0:3])
print(x[3:5])
```

Attention ! la notation $[a : b]$ renvoit les éléments de la position a (inclus) jusqu’à la position b (exclu).

6 Structures de contrôle

Les différentes structures de contrôle gèrent le flux des instructions, c’est-à-dire la façon dont elles s’enchaînent en fonction de certains tests ou boucles.

6.1 Bloc de code

La notion de bloc de code correspond à une hiérarchisation d’un programme : il s’agit de regrouper certaines instructions qui “vont ensemble”, notamment quand elles sont exécutées si une certaine condition est remplie.

bloc principal if condition : … (bloc 2) … if condition2 : … (bloc 3) … fin du bloc 2 fin du bloc principal

Dans les plupart des langages, les blocs de code sont délimités par des accolades (langage C, PHP, etc.). En Python, il n’y a pas de délimiteur spécifique, c’est l’indentation du code qui détermine les blocs ! C’est à la fois un avantage et un défaut : cela force à avoir un code proprement indenté, sans quoi il ne peut s’exécuter correctement. Mais il faut en prendre l’habitude.

6.2 Boucle for

Les boucles sont très utilisées en programmation, ne serait-ce que pour passer en revue des entiers. On utilise pour cela la fonction range :

[ ]:
```
for i in range(4):
    print(i)
```

[ ]:
```
for i in range(4):
print(i)
```

On aura noté que Python est cohérent sur le fait que les indices démarrent à zéro. range(a) renvoit une liste de “a” éléments, commençant par zéro et finissant donc par… “a-1”.

On peut utiliser cette fonction de façon plus complexe, pour voir les utilisations diverses, utiliser le point d’interrogation qui appelle l’aide en ligne :

[ ]:
```
range?
```

On peut utiliser des boucles pour faire des sommes (par exemple).

[ ]:
```
m=0
for i in range(15):
    m=m+i
print(m)
```

Et si le print est indenté ?

[ ]:
```
m=0
for i in range(15):
    m=m+i
    print(m)
```

6.2.1 Exercice

Tester le sens de range(3, 5) et faire la somme des entiers de 32 à 38.

6.2.2 Exercice

Générer une liste L dont le premier élément est le réel e et les suivants sont les entiers de 7 à 11.

6.2.3 Exercice

Que donne(ra) le programme suivant (quand vous l'aurez corrigé) :

[ ]:
```
m=0
for i in L:
m=m+i
print(m)
```

La syntaxe spécifique se comprend par le fait que la boucle for permet d'itérer n'importe quelle liste (et plus généralement tous les objets "itérables"). Or, range fournit une liste.

[ ]:
```
R = range(3)
print(R)
print(R[0], R[1], R[2])
print(type(R))
```

On peut aussi faire des boucles sur une liste :

[ ]:
```
A = [2, 9, "test", np.e, 3.0/2]

for entree in A:
    print(type(entree))
    print(entree)
```

Voici un exemple de code calculant les dix premiers termes de la suite de Fibonacci.

[ ]:
```
a, b = 0, 1
# initialisation

for i in range(10):
    # debut du bloc de code
    c = a + b
    print(c)
    a, b = b, c
    # fin du bloc de code
```

On notera plusieurs choses :

- l'utilisation de commentaires par le signe dièse # ;
- l'affectation multiple sur une même ligne (très pratique !) ;
- on a ajouté ici des commentaires pour marquer le début et la fin du bloc mais c'est inutile.

6.2.4 Exercice

Construire une liste de 5 éléments contenant les termes $\exp(i)$ pour i allant de 1 à 5.

6.2.5 Exercice

Construire une liste de 7 éléments, le premier étant π, chacun des suivants étant le double du précédent.

6.3 Condition if

Il s'agit d'un test qui déclenche l'exécution d'un bloc si une condition est remplie.

[ ]:
```
a = 17
# initialisation

if a > 10 : # test
    # début du bloc
    print("a est plus grand que dix")
    # fin du bloc
```

[ ]:
```
a = 8
# initialisation

if a > 10 : # test
    # début du bloc
    print("a est plus grand que dix")
    # fin du bloc
```

On peut utiliser une syntaxe plus élaborée en disant quoi faire si le test échoue.

[ ]:
```
a = 12

if a > 10 :
    print("a est plus grand que dix")
elif a == 10:
    print("a est égal à dix")
else:
    print("a est plus petit que dix")
```

[ ]:
```
a = 2

if a > 10 :
    print("a est plus grand que dix")
elif a == 10:
    print("a est égal à dix")
else:
    print("a est plus petit que dix")
```

[ ]:
```
a = 10

if a > 10 :
    print("a est plus grand que dix")
elif a == 10:
    print("a est égal à dix")
else:
    print("a est plus petit que dix")
```

elif est l'équivalent de "else if ..." ; on pourrait en enchainer plusieurs pour traiter d'autres sous-cas ; enfin le bloc qui suit else est traité si aucun test n'a été validé en amont.

On fera attention au test d'égalité qui utilise un double signe égal ==, pour qu'il n'y ait pas de confusion avec l'affectation de valeur ($a = b$).

6.3.1 Exercice

En utilisant des tests, faire une boucle de 0 à 100 qui ne fait que la somme des entiers de 35 à 56.

6.4 Condition while

Il s'agit ici d'exécuter une commande ou un bloc de code tant que (while) une certaine condition est remplie.

[ ]:
```
n = 8
#initialisation du nombre de départ

fact = 1
#intialisation de la factorielle

m = 0
```

```
#initialisation variable auxilliaire

while m != n: # test : si m différent de n
    m = m + 1
    # alors on ajoute 1
    fact = fact * m
    # et on multiplie fact par m

    print(m, fact)

print("Factorielle de ", n, " = ", fact)
```

6.4.1 Exercice

En utilisant une boucle “while”, faire la somme des nombres de 1 à 100 et vérifier la formule bien connue.

6.4.2 Exercice

Trouver le plus grand entier n tel que $n!$ est plus petit que 123456.

7 Fonctions

Nous avons déjà vu comment utiliser des fonctions, voyons maintenant comment en créer. L'intérêt est d'avoir nos propres fonctions, qui seront donc des bouts de code réutilisables. Supposons par exemple que nous souhaitions calculer la factorielle de n'importe quel nombre entier à partir du code ci-dessus. Nous allons donc définir une fonction factorielle à l'aide du mot-clef def:

[ ]:
```
def factorielle(n):
    fact, m = 1, 0

    while m != n :
        m = m+1
        fact = fact*m

    return fact
```

[ ]:
```
print(factorielle(10))
```

On passe donc l'entier en paramètre et il en ressort un entier, la factorielle. Il faut cependant être prudent: si on n'envoie pas un entier à notre fonction, on risque des ennuis ! On peut donc améliorer notre fonction en testant le type de n dès le début du code.

La fonction suivante calcule le volume d'un pavé dont on donne les dimensions.

[ ]:
```
def volume(largeur, longueur, hauteur):
    return largeur*longueur*hauteur
```

[ ]:
```
print(volume(3, 5, 4))
```

En plus du type, il faut donc se souvenir de l'ordre des paramètres pour l'appel d'une fonction. Ceci étant, on peut éventuellement utiliser les étiquettes pour éviter les ennuis:

[ ]:
```
print(volume (largeur=3, longueur=2, hauteur=6))
```

On a utilisé ici deux fonctions qui renvoient quelque chose, à l'aide du mot-clef return. Mais une fonction peut aussi ne rien renvoyer (en général, on préfère parler de méthode dans ce cas), comme par exemple:

[ ]:
```
a = [0, 0, 1, -1, 0, 0, 2, 3]

def modify(a):
    a[4] = 100
```

[ ]:
```
print(a[4])

modify(a)

print(a[4])
print(a)
```

7.0.1 Exercices

1. Écrire une fonction sommeDesCarres(n) qui donne la somme des carrés des entiers de 1 à n puis une fonction sommeDesPuissances(k, n) qui donne la somme des puissances k des entiers de 1 à n.
2. Écrire une fonction max2(a,b) qui retourne le plus grand des nombres a et b, puis une fonction max3(a,b,c) qui retourne le plus grand des nombres a, b et c. Écrire enfin une fonction max(a) qui retourne le plus grand des nombres d'une liste a.
3. Programmer une fonction prenant en argument un réel u_0, un entier $N > 1$ et renvoyant le terme u_N de la suite définie par récurrence à partir de u_0 via :

$$u_{n+1} = \frac{1}{1 + u_n^2}$$

4. Programmer une fonction de deux entiers naturels n et p avec $n > p$ qui renvoie le plus grand entier q tel que

$$p \times q \leq n$$

en utilisant une boucle "while". En déduire une fonction qui fait la division euclidienne d'un entier n par un entier p en renvoyant le quotient q et le reste r.

8 Conclusion

Nous n'avons évidemment fait que survoler (et de très haut) les bases du langage Python. Voici un résumé de l'essentiel à retenir pour le moment :

- types de variables simples : entier, flottant, booléen;
- types de variables composées : tuples, listes, chaines de caractères ;
- blocs de code : c'est l'indentation qui détermine le niveau du bloc ;
- structures de contrôle du flux : for, if-elif-else, while ;
- fonctions : on utilise les mots-clef def et return.