Introduction à Python

Cours de modélisation numérique

20 septembre 2017

1 Introduction

Le but de cette première séance est de se familiariser avec le langage de programmation Python, qui sera régulièrement utilisé dans le cadre des exercices.

Nous allons commencer en présentant les types de base, les structures de contrôle et quelques structures de données. Ce document est en partie basé sur le tutoriel Python proposé sur le site officiel [1].

2 Les bases

Python est un langage de script, cela veut dire qu'il n'est pas nécessaire de compiler les codes, comme pour le Java ou le C++. Python existe pour tous les systèmes d'exploitation donc vous trouverez une version convenable pour votre machine.

2.1 Installation et IDE

Dans beaucoup des systèmes d'exploitation actuels, il existe une version de Python pré-installée. Si ce n'est pas le cas, les différentes versions sont téléchargeables sur le site officiel [2]. Nous conseillons les versions de pythons supérieures à la version 3 (les corrections des exercices ne seront pas toutes compatibles avec Python2.x). Comme avec tout langage de programmation, aucun IDE n'est requis pour programmer en python. Il faut simplement pouvoir editer un fichier et l'exécuter dans une console en tapant `python <nom_du_fichier>`. Néanmoins, nous pouvons mentionner les environnements suivants, qui facilitent la tâche et possèdent tous un interpréteur python :

— Pour tous les systèmes : python IDLE, léger et très minimaliste.
— Sous Windows : PyScripter.
— Pour les systèmes unix : PyCharm, WingIDE, ...

Il est à noter que des distributions dédiées à la programmation scientifique, telles que Enthought Canopy, Pythonxy ou Spyder, regroupent un grand nombres de librairies utiles (production de graphiques, calcul matriciel, etc..).

2.2 L'interpréteur

En Python, il est possible de tester interactivement les commandes grâce à l'interpréteur. Pour le lancer, il suffit de taper `python` à la console. Quand le signe `>>>` apparaît, on est près à l'utiliser.

Il est possible de taper des commandes sur une ou plusieurs lignes. Chaque commande est évaluée et donne un résultat immédiatement. Testez les commandes suivantes en tapant `enter` après :

```
>>> # commentaire
>>> 2+2
```

Si nous voulons insérer plusieurs lignes à évaluer, il faut appuyer sur `shift+enter` pour forcer un saut de ligne sans faire l'évaluation de la précédente. Il y a trois points qui apparaissent pour nous signaler qu'il y a des opérations en attente. Essayez :

1. `http://docs.python.org/tutorial/`
2. `https://www.python.org/download/releases/`

```
>>> # shift + enter après ça
... 2+2 # enter après ça
```

Par la suite nous vous demandons de **taper toutes les commandes présentées en exemple à la ligne de commande de l'interpréteur**.

2.3 Scripts exécutables

Si le code que nous voulons exécuter dépasse un certain nombre de lignes, les retaper sur l'interpréteur devient très long. Il est alors possible de sauvegarder toutes ces lignes dans un script que nous exécutons par la suite. Le nom du script finira par `.py`. Pour l'exécuter, nous taperons tout simplement

```
python mon_script.py
```

Il est à noter que l'argument `-i` avant le nom du fichier permet de ne pas fermer l'interpréteur après exécution du script, ce qui est très pratique notamment pour le déboggage.

3 Les types de base

Dans cette section nous utiliserons les types de base Python.

3.1 Les nombres

Les nombres et les opérations basiques (`+`,`-`,`*` et `/`) marchent exactement comme pour les autres langages. L'interpréteur peut être utilisé comme une simple calculatrice :

```
>>> (50-5*6)/4
```

Il est possible de sauvegarder des valeurs dans des variables pour les utiliser après :

```
>>> x = 2*3-16+121
>>> y = z = u = v = 123456
```

Si nous essayons d'accéder à une variable qui n'a pas reçu de valeur, l'interpréteur nous signalera l'erreur :

```
>>> x = 2*n+2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Il faut faire la distinction entre les entiers et les nombres en virgule flottante :

```
>>> 80//7 # division entiere
>>> 80/7.0 # division en virgule flottante (peut aussi s'ecrire 80/7 en Python3)
```

On peut toujours convertir les nombre d'un type à un autre grâce aux fonctions `float()`, `int()` et `long()`.

3.2 Les chaînes de caractères

Les chaînes de caractères s'écrivent entre des apostrophes (') ou des guillemets (") :

```
>>> a = 'une chaine'
>>> b = "une autre chaine"
```

Si jamais nous avons besoin d'avoir une apostrophe ou une guillemet dans la chaîne, nous pouvons soit choisir d'utiliser l'opposé comme délimiteur, soit l'échapper comme dans l'exemple suivant :

```
>>> c = "c'est une chaine"
>>> d = 'c\'est une chaine'
```

Pour accéder à un caractère de la chaîne, il faut donner l'indice qui nous intéresse. On peut donner un ensemble d'indices séparés par `:`. Il faut noter que les indices commencent à 0.

```
>>> w = "Hello"
>>> w[0] # retourne H
>>> w[2] # retourne l
>>> w[1:4] # retourne 'ell'
>>> w[2:] # retourne 'llo'
>>> w[:2] # retourn 'He'
```

Il est possible de concaténer des chaînes (`+`) et de les répéter (`*`) :

```
>>> a = "vive "
>>> b = "la chaine "
>>> a*5
>>> b+a
```

Les chaînes sont immuables en Python, cela veut dire que nous ne pouvons pas les modifier. Si nous voulons par exemple changer un caractère par un autre, il faut recréer une nouvelle chaîne.

Finalement, pour trouver la longueur d'une chaîne, nous pouvons nous servir de la fonction `len()`.

3.3 Les listes

Une liste est un groupe de données hétérogènes. Pour déclarer une liste il suffit d'écrire les composants séparés par des virgules entre des crochets (`[]`) :

```
>>> l = ["ceci", 'est une ', 1, 2]
```

La liste étant un élément central de Python, il est important de bien comprendre son fonctionnement. Pour accéder aux éléments, nous utilisons la même convention que pour les chaînes. Contrairement aux chaînes, il est possible de modifier les contenus d'une liste :

```
>>> p = [2,3,5,7,9]
>>> p[2]
>>> p[3:]
>>> p[3] = 11
>>>
>>> a = [1,2,3,4,5,6]
>>> a.append(7) # rajouter 7 a la fin de a
>>> a.remove(2) # eliminer l'element 2
>>> b = [3,4,5]
>>> a.extend(b) # concatener a et b
>>> a.reverse() # inverser a
>>> a.count(3) # compter combien de fois 3 apparait
>>> a.sort() # trier la liste
```

Nous pouvons avoir des listes imbriquées comme dans l'exemple suivant :

```
>>> l1 = [1,2,3]
>>> l2 = [l1,'aaa']
```

Comme pour les chaînes, pour connaître la longueur d'une liste, nous utilisons la fonction `len()`.

4 Les structures de contrôle

Afin de créer des codes qui sont plus compliqués que la simple somme de deux nombres, en Python, nous possédons toutes les structures de contrôle habituelles, telles que les conditionnelles et les boucles.

4.1 If

Voici un exemple qui utilise la conditionnelle `if` :

```
>>> x = int(raw_input('Entrez un nombre: ')) # demande une entree
>>> if x%2==1:
...     print("impair")
... elif x%2==0:
...     print("pair")
... else:
...     print("impossible")
```

Il faut noter que `elif` est une contraction de `else: if` pour éviter des problèmes d'indentation. les parties `elif` et `else` sont facultatives. Si vous oubliez de taper les deux points après les instructions `if`, `elif` et `else`, vous ne pourrez pas écrire sur la ligne suivante, cela évaluera les instructions et comme elles seront pas complètes, vous aurez une erreur.

Dans l'exemple vous noterez l'absence de délimiteurs de blocs. En effet, en Python, **la délimitation des bloques se fait par leur indentation**. Ceci deviendra plus clair dans les sections suivantes.

4.2 while

La boucle `while` s'exécute tant que ça condition est vraie :

```
>>> s = int(raw_input('Entrez un nombre d\'iterations: '))
>>> while s>0:
...     print("iteration ", s)
...     s -= 1
```

Ce petit code va donc itérer s fois.

4.3 for

La boucle `for` est légèrement différente de ce que nous avons l'habitude d'utiliser en C et C++. La boucle va itérer sur les éléments d'une séquence (par exemple une liste ou une chaîne).

```
>>> myListe = [1,2,3,4,5]
>>> for el in myListe:
...     print("element: ", el)
```

Néanmoins, si nous devons itérer sur une suite de nombres, nous pouvons utiliser la fonction `range()`. Observez les exemples suivants :

```
 >>> range(5) # generer une suite entre 0 et 5
[0, 1, 2, 3, 4]
>>> range(5,10) # generer une suite entre 5 et 10
[5, 6, 7, 8, 9]
>>> range(0,10,3) # suiten entre 0 et 10 avec pas de 3
[0, 3, 6, 9]
```

La borne inférieure appartient toujours à la suite générée, tandis que la borne supérieure pas.

Pour mettre le tout ensemble, comprenez le code suivant :

```
>>> a = ['une','liste','avec','des','strings']
>>> for i in range(len(a)):
...     print(i, a[i])
```

Comme dans d'autres langages on peut se servir des instructions suivantes :

— `break` arrête la boucle à l'itération courante

— `continue` ignore tout le code qui se trouve après l'instruction et revient au début de la boucle

En plus il y a une instruction `else` possible dans une boucle. Cette instruction est exécuté uniquement quand `for` a bouclé sur tous les éléments ou quand la condition d'une boucle `while` devient fausse :

```
>>> for n in range(2,10):
...     for x in range(2,n):
...         if n%x == 0:
...            break
...     else:
...         print(n, " est premier")
```

4.4 Définition de fonctions

Le prochain pas est de définir des fonctions que nous allons réutiliser par la suite. La définition d'une fonction utilise le mot clé `def`. Voici un exemple d'une fonction qui calcule la factorielle d'un nombre n :

```
>>> def factorial(n):
...     f = 1
...     while (n > 0):
...         f = f * n
...         n = n - 1
...     return f
```

L'instruction `return` retourne une la ou les valeurs qui viennent après. Il faut noter qu'une fonction peut ne pas avoir d'instruction `return` ; elle retourne alors `None` par défaut.

La même fonction dans sa version récursive :

```
def factorial(n):
     """ Recursive implementation of factorial """
     if n == 0:
         return 1
     else:
         return n * factorial(n-1)
```

Dans ce deuxième exemple, la première ligne est une chaîne délimité par trois guillemets. Ceci s'appelle une chaîne de documentation. Il est toujours conseillé de l'écrire pour spécifier brièvement la ou les actions de la fonction.

5 Les structures de données

Nous présentons ici des structures de données avancées présentes par défaut en Python.

5.1 Tuples

Un *tuple* est un ensemble de valeurs séparées par des virgules.

```
>>> t = 1,3,"string",23.9 # packing
>>> t
(1, 3, 'string', 23.899999999999999)
>>> a,b,c,d = t # unpacking
```

Le fait de créer un tuple est connu sous le nom de packing et l'opération inverse affecte chaque valeur d'un tuple à une variable. Il faut noter que pendant l'unpacking, il doit y avoir autant de variables que d'éléments dans le tuple.

5.2 Ensembles

Les ensembles sont des collections ordonnées de valeurs uniques. Voici des exemples d'opérations qu'on peut appliquer sur les ensembles :

```
>>> l = [1,22,3,5,22,1]
>>> s = set(l)
>>> s
set([1, 3, 5, 22])
>>> m1 = set('python')
>>> m2 = set('ruby')
>>> m2-m1 # difference d'ensembles
set(['r', 'b', 'u'])
>>> m2 | m1 # union
set(['b', 'h', 'o', 'n', 'p', 'r', 'u', 't', 'y'])
>>> m2 ^ m1 # difference symetrique
set(['p', 'b', 'u', 't', 'h', 'r', 'o', 'n'])
>>> m2 & m1 # intersection
set(['y'])
```

5.3 Dictionnaires

Les dictionnaires sont des séquences qui sont adressées par des *clés* plutôt que par des indices. A chaque *clé* on fait correspondre une *valeur*. Voici un exemple d'utilisation des dictionnaires et les opérations les plus courantes.

```
>>> d = {"un":1,"deux":2,"trois":3}
>>> d.keys()
['un', 'trois', 'deux']
>>> d['un']
1
>>> 'un' in d
True
>>> 'cinq' in d
False
```