Cours : Prise en main de Python

Le langage Python

Python est le langage de programmation open source le plus employé par les informaticiens. Il est multiplateforme et dispose de fonctions évoluées permettant de gagner du temps dans le développement du code. De plus, le caractère open source conduit au développement et au partage de nombreuses bibliothèques dédiées (traitement, d'images, de sons, …).

Il présente de nombreuses qualités :

- Sa syntaxe est très simple et concise : « on code ce que l'on pense », donc facile à apprendre, proche du « langage algorithmique » ;
- Moderne : relativement récent, c'est actuellement le langage de programmation le plus largement utilisé. Il est très largement répandu dans l'industrie, l'enseignement et la recherche, notamment pour ses applications scientifiques ;
- Gratuit. Une large communauté participe à son développement.
- C'est un langage interprété, ce qui permet son usage pour écrire des scripts pour Internet, des applications portables.

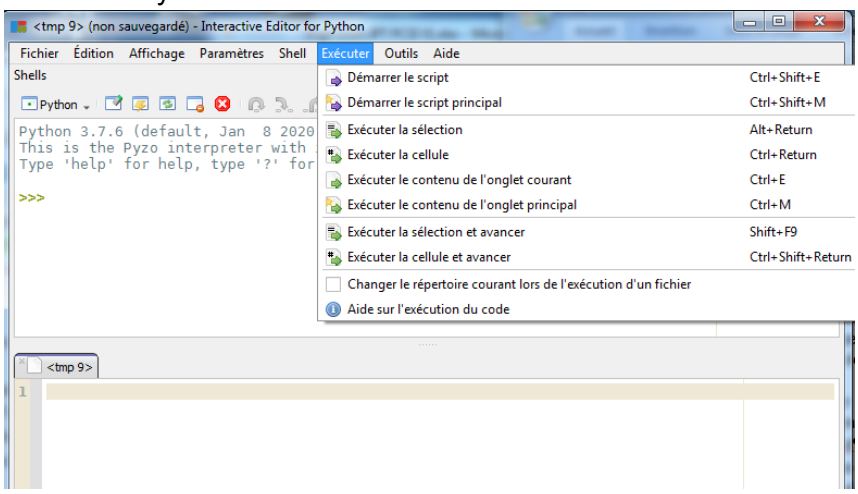
Il présente bien un défaut, celui de ses qualités. C'est un langage interprété (comme JavaScript), et non compilé comme la plupart des langages de programmation. Cela permet l'écriture de scripts web et d'applications portables, mais a pour contrepartie que son exécution est plus lente qu'un langage compilé (qui convertit le programme en langage machine avant de l'exécuter).

Environnement de développement

Pour utiliser Python, il est nécessaire d'installer un environnement de développement qui permettra d'éditer du code, et d'en lancer l'exécution. On peut nommer :

- Pyzo
- Anaconda
- Spyder
- …

Une capture d'écran de Pyzo :



Partie haute : la console. C'est à ce niveau que s'affiche l'exécution du code. On peut également taper des commandes qui seront aussitôt exécutées. Ceci est intéressant pour des tests simples.

Partie basse : l'éditeur. C'est à ce niveau que l'on tape les lignes de code qu'il faudra penser à sauvegarder et dont on peut lancer l'exécution via le menu Exécuter.

Fenêtre « Console » ou « shell » en anglais

Le prompt >>> attend une commande de l'utilisateur.

1) Mode calculatrice

```
>>> 1+3
4
>>> 3**2
9
>>> abs(-2)
2
>>> 1e3
1000.0
>>> len('abcd')
4
```

2) Division euclidienne ou division entière

```
>>> 7//2 # quotient de la division entière
3
>>> 7%2 # reste de la division entière
1
```

3) Typage

Python reconnait la nature de la grandeur traitée et lui affecte automatiquement un type.

```
>>> type(3)
<class 'int'>
>>> type('abcd')
<class 'str'>
>>> type(3.14)
<class 'float'>
```

Les types primitifs sont :

- bool : booléen (True ou False)
- int : entier
- float : nombre flottant
- str : chaîne de caractères (string)

Une variable sans valeur est définie par : None (équivalent à Null dans d'autres langages).

4) Conversion ou transtypage

Si la conversion est possible, il est possible de changer le type d'une grandeur.

```
>>> float(7)
7.0
```

```
>>> int(3.14)
3
>>> str(75009)
'75009'
>>> int('12')
12
>>> int('IPT')
Traceback (most recent call last):
File "<console>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'IPT'
```

5) Variables

Sous Python, les noms de variables doivent en outre obéir à quelques règles simples. Un nom de variable est une séquence de lettres (a → z , A → Z) et de chiffres (0 → 9), qui doit toujours commencer par une lettre. La casse est significative (les caractères majuscules et minuscules sont distingués).

```
>>> var1=123
>>> 1var = 321
File "<console>", line 1
1var = 321
^
    SyntaxError: invalid syntax
```

Remarque : Ne pas utiliser pour nommer des variables, des instructions du langage Python (for, else, if, def, while, …).

6) Affectation

permet d'indiquer à l'interpréteur que cette partie du code n'est pas à exécuter. Ceci indique un commentaire, bien utile pour faciliter la compréhension d'un script Python.

```
>>> var1, var2 = 12, 75009 # Affectation multiple

>>> var1 = var2 = var3 = 75009 # Affectation simultanée

>>> var1, var2 = var2, var1 # Permutation sans faire appel à une variable tampon
```

7) Opérateurs de comparaison

```
>>> True or False # Opérateur OU
True
>>> True and False # Opérateur ET
False
>>> not (True) # Opérateur NON
False
>>> 12 == 15.0 # Test d'égalité (attention aux DEUX signes ==)
False
>>> 12 != 15.0 # Renvoie la négation du test d'égalité
```

```
True
>>> 12<=75 # Opérateur de comparaison
True
>>> 'D' in 'Decour' # Test d'appartenance
True
```

8) Chaîne de caractères

```
>>> var ='Jacques_'+'DECOUR'
>>> var
'Jacques_DECOUR'
>>> var[0] # premier caractère (numérotation qui débute à 0)
'J'
>>> var[-1] # dernier caractère
'R'
>>> var[len(var)-1] # dernier caractère
'R'
>>> var[2:5] # extraire une sous-chaîne, premier inclus, dernier exclu
'cqu'
>>> 'Jacques' > 'Decour' # comparaison selon les règles lexicographiques
True
```

9) Aide en ligne

```
>>> help(len)
Help on built-in function len in module builtins:
len(obj, /)
Return the number of items in a container.
>>> len ?
Return the number of items in a container.
```

Boucle « for »

1) L'itérateur range

```
>>> list(range(0,5)) # 0 valeur par défaut pour initier la liste
[0, 1, 2, 3, 4]
>>> list(range(5)) # itère et renvoie 5 éléments
[0, 1, 2, 3, 4]
>>> list(range(0, 16, 2)) # par pas de 2, 16 exclu
[0, 2, 4, 6, 8, 10, 12, 14]
>>> list(range(16, 0, -2)) # par pas négatif de valeur -2, 0 exclu
[16, 14, 12, 10, 8, 6, 4, 2]
>>> list(range(16, 0)) # renvoie aucun élément
[]
```

2) Parcours par valeur

Notre premier script ou programme Python ! A écrire dans la partie éditeur de code puis à exécuter.

Script Python :

```
var ='Jacques'

for carac in var: # création d'une boucle
        print(carac) # bloc d'instructions qui seront répétées
```

Résultat de l'exécution :

```
>>> (executing file "prise en main.py")
J
a
c
q
u
e
s
```

L'instruction **for** conduit à créer un bloc d'instructions qui seront répétées. Ce bloc d'instructions est indenté d'une tabulation.

Attention :

- L'indentation est structurelle en Python.
- Ne pas oublier les : en fin de ligne initiant un bloc d'instructions.

Script :

```
var ='Jacques'

for carac in var:
        print(var)
print(carac) # non indenté donc hors de la boucle
```

Résultat de l'exécution :

```
>>> (executing file "prise en main.py")
Jacques
Jacques
Jacques
Jacques
Jacques
Jacques
Jacques
s
```

3) Parcours par indice

Script Python :

```
var ='Jacques'
n = len(var)
print('var de longueur : ',n) # on sépare par une virgule
for k in range(n):
        print(k, var[k]) # parcourt par indice, ici k qui va varier de 0 à (n-1)
```

Résultat de l'exécution :

```
>>> (executing file "prise en main.py")
var de longueur : 7
0 J
1 a
2 c
3 q
4 u
5 e
6 s
```

4) Exercice : parcours en ordre inversé

   Ecrire les lignes de code, permettant d'afficher ligne à ligne chaque caractère d'une chaine de caractères pris dans l'ordre inverse.
   Solution 1 :

   Solution 2 :

Fonctions Python

Lorsqu'une tâche doit être réalisée plusieurs fois par un programme avec seulement des paramètres différents, on peut l'isoler au sein d'une fonction. Cette approche est également intéressante si la personne qui définit la fonction est différente de celle qui l'utilise. Par exemple, nous avons déjà utilisé la fonction `print()` qui avait été définie par quelqu'un d'autre.

1) Syntaxe

```
def nom_fonction(liste de paramètres):
      bloc d'instructions
```

Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots-clés réservés du langage, et à la condition de n'utiliser aucun caractère spécial ou accentué (le caractère souligné « _ » est permis).

2) Corps de la fonction

La ligne contenant l'instruction **def** se termine obligatoirement par un deux-points :, qui introduisent un bloc d'instructions qui est précisé grâce à l'indentation. Ce bloc d'instructions constitue le **corps de la fonction**.

3) Appel de la fonction

Script Python :

```
def affiche_carac(var): # ici un seul paramètre
        for carac in var : # une indentation
                print(carac) # une deuxième indentation

affiche_carac('Decour') # appel à la fonction
```

Résultat de l'exécution :

```
>>> (executing file "prise en main.py")
D
e
c
o
u
r
```

Dans le script, on écrit la fonction **affiche_carac**, que l'on peut ensuite utiliser si l'on fait appel à elle. Au niveau de la fonction, le paramètre **var** prendra la valeur avec laquelle on appelle la fonction, à savoir ici la chaine de caractères 'Decour'.

4) Fonction sans paramètre

Script Python :

```
def Bonjour():
        for k in range(3):
                print (' '*k, 'Bonjour')

Bonjour() # appel à la fonction
```

```
>>> (executing file "prise en main.py")
 Bonjour
   Bonjour
     Bonjour
```

5) Paramètre par défaut

Script Python :

```
def exponentiation(x, n=2): # deux paramètres, dont un avec valeur par défaut
        for k in range(0, x, 2) : # le zéro est nécessaire
                print(k**n)

exponentiation(5) # x prendra la valeur 5, et n la valeur 2
```

```
>>> (executing file "prise en main.py")
0
4
16
```

6) L'instruction return

Pour pouvoir utiliser le résultat calculé par une fonction après l'avoir quittée, il est nécessaire que cette fonction renvoie le résultat, ceci est réalisé via l'instruction **return**.

Script Python :

```
def cube(x):
        return x**3

resu = cube(5) # on récupère le résultat de l'exécution de la fonction cube
print ('cube(5) = ', resu)
```

```
>>> (executing lines 1 to 5 of "prise en main.py")
cube(5) = 125
```

7) Variables locale et globale

A chaque fois que nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des **variables locales** à la fonction. Une variable locale peut avoir le même nom qu'une variable de l'espace de noms global mais elle reste néanmoins indépendante.
Les contenus des variables locales sont stockés dans l'espace de noms local qui est inaccessible depuis l'extérieur de la fonction.
Les variables définies à l'extérieur d'une fonction sont des variables globales. Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier.
Script Python avec test1 :

```
b = 5

def test1(a):
        print(a+b) # b n'est pas définie au niveau local,
        # la fonction utilise la valeur de b définie au niveau global

test1(7)
print(b)
```

```
>>> (executing lines 1 to 8 of "prise en main.py")
12
5
```

Script Python avec test2 :

```
b = 5

def test2(a):
        b = 20
```

```
        print(a+b) # b est définie au niveau local,
        # la fonction utilise la valeur de b définie au niveau local

test2(7) # affiche 7 +20 = 27
print(b) # affiche 5

>>> (executing lines 1 to 9 of "prise en main.py")
27
5
```

Puis avec un caractère global :

```
b = 5

def test3(a):
        global b
        print(a+b) # la fct utilise la valeur définie au niveau global
        b = 20 # la fct modifie la valeur de la variable globale

test3(7) # affiche 7 + 5 = 12
print(b) # affiche 20
```

8) Organisation du code

```
val_Pi = 3.1415926 # non défini dasn le Python de base

def cube(x):
        return x**3

def volume_sphere(r):
        return 4/3* val_Pi * cube(r)

R = 5 # unité à définir
print('Volume de la sphère : ', volume_sphere(R))
```

A bien y regarder, ce programme comporte deux parties :

- les deux fonctions **cube**() et **volume_sphere**()
- le **corps principal du programme**.

Dans le corps principal du programme, il y a un appel de la fonction **volume_sphere**().
A l'intérieur de la fonction **volume_sphere**(), il y a un appel de la fonction **cube**().
Notez bien que les deux parties du programme ont été disposées dans un certain ordre :

- d'abord la définition des fonctions,
- et ensuite le corps principal du programme.

Cette disposition est nécessaire, parce que l'interpréteur exécute les lignes d'instructions du programme l'une après l'autre, dans l'ordre où elles apparaissent dans le code source. Dans le script, la définition des fonctions doit donc précéder leur utilisation.

Exercices

On aura besoin d'un test qui s'écrit :

```
        if condition :
```

Bloc d'instructions # exécutées si la condition est vraie

1) Ordre lexicographique 1

   Ecrire une fonction **lexico1**, qui à partir d'une chaîne de caractères renvoie un booléen indiquant si l'ordre lexicographique est respecté en parcourant dans le sens des indices croissants les éléments de la chaîne de caractères.

Solution :

2) Ordre lexicographique 2

   Ecrire une fonction **lexico2**, qui à partir d'une chaîne de caractères renvoie le nombre fois où l'ordre lexicographique n'est pas respecté en parcourant dans le sens des indices croissants les éléments de la chaîne de caractères.

Solution :

3) Ordre lexicographique 3

   Ecrire une fonction **lexico3**, qui à partir d'une chaîne de caractères renvoie la chaîne de caractères dans laquelle on a supprimé le caractère qui ne respecte pas l'ordre lexicographique en parcourant dans le sens des indices croissants les éléments de la chaîne de caractères et en supposant que le non respect de l'ordre se présente une seule fois.

Solution :

4) Melange

   Ecrire une fonction **melange**, qui à partir de deux chaînes de caractères de même longueur renvoie la chaîne de caractères constituée du premier élément de la première chaine, puis le premier élément de la deuxième chaine, puis le second élément de la première chaine, puis le second élément de la deuxième chaine, puis ….