S1-basespython

September 4, 2019

1 Langage Python

logo

> Python est un langage de programmation, développé en **1989** par **Guido Von Rossum**, à l'université d'Amsterdam. Il est simple d'usage, concis, libre et gratuit, multiplate-forme, largement répandu, riche de bibliothèques adaptées et bénéficiant d'une vaste communauté d'auteurs dans le monde éducatif.

Il permet d'interagir avec la machine à l'aide d'un programme appelé *interprète Python*. On peut l'utiliser de deux façons différentes. La première méthode consiste en un *dialogue avec l'interprète*. c'est le mode **interactif**.

Le second consiste à *écrire un programme ou code source dans un fichier*, c'est à dire une suite d'instructions, puis à le faire exécuter par l'interprète Python. C'est le **mode programme**.

On peut utiliser différents environnements de développement : Iddle, Pyzo, Spyder, Pyscripter…, qui permettent d'utiliser les deux modes simultanément.

Mais dans un premier temps, vous allez le faire directement dans ce **notebook**, au fur et à mesure du cours.

1.1 Sommaire

Section **??**

Section **??**

Section **??**

Section **??**

Section **??**

Section **??**

1.2 1. Les opérateurs

On peut utilser l'interpréteur Python comme une simple machine à écrire :

[ ]:
```
5+3
```

[ ]:
```
7-8
```

[ ]:
```
16/5
```

Maintenant certaines opérations un peu moins classiques sont à connaître :

[ ]:
```
5**3
```

[ ]:
```
16//5
```

[ ]:
```
16%5
```

*Exercice*1 : Précisez ce que font les opérateurs :
**
//
%

Opérateurs de comparaison - x == y : test d'égalité - x != y : différent - x > y : supérieur - x >= y : supérieur ou égal - x < y : inférieur - x <= y : inférieur ou égal

1.3 2. Commandes de base

Afficher un message de bienvenue avec **print()**.

[ ]:
```
print("Hello world !")
```

Remarque : Nous sommes dans un interpréteur, il est inutile de d'utiliser **print()**.

[ ]:
```
"Hello world !"
```

On demande une donnée (valeur numérique, texte) à l'utilisateur avec **input()**.

[ ]:
```
input("quel est ton prénom ?")
```

Remarque :
L'ordinateur affiche votre réponse, mais ne la garde pas en mémoire ! Impossible d'utiliser cette réponse par la suite...

C'est pourquoi on utilise des **variables** dans les programmes.

On leur donne un nom, mais pour l'ordinateur il s'agit d'une référence désignant une **adresse mémoire**, c'est à dire un emplacement précis de la mémoire. A cet emplacement est stocké une valeur bien déterminée.

Pour **affecter** une valeur à une variable, on utilise le symbole **=**.

[ ]:
```
var = input("Quel est ton nom ?")
```

Maintenant, on peut récupérer la réponse donnée en appelant la variable dans laquelle on l'a stockée :

[ ]:
```
var
```

[ ]:
```
print("Bonjour ",var)
```

Remarques :

Pour afficher un texte, on l'écrit entre guillemets " ", mais pour afficher une variable on écrit simplement son nom. On peut les afficher simultanément en les séparant par une virgule.

Un nom de variable respecte la syntaxe suivante: - commence par une lettre - ne contient que des lettres (sans accents), des chiffres et le caractère _.

Python est sensible à la casse (c'est à dire l'emploi des majuscules et des minuscules) et a des mots clé réservés.

Par convention on nomme les variables avec des lettres minuscules, mais on peut ajouter une majuscule pour une meilleure lisibilité.

Par exemple : **monPrenom**

On peut combiner plusieurs instructions :

[ ]:
```
print("8+4 =",8+4)
```

1.3.1 Les variables

Il en existe différents types, par exemple :

- les nombres entiers, sont les **integer**, notés **int**
- les nombres décimaux, sont les **float**.
- les chaînes de caractères constituées de caractères alphabétiques, de mots, de phrases ou de suites de symboles quelconques, sont les **strings**, notées **str** (Ce qui est noté entre guillemets simples ou doubles (voir triples) est automatiquement de type str.)
- les booléens, notés *bool*, qui ne prennent que deux valeurs : True ou False. Ils permettent de tester si une expression logique est vraie ou fausse.

Pour connaitre le type d'une variable, il suffit de taper **type(variable)**.

[ ]:
```
a = 2.5   #on utilise le . pour noter les nombres décimaux
type(a)
```

[ ]:
```
b = 10
a < b    #le test est vrai, python renvoie un booléen
```

On peut changer le type d'une variable :

[ ]:
```
c = 2
d = float(2)  #d est un nombre réel
```

```
d
```

[ ]:
```
e = '2'       #e est une chaîne de caractères
print(type(e))
e = int(e)  #e devient un entier
print(type(e))
```

Pour tester si deux variables sont égales ou non, on utlise == (à ne pas confondre avec l'affectation) ou **!=**

La réponse retournée est sous la forme d'un booléen.

[ ]:
```
print(a == 2)   #est-ce que a est égal à 2 ?
print(b == False)
print(a != 5)   #est-ce que a est différent de 5 ?
print(b != 0)
```

Remarque :

Il est très important de commenter vos programmes, pour que d'autres puissent les comprendre et que vous-même puissiez y revenir plus tard sans être perdu.

Pour cela on utilise le symbole **#**. Tout ce qui suit ce caractère ne sera pas exécuté, c'est un commentaire.

Exercice 2 : Ecrire ci-dessous un programme demandant deux nombres entiers et affichant la somme de ces nombres.

[ ]:
```
#code
```

Remarques :

La commande input() donne une chaîne de caractère, même si l'on entre un nombre.

Il vaut mieux utiliser eval(input()) qui reconnaît si la variable donnée est un nombre ou une chaîne de caractères.

[ ]:
```
a = eval(input("Donner un nombre :"))
b = eval(input("Donner un nombre :"))
a+b
```

On peut faire des affectations multiples :

[ ]:
```
x = y = 2
print("x = ",x," y = ",y)
```

On peut affecter des valeurs à plusieurs variables en même temps :

[ ]:
```
a,b = 3,5
print("a =",a," et b = ",b)
```

Exercice 3 : Compléter le code ci-dessous pour échanger les valeurs des variables a et b, puis les afficher.

[ ]:
```
a = 10
b = 15
#suite du code
```

Il peut être nécessaire d'**incrémenter** une variable (l'augmenter d'une certaine valeur) ou de la décrémenter (la diminuer d'une certaine valeur).

[ ]:
```
nb = 5
nb = nb + 1    #on augmente nb de 1
print(nb)
nb += 1        #même chose
print(nb)
nb -= 2        #on diminue nb de 2
print(nb)
```

Remarques :

• Les instructions d'un programme s'exécutent dans l'ordre où elles sont écrites !

• Lorsque l'on fait de la programmation on commet des erreurs (des **bugs**), et l'ensemble des techniques pour les détecter et les corriger s'appelle **debug**.

Il y a trois types d'erreurs :

– les **erreurs de syntaxe** : les règles du langage n'ont pas été respectées (oublie d'une parenthèse, erreur de frappe...) et produisent un arrêt de fonctionnement ;

– les **erreurs sémantiques** : le programme s'exécute parfaitement, pas de message d'erreur, mais le résultat n'est pas celui que l'on attend ;

– les **erreurs à l'exécution** : le programme fonctionne, mais des circonstances particulières (par exemple unfichier déplacé...) se présentent et une erreur se produit.

> Il faudra sans cesse modifier et corriger vos programmes ! Pour vous y aider, l'interpréteur Pyhton affiche des messages d'erreurs, précisant le type de l'erreur et la ligne où elle s'est produite.

1.3.2 Les modules

Beaucoup de programmes ont déjà été écrit sous Pyhton. On en a regroupé certains dans des fichiers appelés **modules** ou **bibliothèques**.

Certains sont installés en même temps que Python. Mais on peut ensuite en utiliser d'autres, suivant nos besoins. Il suffit de les "appeler" en début de programme.

Par exemple, il existe un module de mathématiques (math) contenant des fonctions comme cosinus et sinus, des nombres tels que pi...

Pour utiliser ces modules, on tape la commande : from le-nom-du-module import*

[ ]:
```
from math import*
print(pi)   #on affiche une valeur approchée de pi
print(sqrt(25))  #on obtient la racine carré de 25
```

Un autre module très utile est **random**, car il permet de générer des nombres aléatoires.

[ ]:
```
from random import*
randint(1,50)   #obtenir un nombre entier au hasard entre 1 et 50
```

1.4 3. Les conditions

Pour tester une certaine condition et modifier le comportement du programme en conséquence, on utilise l'instruction **if** (si).

Exercice 4 : Tester le programme ci-dessous avec différentes valeurs :

[ ]:
```
nombre = eval(input("Donner un nombre :"))
if (nombre < 0):
```

```
    print("Le nombre est négatif.")
```

Si la condition indiquée est vraie, ici le nombre est négatif, alors on exécute l'instruction d'affichage.

On peut rajouter une instruction pour le cas où la condition est fausse, voir rajouter d'autres cas.

[ ]:
```
nombre = eval(input("Donner un nombre :"))
if (nombre < 0):
    print("Le nombre est négatif.")
elif (nombre > 0):
    print("Le nombre est positif")
else:
    print("Le nombre est nul.")
```

Remarques :

Le décalage à droite se nomme l'**indentation**. Il est obligatoire pour signifier que les instructions suivantes sont dans la condition.

Les parenthèses autour de la condition ne sont pas obligatoires.

Exercice 5 :

Ecrire un programme qui permet de tester si un nombre donné est pair ou impair, et qui l'affiche.

[ ]:
```
#code
```

1.5 4. Les boucles

L'une des tâches que les machines font le mieux est la répétition de tâches identiques.

Pour cela on peut utiliser une boucle **while** (tant que).

Tant que la condition indiquée est vraie, on répète le bloc d'instructions de la boucle, que l'on repère grâce à l'indentation des lignes. Si la *condition est fausse*, le bloc d'instruction est ignoré.

[ ]:
```
z = 0
while (z < 7):
    z += 1
    print(z,end = " ")
```

Remarques :

L'instruction **end = " "** signifie que l'on veut afficher les nombres sur la même ligne, séparés d'un espace.

La variable évaluée dans la condition doit exister au préalable.

Si la condition reste *toujours vraie*, alors le corps de la boucle est répété indéfiniment (jusqu'à ce que Python cesse de fonctionner). Il faut absoluement l'éviter !

Exercice 6 :

Ecrire un programme qui affiche les 20 premiers termes de la table de multiplication de 8.

[ ]:
```
#code
```

Exercice 7 :

On lance deux dés à 6 faces parfaitement équilibrés et on additionne les deux résultats obtenus.

Ecrire un programme qui simule cette expérience et indique le nombre de lancés nécessaires pour que la somme soit égale à 12.

[ ]:
```
#code
```

Une autre boucle, la boucle **for** (pour).
On répète un certain nombre de fois le bloc d'instructions

[ ]:
```
for i in range(10):
    print(i,end = ";")
```

Remarques :
i est ce qu'on appelle un **compteur**, qui prendra successivement les valeurs entières de 0 à 9.
Et à chaque fois, on exécuter le bloc d'instructions de la boucle.
Quelques variantes :

[ ]:
```
for i in range(1,10):
    print(i,end = ";")
```

[ ]:
```
for i in range(1,20,2):
    print(i,end = ";")
```

Exercice 8 :
Ecrire un programme qui calcule la somme des 25 premiers nombres entiers.

[ ]:
```
#code
```

Exercice 9 :
Ecrire un programme faisant deviner un nombre entier compris entre 1 et 100.
Ce nombre sera choisi au hasard par l'ordinateur ; on indiquera si le nombre proposé est trop grand ou trop petit, ainsi que le nombre de propositions faites jusque-là.
Si l'on dépasse les 6 propositions, on arrête le jeu et on affiche Game Over.

[ ]:
```
#code
```

1.6 5. Les fonctions

Remarque : Les mots suivants ne peuvent être utilisés comme nom de variables, ni comme noms de fonctions :

> and ; as ; assert ; break ; class ; continue ; def ; del ; elif ; else ; except ; False ; finally ; for ; from ; global ; if ; import ; in ; is ; lambda ; None ; nonlocal ; not ; or ; pass ; raise ; return ; True ; try ; while ; with ; yield

On a déjà utilisé certaines **fonctions** préprogrammées, comme **print()**, **input()**, certaines sont regroupées dans des *modules*... ; on peut également en écrire de nouvelles.
Ce sont des suites d'instructions que l'on isolent du reste du programme, auxquelles on donne un nom, ce qui permet d'appeler la fonction par ce nom à n'importe quel endroit du programme.
On peut ainsi utiliser ces fonctions sans avoir à les réécrire, et autant de fois qu'on le souhaite. Elles permettent de rendre le code du programme plus court et plus facile à comprendre.
La syntaxe d'une fonction :
def nom_fonction(paramètres):

> Bloc d'instructions

Remarques :

- il faut indenter les instructions de la fonction
- on peut donner un ou plusieurs paramètres nécessaires à l'exécution des instructions
- une fonction ne fait rien tant qu'elle n'a pas été appelée ; on l'appelle par son nom

[ ]:
```
def table_de_7():              #une fonction sans paramètre
    for i in range(11):
        print("7*",i,"=",7*i,end =" ; ")
```

[ ]:
```
table_de_7()     #on appelle la fonction
```

[ ]:
```
def table_mul(nb,max):    #on utilise deux paramètres
    """paramètres :
       nb : la table demandée
       max : la valeur maximale de la table"""
    for i in range(max+1):
        print(nb,"*",i,"=",nb*i,end =" ; ")
```

[ ]:
```
table_mul(8,20)
```

Remarque : Les triples guillemets dans une fonction, permettent de faire un commentaire où l'on précise les paramètres attendus.

Maintenant, une fonction est créée avant tout pour *renvoyer une valeur*, ce qui se fait avec la commande **return**.

[ ]:
```
def carre(x):
    return x**2  #cette fonction donne le carré du nombre x donné
```

[ ]:
```
carre(5)
```

Exercice 10 :

Ecrire une fonction qui permet de trouver les diviseurs d'un nombre entier positif non nul, et qui les affiche.

[ ]:
```
def diviseurs(n):
    pass            #compléter le code
```

1.6.1 Variables locales et variables globales

Lorsque des variables sont définies à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même : ce sont des **variables locales** ;

Lorsque des variables sont définies à l'extérieur d'une fonction : ce sont des **variables globales**. Leur contenu est visible à l'intérieur d'une fonction, mais la fonction ne peut pas les modifier.

Exercice 11 :

Testez les exemples suivants.

[ ]:
```
def monter():
    v = 5
    return a*2

print(v)  #La variable locale v est inconnue à l'extérieur de la fonction
```