Partie II : Algorithmique et Programmation

Sommaire

Chapitre 3 : Introduction à la programmation en Python

I. Introduction

1. Un peu d'histoire :

Python est un langage de programmation objet interprété. Son origine est le langage de script du système d'exploitation *Amoeba* (1990). Il a été développé par Guido Von Rossum au CWI, à l'Université d'Amsterdam et nommé par rapport au *Monthy Python's Flying Circus*, une série britannique d'un humoristique Monthy Python.

Depuis, Python est devenu un langage de programmation généraliste (*comp.lang.python* premier groupe de discussion des programmeurs Python est créé en 1994). Python offre un environnement complet de développement comprenant un interpréteur performant et de nombreux modules. Un atout indéniable de ce langage est sa disponibilité sur la grande majorité des plates-formes courantes (BeOS, Mac OS X, Unix, Windows).

Python est un langage *open source* supporté, développé et utilisé par une large communauté: 300 000 utilisateurs et plus de 500 000 téléchargements par an.

2. Pourquoi PYTHON ?

Pour résumer Python en quatre points forts.

- *Qualité* L'utilisation de Python permet de produire facilement du code évolutif et maintenable et offre les avantages de la programmation orientée-objet.
- *Productivité* Python permet de produire rapidement du code compréhensible en reléguant nombre de détails au niveau de l'interpréteur.
- *Portabilité* La disponibilité de l'interpréteur sur de nombreuses plates-formes permet l'exécution du même code sur un PDA ou un gros système.
- *Intégration* L'utilisation de Python est parfaitement adaptée l'intégration de composants écrit dans un autre langage de programmation (C, C++, Java avec Jython). Embarquer un interpréteur dans une application permet l'intégration de scripts Python au sein de programmes.

Quelques caractéristiques intéressantes:

- langage interprété (pas de phase de compilation explicite)
- pas de déclarations de types (déclaration à l'affectation)
- gestion automatique de la mémoire (comptage de références)
- programmation orienté objet, procédural et fonctionnel
- par nature dynamique et interactif
- possibilité de générer du byte-code (améliore les performances par rapport à une interprétation perpétuelle)
- interactions standards (appels systèmes, protocoles, etc.)
- intégrations avec les langages C et C++

II. Notions de base :

1. *Variables :*

Une **variable** est une zone de la mémoire dans laquelle une **valeur** est stockée. Aux yeux du programmeur, cette variable est définie par un **nom**, alors que pour l'ordinateur, il s'agit en fait d'une adresse (*i.e.* une zone particulière de la mémoire).

En Python, la **déclaration** d'une variable est son **initialisation** (c.-à-d. la première valeur que l'on va stocker dedans) se fait en même temps.

Exemple :

```
x =  2   # pour créer un entier initialisé par la valeur 2
```

2. *Types des variables :*

Le **type** d'une variable correspond à la nature de celle-ci. Les trois types principaux dont nous aurons besoin sont les **entiers**, les **réels** les **booléens** et les **chaînes de caractères**. Bien sûr, il existe de nombreux autres types (par exemple, les nombres complexes), c'est d'ailleurs un des gros avantages de Python.

Dans l'exemple précédent, nous avons stocké un nombre entier (*int*) dans la variable **x** , mais il est tout à fait possible de stocker des nombres réels (*float*) ou des chaînes de caractères (*string*) :

```
y = 3.14

a = "bonjour"

b = 'salut'

c = """girafe"""
```

Vous remarquez que Python reconnaît certains types de variable automatiquement (entier, réel). Par contre, pour une chaîne de caractères, il faut l'entourer de guillemets (simples, doubles voire triples) afin d'indiquer à Python le début et la fin de la chaîne.

Les triples guillemets sont utilisés surtout pour initialiser une variable avec une chaîne de caractère écrit sur plusieurs lignes.

3. *Nommage des variables :*

Le nom des variables en Python peut-être constitué de lettres minuscules (a à z), de lettres majuscules (A à Z), de chiffres (0 à 9) ou du caractère souligné (_).

Néanmoins, un nom de variable ne doit pas débuter ni par un chiffre, ni par _ et ne peut pas contenir de caractère accentué. Il faut absolument éviter d'utiliser un mot « réservé » par Python comme nom de variable (par ex. : print, range , for , while , etc.).

Python est sensible à la casse, ce qui signifie que les variables **TesT** , test ou **TEST** sont différentes. Enfin, n'utilisez jamais d'espace dans un nom de variable puisque celui-ci est le séparateur d'instructions.

4. Opérateurs :

Un *opérateur* est un symbole utilisé pour effectuer un calcul entre des **opérandes**.

Un opérande est une variable ou un littéral ou bien une **expression**.

Une expression est une suite valide d'opérateurs et d'opérandes.

Par exemple, dans l'expression :

```
x = y + 1
```

Il y a deux opérateurs (= et +) et trois opérandes (x, y et 1).

Il existe différents types d'opérateur :

- les opérateurs logiques
- les opérateurs de comparaisons
- les opérateurs sur les séquences
- les opérateurs numériques
- les opérateurs d'affectation

a) Les opérateurs logiques :

Les expressions avec un opérateur logique sont évaluées à True ou False

- X **or** Y : OU logique, si X évalué à True, alors l'expression est True et Y n'est pas évalué. Sinon, l'expression est évaluée à la valeur booléenne de Y.
- X **and** Y : ET logique, si X est évalué à False, alors l'expression est False et Y n'est pas évalué. Sinon, l'expression est évaluée à la valeur booléenne de Y.
- **not** X : évalué à la valeur booléenne opposée de X.

b) Les opérateurs de comparaisons :

Tout comme les opérateurs logiques, les opérateurs de comparaison renvoient une valeur booléenne True ou False. Les opérateurs de comparaisons s'appliquent sur tous les types de base.

- < strictement inférieur
- > strictement supérieur
- <= inférieur ou égal
- >= supérieur ou égal
- == égal
- **!=** différent
- < > différent, on utilisera de préférence !=

Il est possible d'enchaîner les opérateurs : X < Y < Z, dans ce cas, c'est Y qui est pris en compte pour la comparaison avec Z et non pas l'évaluation de (X < Y) comme on pourrait s'y attendre dans d'autres langages.

c) Les opérateurs mathématiques :

Symbole	Effet	Exemple
+	Addition	6+4==10
-	Soustraction	6-4==2
*	Multiplication	6*4==24
/	Division réelle	6/4==1.5
**	Elévation à la puissance	12**2==144
//	Division entière	6//4==1
%	Reste de la division entière	6%4==2

d) Les opérateurs d'affectation :

- '='
- Affectation multiple : *x = y = z = 3* *Les trois variables x,y, et z auront la même valeur 3*
- Affectation parallèle : *x , y = 1 , 0.5* *La valeur x sera 1 et de y sera 0.5*

5. *Instruction de lecture et d'écriture :*

a) L'instruction d'écriture :

La fonction « **print** » permet d'afficher n'importe quel nombre de valeurs fournies en arguments (c'est-à-dire entre les parenthèses). Par défaut, ces valeurs seront séparées les unes des autres par un espace, et le tout se terminera par un saut à la ligne.

Vous pouvez remplacer le séparateur par défaut (l'espace) par un autre caractère quelconque (ou même par aucun caractère), grâce à l'argument « sep ». Exemple :

```
>>> print("Bonjour", "à", "tous", sep ="*")
Bonjour*à*tous
 >>> print("Bonjour", "à", "tous", sep ="")
Bonjouràtous
```

De même, vous pouvez remplacer le saut à la ligne terminal avec l'argument « end » :

```
>>> print("zut", end ="")
>>> print("zut", end ="")

zutzut
```

b) L'instruction de lecture :

La plupart des scripts élaborés nécessitent à un moment ou l'autre une intervention de l'utilisateur (entrée d'un paramètre, clic de souris sur un bouton, etc.). Dans un script en mode texte (comme ceux que nous avons créés jusqu'à présent), la méthode la plus simple consiste à employer la fonction intégrée **input( )**. Cette fonction provoque une interruption dans le programme courant. L'utilisateur est invité à entrer des caractères au clavier et à terminer avec <*Enter*>. Lorsque cette touche est enfoncée, l'exécution du programme se poursuit, et la fonction fournit en retour une chaîne de caractères correspondant à ce que l'utilisateur a entré. Cette chaîne peut alors être assignée à une variable quelconque, convertie, etc.

On peut invoquer la fonction **input()** en laissant les parenthèses vides. On peut aussi y placer en argument un message explicatif destiné à l'utilisateur. Exemple :

```
prenom = input("Entrez votre prénom : ")
print("Bonjour,", prenom)
```

ou encore :

```
print("Veuillez entrer un nombre positif quelconque : ", end=" ")
ch = input()
nn = int(ch)            # conversion de la chaîne en un nombre entier
print("Le carré de", nn, "vaut", nn**2)
```

Remarque importante :

La fonction **input()** renvoie toujours une chaîne de caractères. Si vous souhaitez que l'utilisateur entre une valeur numérique, vous devrez donc convertir la valeur entrée (qui sera donc de toute façon de type string) en une valeur numérique du type qui vous convient, par l'intermédiaire des fonctions intégrées **int()** (si vous attendez un entier) ou **float()** (si vous attendez un réel). Exemple :

```
a = int(input("Entrez un entier : "))
a = float(input("Entrez un réel : "))
```

III. Structures de contrôles :

1. *Structures conditionnelles :*

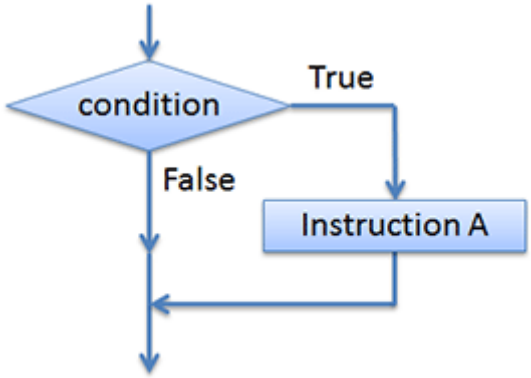
Une structure conditionnelle permet au programme d'agir de façons différentes en fonction d'une condition (test).

a) Structure if :

Syntaxe :

```
if condition:
    Instruction A
```

Organigramme :



condition est une expression booléenne, c'est-à-dire une expression qui prend pour valeur True (Vrai) ou False (Faux).

L'instruction A n'est exécutée que si la condition est vérifiée (c'est-à-dire si elle prend pour valeur True).

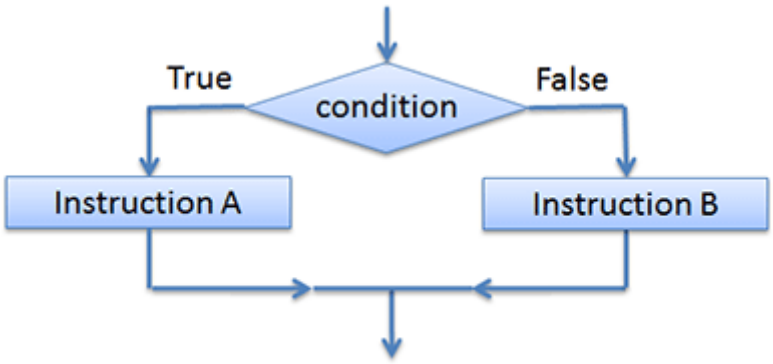
L'instruction if est une instruction composée. Le : (deux-points) à la fin de la ligne introduit le bloc d'instructions qui sera exécuté si la condition est vérifiée.

b) Structure if … else ….:

Syntaxe :

```
if condition:
    Instruction A
else:
    Instruction B
```

Organigramme :



c) Structures if imbriquées :

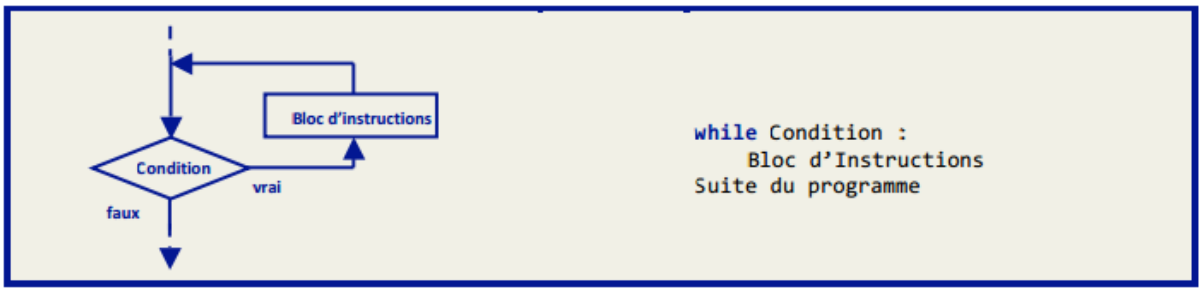
Syntaxe :

```
if condition1 :
   Instruction A
elif condition2 :
   Instruction B
elif condition3 :
   Instruction C
```

2. *Structures répétitives :*

Une structure répétitive ou boucle permet de répéter une portion de code.

a) La boucle while :



Tant que la condition est vraie (True) le bloc d'instructions est exécuté. Le cycle continu jusqu'à ce que la condition soit fausse (False) : on passe alors à la suite du programme.

Exemple 1 : table de multiplication par 8 avec la boucle while

```
print("Table de multiplication par 8")
compteur = 1                    # initialisation de la variable de comptage
while compteur <= 10 :
    # ce bloc est exécuté tant que la condition (compteur<=10) est vraie
    print(compteur,"* 8 =",compteur*8)
    compteur += 1           # incrémentation du compteur : compteur = compteur + 1
# on sort de la boucle
print("Eh voilà !")
```

```
>>>
Table de multiplication par 8
1 * 8 = 8
2 * 8 = 16
3 * 8 = 24
4 * 8 = 32
5 * 8 = 40
6 * 8 = 48
7 * 8 = 56
8 * 8 = 64
9 * 8 = 72
10 * 8 = 80
Eh voilà !
```

b) La boucle for :

```
for élément in séquence :
    Bloc d'Instructions
Suite du programme
```

La séquence est parcourue élément par élément. L'élément peut être de tout type : enier, caractère, élément d'une liste…

L'utilisation de la boucle for est intéressante si le nombre de boucles à effectuer est connu à l'avance.

Exemple 3 : table de multiplication par 9 avec la boucle for

```
print("Table de multiplication par 9")
for compteur in range(1,10) :
    print(compteur,"* 9 =",compteur*9)
# on sort de la boucle
print("Et voilà !")
```

La **valeur initiale** de l'élément compteur est égale à **1**. On **exécute la boucle tant que** l'élément compteur est **inférieur à 10**.

```
>>>
Table de multiplication par 9
1 * 9 = 9
2 * 9 = 18
3 * 9 = 27
4 * 9 = 36
5 * 9 = 45
6 * 9 = 54
7 * 9 = 63
8 * 9 = 72
9 * 9 = 81
Et voilà !
```