Les bases du langage Python

SOMMAIRE

PRESENTATION DU DOCUMENT

Le but de ce document est d'écrire rapidement un programme **simple** dans en Python.

Dans cette optique, le choix a été fait de ne parler que du strict minimum et ne pas présenter les spécificités de ce langage. Il est évident qu'il ne peut remplacer un cours complet.

I. LES TYPES DE VARIABLES

1. Les types de données :

Les types de données de base sont :

Type	Utilisation
int	Entier positif ou négatif
float	Nombre réel à virgule flottante
str	Chaîne de caractère
complex	Nombre complex
bool	Booléen : True ou False uniquement

2. Déclaration et affectation de variables.

Afin que votre code soit le plus compréhensible possible, il faut veiller à utiliser des noms de variables pertinents : le nom de votre variable doit indiquer sa fonction.
`aire, perimetre, volume` à la place de `a, b, c`

Exception faite pour les variables d'incrémentation dont les noms sont traditionnellement `i, j ou k`

Il n'y a pas de déclaration de variable, la définition du type de la variable se fait automatiquement lors de l'affectation :

```
x = 3
pi = 3.14159
message = 'Coucou'
```

Il est possible de connaître le type d'une variable par la fonction *type :*

```
type(x)                    <class 'int'>
type(pi)                   <class 'float'>
type(message)              <class 'str'>
```

3. Exemples

Python

```
i = 0
rayon = 8.2
nom = "Nestor" # ou nom='Nestor'
OuiNon = True
```

4. Chaîne de caractères

Vous allez souvent utiliser des chaînes de caractères pour afficher un message dans le terminal pour débugger ou sur l'écran LCD que vous avez mis en œuvre :

```
>>> message = 'Bonjour'
>>> message
'Bonjour
```

Il est possible d'utiliser les doubles guillemets :

```
>>> message = "Bonjour"
>>> message
'Bonjour'
```

Vous pouvez également ajouter une chaîne à une autre comme ceci :

```
>>> message = 'Bonjour'
>>> nom = 'toto'
>>> message + nom
'Bonjourtoto'
```

☝ Remarquez l'absence d'espace entre `Bonjour` et `toto`. Si vous voulez un espace vous pouvez écrire

```
>>> message + ' ' + nom
'Bonjour toto
```

☝ Nous ne parlerons pas ici de slicing (découpage) des chaînes de caractères.

5. Liste

Les listes en Python sont des collections ordonnées et modifiables d'éléments. Elles peuvent contenir n'importe quel type de données (nombres, chaînes, objets, autres listes, etc.) et sont définies en entourant les éléments de crochets [].

a. Création d'une liste

Une liste peut être vide :

```
uneListeARemplir = []
```

Elle peut également contenir des données de même type :

```
listePrix = [125 , 25 , 35 , 45]
mobiliers = ['tables', 'chaise','canapé']
```

Et même plusieurs types différents :

```
listeMixte = ['toto', 3 , 4.5 , True]
```

☝ La syntaxe d'une liste est la suivante : les valeurs sont séparées par une virgule et l'ensemble est encadré par des crochets

b. L'accès aux éléments

L'accès à un élément de la liste peut se faire par l'intermédiaire de son indice, et comme souvent en informatique, on commence à compter à partir de zéro. Ainsi le premier élément de la liste listePrix est listePrix[0] qui correspond à 125 :
listePrix[0]
Tout logiquement, le second élément est à l'indice 1 :
listePrix[1]

```
premier_nom = noms[0]  # Résultat: "Alice"
dernier_nombre = nombres[-1]  # Résultat: 5
```

c. 4. Ajout d'un élément:

```
noms.append("Diana")  # ["Alice", "Brandon", "Charlie", "Diana"]
```

d. 5. Suppression d'un élément

```
del nombres[2]  # Supprime l'élément à l'index 2.
```

La liste 'nombres' devient [1, 2, 4, 5]

e. 6. Slicing (Découpage)

```
sous_liste = nombres[1:3]  # Résultat: [2, 4]
```

6. Les dictionnaires

Les dictionnaires en Python sont des collections non ordonnées, modifiables et indexées. Ils sont composés de paires clé-valeur et sont définis par des accolades `{}`. Contrairement aux listes, les dictionnaires stockent des éléments en tant que paires clé-valeur, et on accède à ces éléments via leurs clés plutôt que par leur indice.

a. Création de dictionnaires:

```
personne = {"nom": "Alice", "âge": 30, "ville": "Paris"}
notes = {"Maths": 15, "Histoire": 12, "Physique": 18}
```

b. Accès aux éléments:

```
nom = personne["nom"]  # Résultat: "Alice"
note_maths = grades["Maths"]  # Résultat: 15
```

c. Modification d'une valeur:

```
personne["âge"] = 31  # Modifie l'âge d'Alice à 31
```

d. Ajout d'une paire clé-valeur:

```
personne["emploi"] = "Ingénieure"  # Ajoute la clé "emploi" avec la valeur "Ingénieure"
```

e. Suppression d'une paire clé-valeur:

```
del notes["Histoire"]  # Supprime la clé "Histoire" et sa valeur associée
```

f. Obtenir toutes les clés ou toutes les valeurs:

```
clés = personne.keys()  # Résultat: ["nom", "âge", "ville", "emploi"]
valeurs = notes.values()  # Résultat: [15, 18]
   18]
```

7. *Les tuples*

Les tuples en Python sont des collections ordonnées et immuables. Contrairement aux listes, une fois qu'un tuple est créé, vous ne pouvez pas modifier, ajouter ou supprimer ses éléments. Les tuples sont souvent utilisés pour représenter une collection de données hétérogènes ou pour garantir que les données restent inchangées. Ils sont définis en plaçant des éléments entre parenthèses `()`.

a. Création de tuples:

```
point = (3, 4)  # un point en coordonnées (x, y)
date = (2021, "septembre", 10)  # une date avec année, mois et jour
```

b. Accès aux éléments:

```
x_coord = point[0]  # Résultat: 3
mois = date[1]  # Résultat: "septembre"
```

c. Immuabilité:

```
# date[0] = 2022  # Cela générera une erreur car les tuples sont immuables
```

d. Tuple avec un seul élément (il faut une virgule après l'élément):

```
singleton = (5,)
```

e. Emballage et déballage de tuples:

```
dimensions = (1920, 1080)  # Emballage
largeur, hauteur = dimensions  # Déballage
```

f. Fonctionnalités utiles:

Les tuples peuvent être utilisés comme clés dans les dictionnaires et peuvent aussi être des éléments de sets, grâce à leur nature immuable.

II. INTERAGIR AVEC L'UTILISATEUR

1. Afficher avec print

L'affichage peut se faire simplement sans avoir recours à l'écriture formatée.

- Pour un texte :

```
print ('Hello World')
print ('Hello' + 'World') #Affiche HelloWorld
print (3 * '*') #Affiche ***
```

- Une variable :

```
rayon = 8.2
print (rayon)
```

- Texte + variable :

```
print ('rayon = ', rayon)
```

Ou alors convertir la variable en string pour la concaténer avec la chaîne 'rayon' comme ceci :

```
print('rayon = ' + str(rayon) )
```

2. Lire avec input

La fonction essentielle est *input* qui lit une chaîne de caractère :

```
print('Entrez un message')
message = input()
print(message)
```

Une écriture équivalente quasi (hormis le saut de ligne) équivalente est :

```
message = input('Entrez un message : ')
print(message)
```

Pour les autres types de variables que *string*, il faut convertir la chaîne :

- Pour lire un entier : `i = int(input())`
- Un réel : `rayon = float(input())`

3. Exemples

Python

```
print ('Hello World')

#Affiche HelloWorld :
#Ces deux lignes affiche HelloWorld0000
print ('Hello' + 'World')
# end = "" -> sans saut de ligne
print ('Hello', end = "");
print ('World');

#Affiche SNECSNECSNEC
print (3 * 'SNEC')

rayon = 8.2
print (rayon)
print ('rayon = ', rayon)


message = input ('Entrez un message : ')
#Avec + pas de saut de ligne
print ('Le message est ' + chaine)

i = int (input ('i =  '))
rayon = float (input ('rayon = '))
print ('i = ' + str (i))
print ('rayon = ' + str (rayon))
```

III. LES BRANCHEMENTS CONDITIONNELS

1. Si ... alors…(sinon)

Condition du test : ==, <,>,<=,>=, !=,&&,|| …

Python	Explications

note = 8
if note > 10:
print('reçu')
print('bravo !')
else:
print('recalé')
	1. L'instruction if
	2. Le test conditionnel se termine par « : » le signe **deux points.**
	3. A la place des parenthèses du langage C, python utilise **l'indentation** : quatre espaces ou une tabulation, nous préférons le second format
	4. Un bloc d'instruction (une ou plusieurs lignes) dans le cas où la condition est vérifiée
	5. Else pour indique le cas où la condition n'est pas vérifiée
	6. A nouveau les deux points
	7. Et un bloc d'instructions

Remarques

- il ne faut pas confondre l'opérateur d'égalité **==** et celui de l'affection **=**
- *sinon* est optionnel

Les condition du test sont : ==,<,>,<=,>=, !=, **and**, **or.**

2. Il est possible Avec else if contracté en elif

Remarquer les **indentations** (elif comme else est au même niveau que le if)

```
x = 'Roger'
if x == 'roger':
    print("manque une majuscule")
elif x == 'Roger ':
    print("un espace en trop")
else:
    print('Pas trouvé!')
```

3. Exemple

Python

```
if note>10:
    print("reçu")#une tabulation avant
    print("bravo")
else:
    print("recalé")
```

IV. COMMENTAIRES ET MODULES

1. Les commentaires

Un code bien écrit doit être facilement compréhensif : en le lisant on doit comprendre ce que vous avez fait. Pour cela, il faut au moins utiliser des noms de variables qui indiquent leurs fonctions.Si cela n'est pas suffisant, les commentaires servent à préciser votre code.

L'explication sur un ligne débute par « # » :

```
# Sur une ligne
```

Et sur plusieurs lignes, encadré par « """ »

```
""" Sur plusieurs
lignes   """
```

2. Les modules

Par exemple pour importer la fonction sqrt (square root = racine carrée) :

```
from math import sqrt
print(sqrt(16))
```

Il est possible d'en importer plusieurs à la fois :

```
from math import sqrt, pi, cos
print(cos(pi/2))
```

Et même toutes les fonctions du module math :

```
import math
print(math.sqrt(16)) # Affiche 4
```

Un autre exemple où en plus d'importer, on a défini un synonyme (PI en majuscule au lieu de pi)

```
from math import pi as PI
print(PI)
```

Et même renommer un module :

```
import numpy as np
import matplotlib.pyplot as plt
```

V. LES STRUCTURES DE CONTROLE

1. La boucle for pour répéter un nombre de fois

Cette boucle est utilisée lorsqu'on connait le nombre d'itérations avant même de l'appeler c'est-à-dire :
La boucle pour est utilisée lorsque l'on connait le nombre de fois que la boucle va être parcourue

Par exemple au lieu d'écrire :

```
print('bonjour') # ceci est un commentaire 1
print('bonjour') # 2
print('bonjour') # 3
print('bonjour') # 4
print('bonjour') # 5
print('bonjour') # 6
print('bonjour') # 7
print('bonjour') # 8
print('bonjour') # 9
print('bonjour') # 10
```

Il est possible d'utiliser la boucle *for* :

Python
```
for  i in range(0,10):
    print('bonjour') #tabulation au début
```

Comme pour le if, python utilise **l'indentation** qui est soit une tabulation, soit quatre espaces.

La fonction range()

La fonction *range* permet de créer une liste de nombres compris entre un nombre de départ (inclus) et un nombre de fin (exclus).

Range() a trois paramètres : début, fin et pas, exemple :

```
for i in range(10): # de 0 à 9 (10 non inclus)
for i in range(1,10): # de 1 à 9
for i in range(0,100,5): # de 0 à 95 par pas de 5
for i in range(0,-10,-3): # de 0 à -9 par pas de -3
```

Par exemple pour afficher 10 fois bonjour :

```
for  i in range(0,10):
    print('bonjour') # indentation au début de ligne
```

Exemple simple

```
for i in range(1,11):
    print("i=",i)
```

Boucles imbriquées :

```
for i in range(101):
  for j in range (10):
    print(j)
    if (j%2 == 0) :
      print('Pair') # dans le if
    print(i*j) # dans le 2 for
  print(i) # dans le 1er
print('Fin du programme') # à la fin
```

2. While pour répéter en respectant une condition

Cette boucle est utilisée quand on ne connait pas le nombre de fois que la boucle doit être itérée.

Il ne faut pas oublier les deux points « : » à la fin de la ligne et l'indentation (tabulation ou espace).

```
x=45
y=55
while x<50 and y <70:
    x=x+1
    y=y+1
    print(x,y)
```

Exemples

Tant que la somme <100

```
somme = 0
while somme<100:
    somme = 2*somme + 1
print('somme=',somme)
```

A la place d'une boucle for

Python

```
for i in range(1,11):
    print("i =",i)
```

```
i = 1
while ( i <10 ):
  print("i =",i)
  i = i + 1
```

VI. LES FONCTIONS

1. Le concept de fonction

Les fonctions déjà présentes jusqu'alors

Vous avez déjà utilisé en Python des fonctions : print(…), int(…), range(…)
Elles sont reconnaissables par des parenthèses à la fin de leur nom. Elles ont toutes été conçues et mises au point par d'autres personnnes, puis réunies dans des modules (Python).

Besoin :

Dès que le programmme commence à prendre de l'ampleur, il est nécessaire d'écrire vos propres fonctions : un programme de plus de cent lignes commencent à être illisible, difficile à corriger et des parties sont répétitives. De plus, il est nécessaire de décomposer votre travail, en sous-tâches, qui vont être codées en fonctions.

Une fonction c'est :

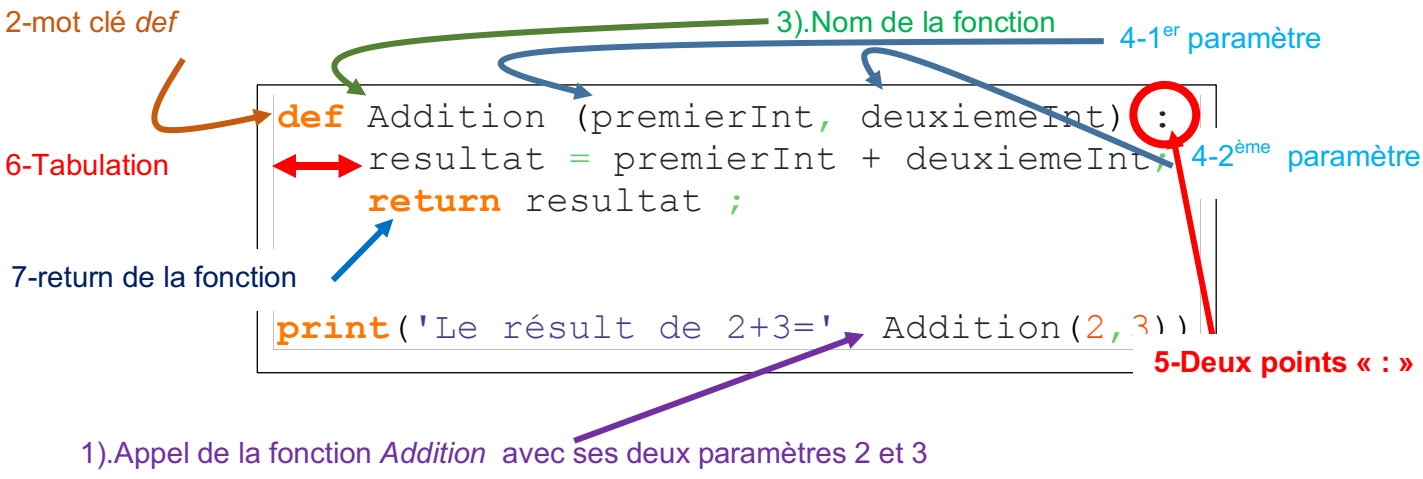
Une suite instructions isolées du reste du programme, qui possède un nom, et qui peut être appelée par ce nom à n'importe quel endroit du programme et autant de fois que l'on veut.

Points essentiels :

- Un programme écrit sans fonction devient difficile à comprendre dès qu'il dépasse un centaine de lignes
- Les fonctions permettent de scinder le programme principal en plusieurs parties
- Le programme principal regroupe les fonctions en décrivant les enchaînements
- Une fonction peut elle-même, être découpée en plusieurs autres fonctions.
- Et pour terminer, et peut-être le plus essentiel, les fonctions permettent le partage de tâches : dans un projet important où travaille plusieurs développeurs, chacun est responsable d'une ou de plusieurs fonctions, qui sont ensuite utilisées par les autres programmeurs et mise en commun dans le programme principal.

Dans cette partie, nous ne parlerons pas de modularité (découper le fichier principal en plusieurs fichiers), mais cette notion est indispensable pour les plus longs programme.

2. Exemple commenté




```
def Addition (premierInt, deuxiemeInt) :
    resultat = premierInt + deuxiemeInt;
    return resultat ;

print('Le résult de 2+3=', Addition(2,3))
```

1. Cette fonction *Addition* est appelée avec deux paramètres. *Addition* renvoie une valeur de retour.
2. Dans la déclaration de la fonction, le premier terme est **def**
3. Le second terme est le nom de la fonction
4. Ensuite les définitions des paramètres
5. **Deux points** « **:** »
6. **Une tabulation** (ou indentation)
7. Un renvoie de la valeur par *return*

Remarques :

- On retrouve les deux points et la tabulation des boucle *for* et *while*
- **Ne jamais mettre de print dans les fonctions, sauf nécessité, utilisez *return* pour renvoyer une valeur puis l'afficher.**
- L'instruction *return* renvoie une valeur et met fin à l'exécution de la fonction et redonne le contrôle à la fonction appelante.
- Par (mauvaises) habitudes de nombreux programmeurs utilisent des parenthèses pour encadrer la valeur de retour. *return* est une **instruction** pas une fonction

3. Fonction sans paramètre ne retournant pas de valeur

```
import random


def Affiche10Fois():
    '''
    Fonction sans paramètre
    ne retournant pas de valeur
    '''
    for i in range(10):
        print('coucou')
```

4. Fonction sans paramètre, retournant une valeur

```
def Aleatoire():
    '''
    Fonction sans paramètre,
    retournant une valeur
    '''
    return random.randint(0, 20)
```

5. Deux paramètres sans retour de valeur

```
def AfficherAddition(i, j):
    '''
    Fonction utilisant un ou plusieurs paramètres
    ne retournant pas de valeur
    '''
    print('addition entre',i,'et','j','est',i+j)
```

6. Plusieurs paramètres et retournant une valeur

```
def Discriminant(b, a, c):
    '''
    Fonction utilisant plusieurs paramètres
    retournant une valeur
    '''
    delta = b * b - 4 * a * c;
    return delta
```

7. Les fonctions sont appelées

```
Affiche10Fois()
nombre = Aleatoire()
print('un nombre aléatoire entre 0 et 20 : ', nombre);
AfficherAddition(2, 3)  # Affiche L'addition entre 2 et 3 est 5
nombre = Discriminant(6, 3, 2);
print('Le Discriminant est ', nombre); # Affiche le delta
```

Remarques

- Quatre fonctions ont été écrites puis utilisées plus bas
- le programme principal devient alors court et lisible
- il est possible de « mimer » une fonction *main* comme étant le programme principal à l'aide de l'instruction

```
if __name__ == "__main__":
```