LE LANGAGE PYTHON

1 – LANGAGE PYTHON : PRESENTATION

1 – PRESENTATION

Le langage Python est un **langage de programmation objet interprété**. Il a été développé par **Guido Von Rossum en 1989** à l'Université d'Amsterdam. Ce langage a été nommé ainsi en référence à la série télévisée Monthy Python's Flying Circus.

Python offre un **environnement complet de développement** comprenant un interpréteur performant et de nombreux modules.

Un atout indéniable est sa **disponibilité sur la grande majorité des plates-formes informatiques** courantes : Mac OS X, Unix, Windows ; Linux, Android, IOS….

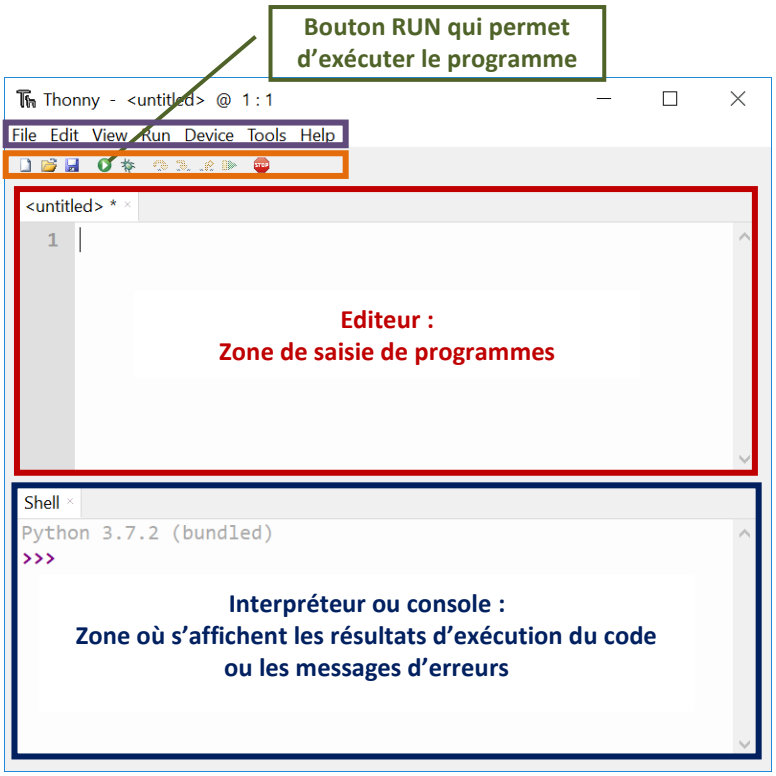
Python est un langage open source. **Libre et gratuit**, il est supporté, développé et utilisé par une large communauté : 300 000 utilisateurs et plus de 500 000 téléchargements par an.

Avec le langage Python il est possible de faire :

- du calcul scientifique (librairie **NumPy**) ;
- des graphiques (librairie **matplotlib**) ;
- du traitement du son ;
- du traitement d'image (librairie **PIL**) ;
- des applications avec interface graphique GUI (librairies **Tkinter**, **PyQt**, **wxPython**, **PyGTK** ...)
- des jeux vidéo en temps réel (librairie **Pygame**)
- des applications Web (serveur Web **Zope** ; framework Web **Django**, **Karrigell** ; framework JavaScript **Pyjamas**)
- interfacer des systèmes de gestion de base de données (librairie **MySQLdb** ...) ;
- des applications réseau (framework **Twisted**) ;
- communiquer avec des ports série RS232, Bluetooth... (librairie **PySeria**l) ;
- ...

2 – PREMIERS PAS AVEC PYTHON

Il existe beaucoup d'IDE, interface de développement, permettant de programmer en Python. Nous utiliserons « Thonny » qui est projet libre de l'université de Tartu (Estonie), qui présentant une interface simple (en anglais) et un outil de débogage qui permet un suivi de chacune des opérations réalisées lors de l'exécution d'un programme en Python.



2.1 – Interpréteur ou console

L'interpréteur est facilement reconnu. C'est lui qui contient le triple chevron >>> qui est l'invite de Python (**prompt** en anglais) et qui signifie que Python attend une commande. L'utilisation de l'interpréteur ressemble à **l'utilisation d'une calculatrice** :

Ligne de code dans l'interprétateur

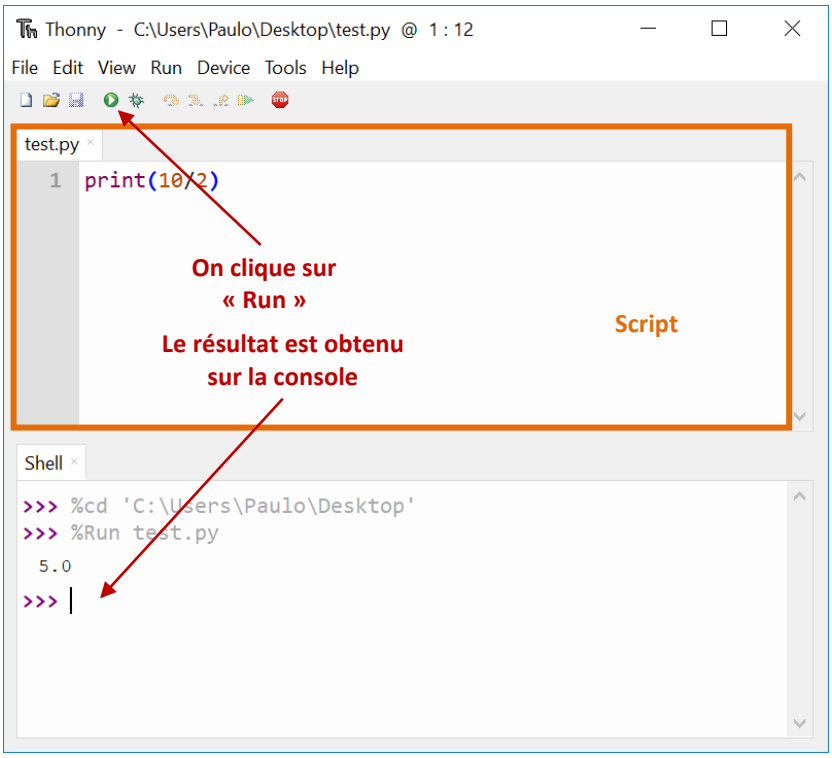
```
>>> 10*2+5-3
22
```

2.2 – Editeur

Un **script Python** est formé d'une **suite d'instructions** exécutées en séquence de haut en bas. L'éditeur permet d'écrire des scripts. L'extension des scripts python est **.py**.

Ligne de code dans l'éditeur

```
print(10/2)
```



2.3 –Quelques recommandations

Un **commentaire** commence par le caractère **#** et s'étend jusqu'à la fin de la ligne.

Exemples de commentaires

```
#-----------------------------------------------------
# Ceci est un commentaire
#-----------------------------------------------------
print(10/2)  # En voici un autre
```

Le langage Python compte 33 mots clés :

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

Les **instructions Python** sont éditées en **minuscules**. Les noms de variables **peuvent contenir des majuscules** mais il faudra les écrire toujours de la même façon en respectant les minuscules et majuscules

2 – TYPES, VARIABLES ET OPERATEURS

2.1 – TYPES DE DONNEES

Les types primitifs des données en langage Python sont : **booléen** (bool), **entier** (int), **entier long** (long), **nombre flottant** (float) et **chaîne de caractères** (str).

2.1.1 – Type INT (integer : nombres entiers)

Par défaut, les **entiers** sont des nombres décimaux, mais il est possible d'utiliser également la base binaire ou hexadécimale.

Entiers

```
>>> 2013                # décimal
2013

>>> 0b11111011101       # binaire
2013

>>> 0x7DD               # hexadécimal
2013
```

Les opérations arithmétiques sur les entiers sont les suivantes :

Addition

```
>>> 50 + 3              # addition
53
```

Soustraction

```
>>> 50 - 3              # soustraction
47
```

Multiplication

```
>>> 50 * 3              # multiplication
150
```

Division

```
>>> 50 / 3              # division
16.666666666666668
```

Division entière

```
>>> 50 // 3       # division entière
16
```

Modulo

```
>>> 50 % 3             # modulo
2
```

2.1.2 – Type FLOAT (flottant ou à virgule flottante)

Une donnée de type **float** ou **réelle** est notée avec un point décimal ou en notation exponentielle :

Flottants

```
>>> 4.215
4.215

>>> .0087
0.0087

>>> 8e9
8000000000.0

>>> 1.025e38
1.025e+38
```

Les flottants **supportent les mêmes opérations** que les entiers mais ils ont une **précision finie**.

L'importation du **module math** permet l'utilisation de **fonctions mathématiques usuelles**.

Fonctions mathématiques usuelles

```
>>> import math

>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc',
'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']

>>> math.sin(math.pi/4)            # sin(pi/4)
0.7.71067811865475

>>> math.degrees(math.pi)          # pi en degrés
180.0

>>> math.sqrt(2)                   # racine carrée de 2
1.4142125623730951
```

2.1.3 – Type BOOL (booléen)

Les données du type **bool** ne présentent que deux valeurs : **False** et **True**. Les opérations logiques et comparaison **sont évaluées** et le résultat est un **booléen**.

Opérateurs de comparaison

```
>>> 2 < 8          # strictement inférieur
True

>>> 2 <= 8         # inférieur ou égal
True

>>> 2 == 8         # égal
False

>>> 2 > 8          # strictement supérieur
False

>>> 2 >= 8         # supérieur ou égal
False

>>> 2 != 8         # différent
True
```

Opérateurs logiques

```
>>> (3 == 3) or (9 > 24)          # OU logique
True

>>> (9 > 24) and (3 == 3)         # ET logique
False

>>> not(3 == 3)                   # NON logique
False
```

2.1.4 – Type STR (string ou chaîne de caractère)

Une donnée de type **str** représente une séquence constituée de caractères.

Représentation d'une chaîne de caractères

```
>>> "Dupont"           # utilisation des guillemets
'Dupont'

>>> 'Pierre'           # utilisation des apostrophes
'Pierre'
```

Pour une chaîne de caractères avec apostrophes, il faut utiliser la **séquence d'échappement **.

**Séquence d'échappement **

```
>>> 'Aujourd'hui'
      File "<pyshell>", line 1
      'Aujourd'hui'
               ^
      SyntaxError: invalid syntax

>>> 'Aujourd\'hui'            # utilisation de la séquence d'échappement
"Aujourd'hui"
```

Pour un **saut à la ligne** il faut utiliser la **séquence d'échappement \n** ou la forme **multi-lignes** avec **triples guillemets**.

Saut à la ligne

```
>>> chaine = 'Dupont\nPierre'      # séquence d'échappement \n

>>> print(chaine)
Dupont
Pierre

>>> chaine = """Dupont
Pierre"""                          # Forme multi-lignes

>>> print(chaine)
Dupont
Pierre
```

Opérations sur les chaines de caractères :

Opérations sur les chaînes de caractères

```
>>> 'Dupont'+' '+'Pierre'     # concaténation de chaînes de caractères
'Dupont Pierre'

>>> chaine = 'Dupont Pierre'

>>> len(chaine)               # longueur d'une chaîne de caractères
13

>>> chaine = 'Ha ! '

>>> chaine * 3                # répétition
'Ha ! Ha ! Ha ! '
```

Indexage

```
>>> chaine = 'Dupont Pierre'

>>> print(chaine[0])    # premier caractère
D

>>> print(chaine[-1])   # dernier caractère
e

>>> print(chaine[2:6]) # du 3ième au 7ième caractère
pont
```

Il n'est pas possible de réaliser des opérations arithmétiques sur des chaînes de caractères. La fonction **float()** permet de convertir **un type str en type float** et la fonction **int()** permet de convertir **un type str en type int**.

fonctions int() et float()

```
>>> '17.45' + 2                     # opération impossible
Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str

>>> float('17.45') + 2              # utilisation de la fonction float()
19.45

>>> int('3') * 2                    # utilisation de la fonction int()
6
```

2.2 – LISTES

Une liste est une structure de données qui peut s'apparenter à un tableau. Chaque élement d'une liste est repéré par **indice** (ou **index**). Le premier élément d'une liste possède **l'indice 0**.

Une liste peut être constituée **d'éléments types différents** : entiers, flottants, chaînes de cracatères voir d'autres listes.

Listes

```
>>> Donnees = ['Dupont','Pierre',17,1.75,72.5]      # liste : str, int et float

>>> print(Donnees)
['Dupont','Pierre',17,1.75,72.5]

>>> print('Nom : ',Donnees[0])           # premier élément indice 0
Nom : Dupont

>>> print('Age : ',Donnees[2])           # troisième élément indice 2
Age : 17

>>> print('Taille : ',Donnees[3])        # quatrième élément indice 3
Taille : 1.75
```

Il est possible de créer des **listes à 2 dimensions**.

Liste à 2 dimensions

```
>>> liste = [[0,1,2],[3,4,5],[6,7,8]]    # liste à 2 dimensions

>>> print(liste)
[[0,1,2],[3,4,5],[6,7,8]]

>>> print(liste[0])          # éléments 1ère ligne
[0,1,2]

>>> print(liste[1][2])       # élément 2nde ligne et 3ième colonne
5
```

Pour créer une liste vide :

Créer une liste vide

```
>>> liste = []

>>> print(liste)
[]
```

Pour ajouter un élément à une liste il est possible d'utiliser la méthode « append ». Les éléments sont ajoutés à la fin de la liste.

Ajouter un élément à une liste – méthode « append »

```
>>> liste = []
>>> print(liste)
[]

>>> liste.append(1)
>>> print(liste)
[1]

>>> liste.append("Bonjour")
>>> print(liste)
[1, 'Bonjour']
```

Pour supprimer un élément d'une liste il est possible d'utiliser la fonction « del » ou la méthode « remove ». La fonction « del » utilise l'indice de l'élément.

Supprimer un élément d'une liste – fonction « del »

```
>>> Donnees = ['Dupont','Pierre',17,1.75,72.5]
>>> print(Donnees)
['Dupont','Pierre',17,1.75,72.5]

>>> del Donnees[1]
>>> print(Donnees)
['Dupont',17,1.75,72.5]
```

La méthode « remove » la valeur de l'élément. S'il y a plusieurs fois la même valeur dans la liste, la méthode « remove » supprime la première valeur rencontrée.

Supprimer un élément d'une liste – méthode « remove »

```
>>> Donnees = ['Dupont','Pierre',17,1.75,72.5]
>>> print(Donnees)
['Dupont','Pierre',17,1.75,72.5]

>>> Donnees.remove[1.75]
>>> print(Donnees)
['Dupont','Pierre',17,72.5]
```

Pour connaître le nombre d'éléments d'une liste il faut utiliser la fonction « len » :

Exemple 18 : Nombre d'éléments d'une liste – fonction « len »

```
>>> Donnees = ['Dupont','Pierre',17,1.75,72.5]
>>> len(Donnees)
5
```

Pour connaître le nombre d'occurrence d'une valeur dans une liste il faut utiliser la méthode « count » :

Nombre d'occurrence – méthode « count »

```
>>> liste = ['a','a','a','b','c','c']
>>> liste.count('a')
3
>>> liste.count('c')
2
```

Pour connaître la position (indice) d'une valeur dans une liste il faut utiliser la méthode « index ». S'il y a plusieurs fois la même valeur dans la liste, la méthode « index » retourne la position de la première valeur rencontrée.

Position d'une valeur – méthode « index »

```
>>> liste = ['a','a','a','b','c','c']
>>> liste.index('b')
3

>>> liste.index('c')
4
```

2.3 – DICTIONNAIRES

Un dictionnaire permet de **stocker des données sous la forme (clé ; valeur)**. Une clé est unique et n'est pas nécessairement un entier.

Dictionnaires

```
>>> moyenne = {'Math':14,'Anglais':12.5,'Français':13}

>>> print(moyenne)                  # tout le dictionnaire
{'Anglais':12.5,'Français':13,'Math':14}

>>> print(moyenne['Math'])          # la valeur qui a pour clé « Math »
14

>>> moyenne['Anglais'] = 16         # nouvelle affectation

>>> print(moyenne)                  # tout le dictionnaire
{'Anglais':16,'Français':13,'Math':14}
```

2.4 – VARIABLES

Une variable est un **espace mémoire** dans lequel il est possible de **stocker une valeur** (une donnée). Il s'agit donc d'un **identifiant associé à une valeur**.

La notion de variable n'existe pas dans le langage Python. On parle plutôt de **référence d'objet**. Il s'agit donc d'une **référence d'objet située à une adresse mémoire**.

On **affecte une variable** par une valeur en utilisant le signe **=**. Dans une affectation, le membre de gauche reçoit le membre de droite.

Affectations simples de variables

```
>>> a = 2                # la variable a reçoit la valeur 2
>>> b = math.sqrt(2)     # la variable b reçoit la valeur racine carrée 2
>>> c = a*b              # la variable c reçoit la valeur de a fois la valeur de b
>>> print(c)
2.8284271247461903
```

Outre l'affectation simple, on peut aussi utiliser les formes suivantes :

Autres formes d'affectations de variables

```
>>> a = 3                    # affectation simple
>>> print(a)
3

>>> a += 3                   # affectation augmentée a = a + 3
>>> print(a)
6

>>> a = b = 7                # affectations multiples
>>> print(a)
7
>>> print(b)
7

>>> a,b = 2.7,5.1            # affectation parallèle de séquences : tuple
>>> print(a)
2.7
>>> print(b)
5.1

>>> a,b,c = ['A','B','C']    # affectation parallèle de séquences : liste
>>> print(a)
A
>>> print(b)
B
>>> print(c)
C
```

3 – ENTREES / SORTIES

3.1 – FONCTION INPUT()

La fonction standard « **input()** » **interrompt le programme** et **attend** que l'utilisateur **entre une donnée et la valide**.

Fonction input()

```
>>> nb_j = input("Nombres de joueurs : ")      # nb_j est une chaîne de caractères
                        >>> nb_j = input("Nombres de joueurs : ")
                         Nombres de joueurs : 2
                        >>>

>>> print(nb_j)
2
>>> print(type(nb_j))         # nb_j est une chaîne de caractères
<class 'str'>

>>> nb_j = int(input("Nombres de joueurs : ")) # nb_j est transtypé en int
                      >>> nb_j = int(input("Nombres de joueurs : "))
                       Nombres de joueurs : 2
                      >>>

>>> print(nb_j)
2
>>> print(type(nb_j))         # nb_j est un entier
<class 'int'>

>>> nb_j = float(input("Nombres de joueurs : "))     # nb_j est transtypé en float
                     >>> nb_j = float(input("Nombres de joueurs : "))
                      Nombres de joueurs : 2
                     >>>

>>> print(nb_j)
2.0
>>> print(type(nb_j))         # nb_j est un flottant
<class 'float'>
```

3.2 – FONCTION PRINT()

La fonction **print()** est indispensable pour l'affichage des résultats.

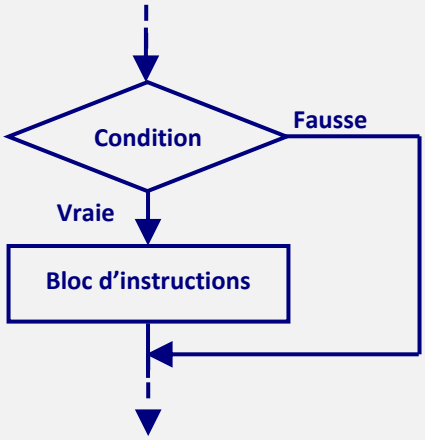
Fonction print()

```
>>> a,b = 2,5
>>> print(a,b)
2 5


>>> a,b = 2,5

>>> print("Somme = ", a + b)
Somme = 7


>>> a,b = 2,5

>>> print("Le produit de ",a," par ",b," vaut : ",a * b)
Le produit de 2 par 7 vaut : 10


>>> print()             # affiche une nouvelle ligne (saut de ligne)
```

4 – STRUCTURES ALTERNATIVES

4.1 – INSTRUCTION IF (SI)

Syntaxe




```
if Condition :
      Bloc d'Instructions
Suite du programme
```

Si la condition est **vraie** (**True**) alors le **bloc d'instructions est exécuté**. **Si** la condition est **fausse** (**False**) on passe directement à la **suite du programme**.

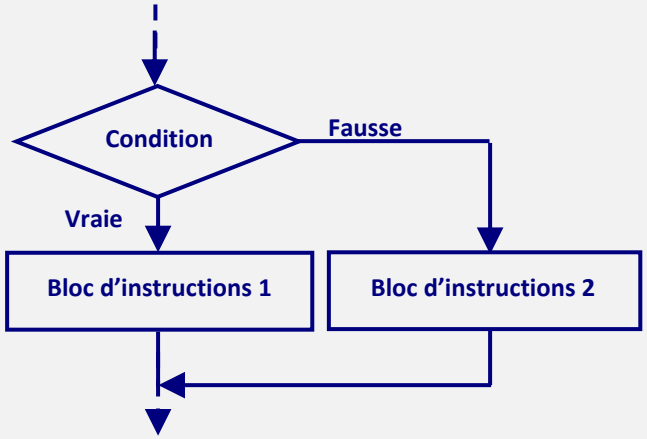
Instruction « if »

```
nb = input("Entrer un nombre plus petit que 100 : ")
nb = float(nb)
if nb < 100 :
      print("Le nombre",nb,"convient")

>>>
Entrer un nombre plus petit que 100 : 50
Le nombre 50.0 convient
```

4.2 – INSTRUCTION ELSE (SINON)

Une instruction « **else** » est toujours associée à une instruction **if**.

Syntaxe




```
if Expression :
      Bloc d'Instructions 1
else :
      Bloc d'Instructions 2
Suite du programme
```

Instruction « else »

```
nb = input("Entrer un nombre plus petit que 100 : ")
nb = float(nb)
if nb < 100 :
      print("Le nombre",nb,"convient")
else :
      print("Le nombre",nb,"est trop grand")

>>>
Entrer un nombre plus petit que 100 : 10
Le nombre 10.0 convient

>>>
Entrer un nombre plus petit que 100 : 120
Le nombre 120.0 est trop grand
```

4.3 – INSTRUCTION ELIF

Dans le cas de **structures alternatives imbriquées**, il est possible d'utiliser une instruction « **elif** » « (contraction de else if).

Syntaxe

```
if Condition 1 :
      Bloc d'Instructions 1
elif Condition 2 :
      Bloc d'Instructions 2
else :
      Bloc d'Instructions 3
Suite du programme
```

Instruction « elif »

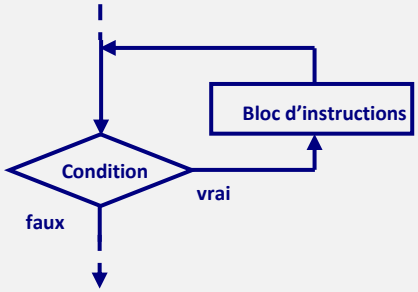
```
nb = input("Entrer un nombre plus petit que 100 : ")
nb = float(nb)
if nb == 100 :
      print("Ce nombre vaut 100")
elif nb == 0 :
      print("Ce nombre est nul")
elif nb > 0 and nb < 100:
      print("Le nombre",nb,"convient")
else :
      print("Le nombre",nb,"est trop grand")

>>>
Entrer un nombre plus petit que 100 : 100
Ce nombre vaut 100

>>>
Entrer un nombre plus petit que 100 : 0
Ce nombre est nul

>>>
Entrer un nombre plus petit que 100 : 20
Le nombre 20.0 convient

>>>
Entrer un nombre plus petit que 100 : 200
Le nombre 200.0 est trop grand
```

5 – BOUCLES – STRUCTURES ITERATIVES

Une **structure itérative** ou **boucle** permet de **répéter une portion de code**.

5.1 – BOUCLE « WHILE » (TANT QUE)

Syntaxe




```
while Condition :
      Bloc d'Instructions
Suite du programme
```

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.

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 !
```

Affichage de l'heure courante avec la boucle « while »

```
import time       # importation du module time
quitter = 'n'     # initialisation de la réponse

while quitter != 'o' :
      # ce bloc est exécuté tant que la condition (quitter != 'o') est vraie
      print("Heure courante",time.strftime('%H:%M:%S'))
      quitter = input("Voulez-vous quitter le programme (o/n) ? ")

# on sort de la boucle
print("A bientôt")

>>>
Heure courante 13:56:25
Voulez-vous quitter le programme (o/n) ? n
Heure courante 13:56:30
Voulez-vous quitter le programme (o/n) ? n
Heure courante 13:56:33
Voulez-vous quitter le programme (o/n) ? o
A bientôt
```

5.2 – BOUCLE « FOR »

Syntaxe

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

La **séquence est parcourue élément par élément**. L'élément peut être de tout type : entier, 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**.

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("Eh 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 avec la boucle « for »

```
>>>
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 !
```

Parcours d'une liste avec une boucle « for »

```
liste = ["Pierre","Dupont",67.5,17]
for element in liste :        # element est la variable d'itération
      print(element)
# on sort de la boucle
print("Fin de la boucle")
```

La variable **liste** est **initialisée avec le premier élément de la séquence** ('**Pierre**'). La boucle est exécutée jusqu'à ce on arrive au dernier élément de la séquence ('**17**').

```
>>>
Pierre
Dupont
67.5
17
Fin de la boucle
```

Remplissage d'une liste avec le même élément

```
liste = ["A" for element in range(0,5)]
print("liste = ",liste)
```

La **liste** contient 5 fois le caractère « A »

```
>>>
liste = ['A', 'A', 'A', 'A', 'A']
```

Parcours d'une chaîne de caractères avec une boucle « for »

```
chaine = "Python"
for lettre in chaine :        # lettre est la variable d'itération
      print(lettre)
# on sort de la boucle
print("Fin de la boucle")
```

La variable **lettre** est **initialisée avec le premier élément de la séquence** ('**I**'). Le bloc d'instructions est alors exécuté. Puis la variable **lettre est mise à jour avec le second élém**ent de la séquence ('**n**') et le bloc d'instructions à nouveau exécuté... La **boucle est exécutée jusqu'à ce on arrive au dernier élément** de la séquence ('**e**').

```
>>>
P
y
t
h
o
n
Fin de la boucle
```

5.3 – INSTRUCTION « BREAK »

L'instruction « **break** » provoque une sortie immédiate d'une boucle « **while** » ou d'une boucle « **for** ».

Instruction « break »

```
import time             # importation du module time

while True :            # l'expression est toujours vraie
      print("Heure courante",time.strftime('%H:%M:%S'))
      quitter = input("Voulez-vous quitter le programme (o/n) ? ")
      if quitter = 'o' :
            break

# on sort de la boucle
print("A bientôt")
```

L'expression « **True** » **est toujours vraie** : il s'agit d'une **boucle sans fin**. L'instruction « **break** » est donc le seul moyen de sortir de la boucle.

```
>>>
Heure courante 09:04:02
Voulez-vous quitter le programme (o/n) ? o
A bientôt
```

6 – FONCTIONS

6.1 – FONCTIONS

Une fonction est une **portion de code** (sorte de sous-programme) que l'on peut appeler au besoin. L'utilisation des fonctions permet :

- **d'éviter la répétition** ;
- de **mettre en relief les données et les résultats** : entrées et sorties de la fonction ;
- la **réutilisation dans d'autres scripts** par l'intermédiaire du mécanisme de l'import ;
- de **décomposer une tâche complexe** en tâches plus simples.

On obtient ainsi des **programmes plus courts et plus lisibles**.

Syntaxe

```
def nomFonction(parametres1,parametre2,parametre3):
      """ Documentation de la fonction.
            ……."""
      bloc_instructions>
      return resultat
```

Fonction « Conversion degrés Celsius en degrés Kelvin »

```
def conv_celsius_kelvin(degres_celsius) :
      """Cette fonction permet de convertir des
            degrés Celsius en degrés Kelvin"""
      degres_kelvin = degres_celsius + 273
      return degres_kelvin

>>> conv_celsius_kelvin(0)
273

>>> conv_celsius_kelvin(-273)
0

>>> conv_celsius_kelvin(30)
303
```

6.2 – PASSAGE DE PARAMETRES

Le **passage de paramètres** permet de **fournir les données** utiles à la fonction. Ce passage s'effectue **lors de l'appel** de la fonction. Il est possible de fournir **plusieurs paramètres** à la fonction. Dans l'exemple précédant, il faut **fournir le paramètre** « degres_celsius » à la fonction pour son exécution.

Fonction « Portion de table de multiplication quelconque »

```
def Table_Mul(table,debut,fin) :
      """---------------------------------------
      cette fonction permet d'afficher une portion
      d'une table de mulitiplication quelconque
      table : table de multiplication attendue
      debut : à partir de quelle valeur
      fin : jusqu'à quelle valeur
      ---------------------------------------"""
      n = debut
      while n <= fin :
            print(n,"*",table,"=",n*table)
            n = n + 1

# programme principal
num_table = int(input("Quelle table voulez-vous ?"))
num_debut = int(input("A partir de quelle valeur ?"))
num_fin = int(input("Jusqu'à quelle valeur ?"))
print("Table de multiplication par",num_table,"de",num_debut,"à",num_fin)
Table_Mul(num_table,num_debut,num_fin)

>>>
Quelle table voulez-vous ? 2
A partir de quelle valeur ? 5
Jusqu'à quelle valeur ? 8
Table de multiplication par 2 de 5 à 8
5 * 2 = 10
6 * 2 = 12
7 * 2 = 14
8 * 2 = 16
```

Dans l'exemple ci-dessus, il fournir les paramètres « table », « debut » et « fin » à la fonction « Table_Mul ». Par contre, le corps d'instruction de la fonction « Table_Mul » **ne contient pas de return**, c'est-à-dire qu'elle ne **retourne pas de résultat**. Il s'agit d'une **procédure**.

Les paramètres passés en arguments peuvent de **types simples** (**int**, **float**, **str**…) mais également de **types plus complexes** (**tuple**, **list**, **dict**…). Il est également possible de passer en argument **d'autres fonctions**.

6.3 – RETOUR DE RESULTATS

L'instruction « return » **stoppe l'exécution** de la fonction et **retourne une ou plusieurs données**.

Fonction « Calcul de la surface et du volume d'une sphère »

```
import math
def surface_volume_sphere(R) :
      """---------------------------------------
      cette fonction calcule et retourne à
      partir du rayon R, la surface S et le
      volume V d'une sphère
      ---------------------------------------"""
      S = 4.0 * math.pi * R**2
      V = S * R / 3
      return S,V
# programme principal
rayon = float(input("Rayon (en cm): "))
s,v = surface_volume_sphere(rayon)
print("Sphère de rayon",rayon,"cm")
print("Sphère de surface",s,"cm²")
print("Sphère de volume",v,"cm3")

>>>
Rayon (en cm): 2
Sphère de rayon 2.0 cm
Sphère de surface 50.26548245743669 cm²
Sphère de volume 33.510321638291124 cm3
```

6.4 – PORTEE DES VARIABLES : VARIABLES LOCALES OU GLOBALES

La portée d'une variable dépend de **l'endroit du programme où on peut accéder à la variable**. Une **variable globale** est visible et utilisable dans **tout le programme**. Une **variable locale** est crée par une fonction et **n'est visible que par cette fonction**. Lors de la **sortie de la fonction**, la variable est **détruite**.

Variable globale, variable locale

```
x = 10                          # variable globale
def ma_fonction() :
      x = 20                    # variable locale
      print("La variable locale est",x)
# programme principal
print("La variable globale est",x)
ma_fonction()

>>>
La variable globale est 10
La variable locale est 20
```

Bien que possédant, le même identifiant, les **deux variables x sont distinctes**.