Gerald Baratoux

Introduction au langage Python

Cours librement adapté à partir du travail de Fabrice Sincère

Contenu

Préambule

En 1989, le hollandais **Guido van Rossum** commence le développement du langage de programmation Python.

Python est un langage **multiplateforme**, c'est-à-dire disponible sur plusieurs architectures (compatible PC, tablettes, smartphones, ordinateur low cost Raspberry Pi...) et systèmes d'exploitation (Windows, Linux, Mac, Android...).

Le langage Python est gratuit, sous **licence libre**.

C'est un des langages informatiques les plus populaires avec C, C++, C#, Objective-C, Java, PHP, JavaScript, Delphi, Visual Basic, Ruby et Perl (liste non exhaustive).

Actuellement, Python en est à sa version 3.
Cependant, la version 2 est encore largement utilisée.
Attention : Python 2 n'est pas compatible avec Python 3 !

Que peut-on faire avec Python ?

Beaucoup de choses !

- du calcul scientifique (librairie `NumPy`)
- des graphiques (librairie `matplotlib`)
- du traitement du son, de la synthèse vocale (librairie `eSpeak`)
- du traitement d'image (librairie `PIL`), de la vision artificielle par caméra (framework `SimpleCV`)
- des applications avec interface graphique GUI (librairies `Tkinter`, `PyQt`, `wxPython`, `PyGTK`...)
- des jeux vidéo en 2D (librairie `Pygame`)
- des applications multi-touch (framework `kivy` pour tablette et smartphone à écran tactile)
- des applications Web (serveur Web `Zope` ; frameworks Web `Flask`, `Django` ; 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 (librairie `PySerial`), en Bluetooth (librairie `pybluez`)...
- ...

Des dizaines de milliers de librairies sont disponibles sur le dépôt officiel PyPI.

I - Variables, types et opérateurs

1 - Le type `int` (integer : nombres entiers)

Pour affecter (on dit aussi assigner) la valeur 17 à la variable nommée `Age` :

```
>>> Age = 17
```

La fonction `print()` affiche la valeur de la variable :

```
>>> print(Age)
17
```

La fonction `type()` retourne le type de la variable :

```
>>> print(type(Age))
<class 'int'>
```

`int` est le type des nombres entiers.

```
>>> # ceci est un commentaire
>>> Age = Age + 1  # en plus court : Age += 1
>>> print(Age)
18
>>> Age = Age - 3  # en plus court : Age -= 3
>>> print(Age)
15
>>> Age = Age * 2  # en plus court : Age *= 2
>>> print(Age)
30
>>> a = 6*3-20
>>> print(a)
-2
>>> b = 25
>>> c = a + 2*b
>>> print(b, c)          # ne pas oublier la virgule
25 48
```

L'opérateur `//` donne la division entière :

```
>>> d = 450//360
>>> print(d)
1
```

L'opérateur `%` donne le reste de la division (opération modulo) :

```
>>> reste = 450 % 360
>>> print(reste)
90
```

L'opérateur `**` donne la puissance :

```
>>> Mo = 2**20
>>> print(Mo)
1048576
```

2 - Le type `float` (nombres en virgule flottante)

```
>>> b = 17.0       # le séparateur décimal est un point (et non une virgule)
>>> print(b)
17.0
>>> print(type(b))
<class 'float'>
>>> c = 14.0/3.0
>>> print(c)
4.66666666667
>>> c = 14.0//3.0  # division entière
>>> print(c)
4.0
```

Attention : avec des nombres entiers, l'opérateur `/` fait une division classique et retourne un type `float` :

```
>>> c = 14/3
>>> print(c)
4.66666666667
```

Notation scientifique :

```
>>> a = -1.784892e4
>>> print(a)
-17848.92
```

3 - Les fonctions mathématiques

Pour utiliser les fonctions mathématiques, il faut commencer par importer le module `math` :

```
>>> import math
```

La fonction `dir()` retourne la liste des fonctions et données d'un module :

```
>>> 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', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Pour appeler une fonction d'un module, la syntaxe est la suivante :
module.fonction(arguments)

Pour accéder à une donnée d'un module :
module.data

```
>>> print(math.pi)          # donnée pi du module math (nombre pi)
3.14159265359
>>> print(math.sin(math.pi/4.0)) # fonction sin() du module math (sinus)
0.707106781187
>>> print(math.sqrt(2.0))   # fonction sqrt() du module math (racine carrée)
1.41421356237
>>> print(math.sqrt(-5.0))
Traceback (most recent call last):
    print math.sqrt(-5.0)
ValueError: math domain error
>>> print(math.exp(-3.0))   # fonction exp() du module math (exponentielle)
0.0497870683679
>>> print(math.log(math.e)) # fonction log() du module math (logarithme népérien)
1.0
```

4 - Le type `str` (string : chaîne de caractères)

```
>>> Nom = 'Dupont' # entre apostrophes
>>> print(Nom)
Dupont
>>> print(type(Nom))
<class 'str'>
>>> Prenom = "Pierre"                # on peut aussi utiliser les guillemets
>>> print(Prenom)
Pierre
>>> print(Nom,Prenom)                # ne pas oublier la virgule
Dupont Pierre
```

La concaténation désigne la mise bout à bout de plusieurs chaînes de caractères.
La concaténation utilise l'opérateur +

```
>>> chaine = Nom + Prenom    # concaténation de deux chaînes de caractères
>>> print(chaine)
DupontPierre
>>> chaine = Prenom + Nom    # concaténation de deux chaînes de caractères
>>> print(chaine)
PierreDupont
>>> chaine = Prenom + ' ' + Nom
>>> print(chaine)
Pierre Dupont
>>> chaine = chaine + ' 18 ans'                # en plus court : chaine += ' 18 ans'
>>> print(chaine)
Pierre Dupont 18 ans
```

La fonction `len()` retourne la longueur (length) de la chaîne de caractères :

```
>>> print(len(chaine))
20
>>> print(chaine[0])        # premier caractère (indice 0)
P
>>> print(chaine[1])        # deuxième caractère (indice 1)
i
>>> print(chaine[1:4])
ier
>>> print(chaine[2:])
erre Dupont 18 ans
>>> print(chaine[-1])       # dernier caractère (indice -1)
s
>>> print(chaine[-6:])
18 ans
>>> chaine = 'Aujourd'hui'
SyntaxError: invalid syntax
>>> chaine  = 'Aujourd\'hui'          # séquence d'échappement \'
>>> print(chaine)
Aujourd'hui
>>> chaine  = "Aujourd'hui"
>>> print(chaine)
Aujourd'hui
```

La séquence d'échappement `\n` représente un saut ligne :

```
>>> chaine = 'Première ligne\nDeuxième ligne'
>>> print(chaine)
Première ligne
Deuxième ligne
```

Plus simplement, on peut utiliser les triples guillemets (ou les triples apostrophes) pour encadrer une chaîne définie sur plusieurs lignes :

```
>>> chaine = """Première ligne
Deuxième ligne"""
>>> print(chaine)
Première ligne
Deuxième ligne
```

On ne peut pas mélanger les serviettes et les torchons (ici type `str` et type `int`) :

```
>>> chaine = '17.45'
>>> print(type(chaine))
<class 'str'>
>>> chaine = chaine + 2
TypeError: Can't convert 'int' object to str implicitly
```

La fonction `float()` permet de convertir un type `str` en type `float`

```
>>> nombre = float(chaine)
>>> print(nombre)
17.45
>>> print(type(nombre))
<class 'float'>
>>> nombre = nombre + 2              # en plus court : nombre += 2
>>> print(nombre)
19.45
```

La fonction `input()` lance une invite de commande (en anglais : prompt) pour saisir une chaîne de caractères.

```
>>> # saisir une chaîne de caractères et valider avec la touche Enter
>>> chaine = input("Entrer un nombre : ")
Entrer un nombre : 14.56
>>> print(chaine)
14.56
>>> print(type(chaine))
<class 'str'>
>>> nombre = float(chaine)   # conversion de type
>>> print(nombre**2)
211.9936
```

5 - Le type `list` (liste)

Une liste est une structure de données.

Le premier élément d'une liste possède l'indice (l'index) 0.

Dans une liste, on peut avoir des éléments de plusieurs types.

```
>>> InfoPerso = ['Pierre' , 'Dupont' , 17 , 1.75 , 72.5]
>>> # la liste InfoPerso contient 5 éléments de types str, str, int, float et float
>>> print(type(InfoPerso))
<class 'list'>
>>> print(InfoPerso)
['Pierre', 'Dupont', 17, 1.75, 72.5]
>>> print('Prénom : ',InfoPerso[0])            # premier élément (indice 0)
Prénom :  Pierre
>>> print('Age : ',InfoPerso[2])               # le troisième élément a l'indice 2
Age :  17
>>> print('Taille : ',InfoPerso[3])            # le quatrième élément a l'indice 3
Taille :  1.75
```

La fonction `range()` crée une liste d'entiers régulièrement espacés :

```
>>> maliste = range(10)
>>> print(list(maliste))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print(type(maliste))
<class 'range'>
>>> maliste = range(1,10,2)  # range(début,fin non comprise,intervalle)
>>> print(list(maliste))
[1, 3, 5, 7, 9]
>>> print(maliste[2])                # le troisième élément a l'indice 2
5
```

On peut créer une liste de listes, qui s'apparente à un tableau à 2 dimensions (ligne, colonne) :

0 1 2
10 11 12
20 21 22

```
>>> maliste = [[0,1,2],[10,11,12],[20,21,22]]
>>> print(maliste[0])
[0, 1, 2]
>>> print(maliste[0][0])
0
>>> print(maliste[2][1])    # élément à la troisième ligne et deuxième colonne
21
>>> maliste[2][1] = 69               # nouvelle affectation
>>> print(maliste)
[[0, 1, 2], [10, 11, 12], [20, 69, 22]]
```

6 - Le type `bool` (booléen)

Deux valeurs sont possibles : `True` et `False`

```
>>> a = True
>>> print(type(a))
<class 'bool'>
```

7 - Les opérateurs de comparaison :

Opérateur	Signification	Remarques
<	strictement inférieur	
<=	inférieur ou égal	
>	strictement supérieur	
>=	supérieur ou égal	
==	égal	Attention : deux signes ==
!=	différent	
is	identique	Deux conditions : égal et même type
is not	non identique	

```
>>> b = 10
>>> print(b>8)
True
>>> print(b==5)
False
>>> print(b!=10)
False
>>> print(0<= b <=20)
True
```

Les opérateurs logiques : `and, or, not`

```
>>> note=13.0
>>> mentionAB = note>=12.0 and note<14.0 # ou bien : mentionAB = 12.0 <= note < 14.0
>>> print(mentionAB)
True
>>> print(not mentionAB)
False
>>> print(note==20.0 or note==0.0)
False
```

L'opérateur `in` s'utilise avec des chaînes (type `str`) ou des listes (type `list`) :

```
>>> chaine = 'Bonsoir'
>>> # la sous-chaîne 'soir' fait-elle partie de la chaîne 'Bonsoir' ?
>>> a = 'soir' in chaine
>>> print(a)
True
>>> print('b' in chaine)
False
>>> maliste = [4,8,15]
>>> # le nombre entier 9 est-il dans la liste ?
>>> print(9 in maliste)
False
>>> print(8 in maliste)
True
>>> print(14 not in maliste)
True
```

8 - Le type `dict` (dictionnaire)

Un dictionnaire stocke des données sous la forme clé ⇒ valeur.

Une clé est unique et n'est pas nécessairement un entier (comme c'est le cas de l'indice d'une liste).

```
>>> moyenne = {'math':12.5,'anglais':15.8}      # entre accolades
>>> print(type(moyenne))
<class 'dict'>
>>> print(moyenne['anglais'])                   # entre crochets
15.8
>>> moyenne['anglais']=14.3           # nouvelle affectation
>>> print(moyenne)
{'anglais': 14.3, 'math': 12.5}
>>> moyenne['sport']=11.0             # nouvelle entrée
>>> print(moyenne)
{'sport': 11.0, 'anglais': 14.3, 'math': 12.5}
```

9 - Autres types

Nous avons vu les types les plus courants.

Il en existe bien d'autres :

- complex (nombres complexes, par exemple 1+2.5j)
- tuple (structure de données)
- set (structure de données)
- file (fichiers)
- ...

10 - Programmation Orientée Objet (POO)

Python est un langage de programmation **orienté objet** (comme les langages C++, Java, PHP, Ruby...).
Une variable est en fait un **objet** d'une certaine **classe**.

Par exemple, la variable `amis` est un objet de la classe `list`.
On dit aussi que la variable `amis` est une **instance** de la classe `list`.
L'**instanciation** (action d'instancier) est la création d'un objet à partir d'une classe (syntaxe : **NouvelObjet = NomdelaClasse(arguments)**) :

```
>>> # instanciation de l'objet amis de la classe list
>>> amis = ['Nicolas','Julie']        # ou bien : amis = list(['Nicolas','Julie'])
>>> print(type(amis))
<class 'list'>
```

Une classe possède des fonctions que l'on appelle **méthodes** et des données que l'on appelle **attributs**.

La méthode `append()` de la classe `list` ajoute un nouvel élément en fin de liste :

```
>>> # instanciation d'une liste vide
>>> amis = []                         # ou bien : amis = list()
>>> amis.append('Nicolas')   # synthase générale : objet.méthode(arguments)
>>> print(amis)
['Nicolas']
>>> amis.append('Julie')     # ou bien : amis = amis + ['Julie']
>>> print(amis)
['Nicolas','Julie']
>>> amis.append('Pauline')
>>> print(amis)
['Nicolas','Julie','Pauline']
>>> amis.sort()                       # la méthode sort() trie les éléments
>>> print(amis)
['Julie', 'Nicolas', 'Pauline']
>>> amis.reverse()           # la méthode reverse() inverse la liste des éléments
>>> print(amis)
['Pauline', 'Nicolas', 'Julie']
```

La méthode `lower()` de la classe `str` retourne la chaîne de caractères en casse minuscule :

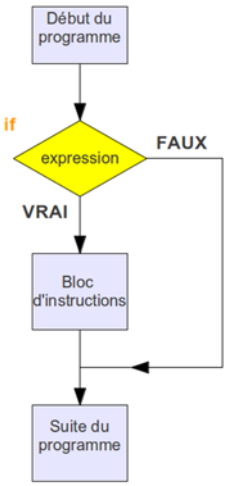
```
>>> # la variable chaine est une instance de la classe str
>>> chaine = "BONJOUR"                # ou bien : chaine = str("BONJOUR")
>>> chaine2 = chaine.lower()# on applique la méthode lower() à l'objet chaine
>>> print(chaine2)
bonjour
>>> print(chaine)
BONJOUR
```

La méthode `pop()` de la classe `dict` supprime une clé :

```
>>> # instanciation de l'objet moyenne de la classe dict
>>> moyenne = {'sport': 11.0, 'anglais': 14.3, 'math': 12.5}
>>> # ou : moyenne = dict({'sport': 11.0, 'anglais': 14.3, 'math': 12.5})
>>> moyenne.pop('anglais')
14.3
>>> print(moyenne)
{'sport': 11.0, 'math': 12.5}
>>> print(moyenne.keys())    # la méthode keys() retourne la liste des clés
dict_keys(['sport', 'math'])
>>> print(moyenne.values()) # la méthode values() retourne la liste des valeurs
dict_values([11.0, 12.5])
```

II - Les conditions

1 - L'instruction `if`



Syntaxe

```
if expression:                          # ne pas oublier le signe de ponctuation ':'
    bloc d'instructions                 # attention à l'indentation
# suite du programme
```

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

```
# script Condition1.py

chaine = input("Note sur 20 : ")
note = float(chaine)
if note>=10.0:
    # ce bloc est exécuté si l'expression (note>=10.0) est vraie
    print("J'ai la moyenne")
print("Fin du programme")
>>>
Note sur 20 : 16
J'ai la moyenne
Fin du programme
>>>
Note sur 20 : 5
Fin du programme
```

2 - L'instruction `else`

Une instruction `else` est toujours associée à une instruction `if`

Syntaxe

```
if expression:
    bloc d'instructions 1            # attention à l'indentation
else:                                        # else est au même niveau que if
    bloc d'instructions 2            # attention à l'indentation
# suite du programme
```

Si l'expression est vraie (True) alors le bloc d'instructions 1 est exécuté.
Si l'expression est fausse (False) alors c'est le bloc d'instructions 2 qui est exécuté.

```
# script Condition2.py

chaine = input("Note sur 20 : ")
note = float(chaine)
if note>=10.0:
    # ce bloc est exécuté si l'expression (note>=10.0) est vraie
    print("J'ai la moyenne")
else:
    # ce bloc est exécuté si l'expression (note>=10.0) est fausse
    print("C'est en dessous de la moyenne")
print("Fin du programme")
>>>
Note sur 20 : 15
J'ai la moyenne
Fin du programme
>>>
Note sur 20 : 8.5
C'est en dessous de la moyenne
Fin du programme
>>>
Note sur 20 : 56
J'ai la moyenne
Fin du programme
```

Pour traiter le cas des notes invalides (<0 ou >20), on peut imbriquer des instructions conditionnelles :

```
# script Condition3.py

chaine = input("Note sur 20 : ")
note = float(chaine)
if note>20.0 or note<0.0:
    # ce bloc est exécuté si l'expression (note>20.0 or note<0.0) est vraie
    print("Note invalide !")
else:
    # ce bloc est exécuté si l'expression (note>20.0 or note<0.0) est fausse
    if note>=10.0:
        # ce bloc est exécuté si l'expression (note>=10.0) est vraie
        print("J'ai la moyenne")
    else:
        # ce bloc est exécuté si l'expression (note>=10.0) est fausse
        print("C'est en dessous de la moyenne")
print("Fin du programme")
>>>
Note sur 20 : 56
Note invalide !
Fin du programme
>>>
Note sur 20 : 14.6
J'ai la moyenne
Fin du programme
```

On ajoute encore un niveau d'imbrication pour traiter les cas particuliers 0 et 20 :

```
# script Condition4.py

chaine = input("Note sur 20 : ")
note = float(chaine)
if note>20.0 or note<0.0:
    print("Note invalide !")
else:
    if note>=10.0:
        print("J'ai la moyenne")
        if note==20.0:
            # ce bloc est exécuté si l'expression (note==20.0) est vraie
            print("C'est même excellent !")
    else:
        print("C'est en dessous de la moyenne")
        if note==0.0:
            # ce bloc est exécuté si l'expression (note==0.0) est vraie
            print("... lamentable !")
print("Fin du programme")
>>>
Note sur 20 : 20
J'ai la moyenne
C'est même excellent !
Fin du programme
>>>
Note sur 20 : 3
C'est en dessous de la moyenne
Fin du programme
```

3 - L'instruction `elif`

Une instruction `elif` (contraction de else if) est toujours associée à une instruction `if`

Syntaxe

```
if expression 1:
    bloc d'instructions 1
elif expression 2:
    bloc d'instructions 2
elif expression 3:
    bloc d'instructions 3    # ici deux instructions elif, mais il n'y a pas de limitation
else:
    bloc d'instructions 4
# suite du programme
```

Si l'expression 1 est vraie alors le bloc d'instructions 1 est exécuté, et on passe à la suite du programme.
Si l'expression 1 est fausse alors on teste l'expression 2 :

si l'expression 2 est vraie on exécute le bloc d'instructions 2, et on passe à la suite du programme.

si l'expression 2 est fausse alors on teste l'expression 3, etc...

Le bloc d'instructions 4 est donc exécuté si toutes les expressions sont fausses (c'est le bloc "par défaut").

Parfois il n'y a rien à faire.
Dans ce cas, on peut omettre l'instruction `else` :

```
if expression 1:
    bloc d'instructions 1
elif expression 2:
    bloc d'instructions 2
elif expression 3:
    bloc d'instructions 3
# suite du programme
```

L'instruction `elif` évite souvent l'utilisation de conditions imbriquées (et souvent compliquées).

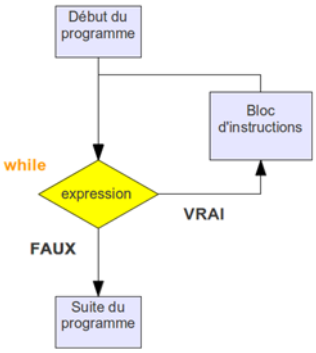
Exemple

```
# script Condition5.py
# ce script fait la même chose que Condition4.py

note = float(input("Note sur 20 : "))
if note==0.0:
    print("C'est en dessous de la moyenne")
    print("... lamentable !")
elif note==20.0:
    print("J'ai la moyenne")
    print("C'est même excellent !")
elif note<10.0 and note>0.0: # ou bien : elif 0.0 < note < 10.0:
    print("C'est en dessous de la moyenne")
elif note>=10.0 and note<20.0:        # ou bien : elif 10.0 <= note < 20.0:
    print("J'ai la moyenne")
else:
    print("Note invalide !")
print("Fin du programme")
>>>
Note sur 20 : 20
J'ai la moyenne
C'est même excellent !
Fin du programme
>>>
Note sur 20 : 3
C'est en dessous de la moyenne
Fin du programme
>>>
Note sur 20 : 77
Note invalide !
Fin du programme
```

III - Les boucles

Une boucle permet d'exécuter une portion de code plusieurs fois de suite.

1 - L'instruction `while`



Syntaxe

```
while expression:           # ne pas oublier le signe de ponctuation ':'
    bloc d'instructions              # attention à l'indentation
# suite du programme
```

Si l'expression est vraie (`True`) le bloc d'instructions est exécuté, puis l'expression est à nouveau évaluée.
Le cycle continue jusqu'à ce que l'expression soit fausse (`False`) : on passe alors à la suite du programme.

Exemple : un script qui compte de 1 à 4

```
# script Boucle1.py

# initialisation de la variable de comptage
compteur = 1
while compteur<5:
    # ce bloc est exécuté tant que la condition (compteur<5) est vraie
    print(compteur, compteur<5)
    compteur += 1    # incrémentation du compteur, compteur = compteur + 1
print(compteur<5)
print("Fin de la boucle")
>>>
1 True
2 True
3 True
4 True
False
Fin de la boucle
```

Table de multiplication par 8

```
# script Boucle2.py

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
print("Et voilà !")
>>>
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
Et voilà !
```

Affichage de l'heure courante

```
# script Boucle3.py

import time     # importation du module time
quitter = 'n'   # initialisation
while quitter != 'o':
    # ce bloc est exécuté tant que la condition est vraie
    # strftime() est une fonction du module time
    print('Heure courante ',time.strftime('%H:%M:%S'))
    quitter = input("Voulez-vous quitter le programme (o/n) ? ")
print("A bientôt")
>>>
Heure courante  09:48:54
Voulez-vous quitter le programme (o/n) ? n
Heure courante  09:48:58
Voulez-vous quitter le programme (o/n) ? o
A bientôt
```

2 - L'instruction `for`

Syntaxe

```
for élément in séquence :
    bloc d'instructions
# suite du programme
```

Les éléments de la séquence sont issus d'une chaîne de caractères ou bien d'une liste.

Exemple avec une séquence de caractères

```
# script Boucle4.py

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

La variable `lettre` est initialisée avec le premier élément de la séquence (`'B'`).
Le bloc d'instructions est alors exécuté.
Puis la variable `lettre` est mise à jour avec le second élément de la séquence (`'o'`) et le bloc d'instructions à nouveau exécuté...
Le bloc d'instructions est exécuté une dernière fois lorsqu'on arrive au dernier élément de la séquence (`'r'`) :

```
>>>
B
o
n
s
o
i
r
Fin de la boucle
```

Exemple avec les éléments d'une liste

```
# script Boucle5.py

maliste = ['Pierre',67.5,18]
for element in maliste:
    print(element)
print("Fin de la boucle")
```

Là, on affiche dans l'ordre les éléments de la liste :

```
>>>
Pierre
67.5
18
Fin de la boucle
```

Fonction range()

L'association avec la fonction `range()` est très utile pour créer des séquences automatiques de nombres entiers :

```
# script Boucle6.py
print(list(range(1,5)))

for i in range(1,5):
    print(i)
print("Fin de la boucle")
>>>
[1, 2, 3, 4]
1
2
3
4
Fin de la boucle
```

Table de multiplication

La création d'une table de multiplication paraît plus simple avec une boucle `for` qu'avec une boucle `while` :

```
# script Boucle7.py

for compteur in range(1,11):
    print(compteur, '* 9 =', compteur*9)
print("Et voilà !")
>>>
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
10 * 9 = 90
Et voilà !
```

3 - L'instruction `break`

L'instruction `break` provoque une sortie immédiate d'une boucle `while` ou d'une boucle `for`.

Dans l'exemple suivant, l'expression `True` est toujours ... vraie : on a une boucle sans fin.
L'instruction `break` est donc le seul moyen de sortir de la boucle.

Affichage de l'heure courante

```
# script Boucle8.py

import time     # importation du module time
while True:
    # strftime() est une fonction du module time
    print('Heure courante ',time.strftime('%H:%M:%S'))
    quitter = input('Voulez-vous quitter le programme (o/n) ? ')
    if quitter == 'o':
        break
print("A bientôt")
>>>
Heure courante  14:25:12
Voulez-vous quitter le programme (o/n) ? n
Heure courante  14:25:20
Voulez-vous quitter le programme (o/n) ? o
A bientôt
```

Astuce

Si vous connaissez le nombre de boucles à effectuer, utiliser une boucle `for`.
Autrement, utiliser une boucle `while` (notamment pour faire des boucles sans fin).

IV - Les fonctions

Nous avons déjà vu beaucoup de fonctions : print(), type(), len(), input(), range()...
Ce sont des fonctions pré-définies (built-in functions).
Nous avons aussi la possibilité de créer nos propres fonctions !

Intérêt des fonctions

Une fonction est une portion de code que l'on peut appeler au besoin (c'est une sorte de sous-programme).

L'utilisation des fonctions évite des redondances dans le code : on obtient ainsi des programmes plus courts et plus lisibles.

Par exemple, nous avons besoin de convertir à plusieurs reprises des degrés Celsius en degrés Fahrenheit :

```
>>> print(100.0*9.0/5.0+32.0)
212.0
>>> print(37.0*9.0/5.0+32.0)
98.6
>>> print(233.0*9.0/5.0+32.0)
451.4
```

La même chose en utilisant une fonction :

```
>>> def F(DegreCelsius):
        print(DegreCelsius*9.0/5.0+32.0)
```

```
>>> F(100)
212.0
>>> F(37)
98.6
>>> x = 233
>>> F(x)
451.4
```

Rien ne vous oblige à définir des fonctions dans vos scripts, mais cela est tellement pratique qu'il serait improductif de s'en passer !

L'instruction `def`

Syntaxe

```
def NomDeLaFonction(parametre1,parametre2,parametre3,...):
    """ Documentation
    qu'on peut écrire
    sur plusieurs lignes """ # docstring entouré de 3 guillemets (ou apostrophes)

    bloc d'instructions              # attention à l'indentation

    return resultat          # la fonction retourne le contenu de la variable resultat
```

Exemple n°1

```
# script Fonction1.py

def MaPremiereFonction():     # cette fonction n'a pas de paramètre
    """ Cette fonction affiche 'Bonjour' """
    print("Bonjour")
    return                    # cette fonction ne retourne rien ('None')
                              # l'instruction return est ici facultative
```

Une fois la fonction définie, nous pouvons l'appeler :

```
>>> MaPremiereFonction()      # ne pas oublier les parenthèses ()
Bonjour
```

L'accès à la documentation se fait avec la fonction pré-définie `help()` :

```
>>> help(MaPremiereFonction) # affichage de la documentation
Help on function MaPremiereFonction in module __main__:

MaPremiereFonction()
    Cette fonction affiche 'Bonjour'
```

Exemple n°2

La fonction suivante simule le comportement d'un dé à 6 faces.
Pour cela, on utilise la fonction `randint()` du module `random`.

```
# script Fonction2.py

def TirageDe():
    """ Retourne un nombre entier aléatoire entre 1 et 6 """
    import random
    valeur = random.randint(1,6)
    return valeur
>>> print(TirageDe())
3
>>> print(TirageDe())
6
>>> a = TirageDe()
>>> print(a)
1
```

Exemple n°3

```
# script Fonction3.py

# définition des fonctions
def Info():
    """ Informations """
    print("Touche q pour quitter")
    print("Touche Enter pour continuer")

def TirageDe():
    """ Retourne un nombre entier aléatoire entre 1 et 6 """
    import random
    valeur = random.randint(1,6)
    return valeur

# début du programme
Info()
while True:
    choix = input()
    if choix == 'q':
        break
    print("Tirage :",TirageDe())
>>>
Touche q pour quitter
Touche Enter pour continuer

Tirage : 5

Tirage : 6
q
>>>
```

Exemple n°4

Une fonction avec deux paramètres :

```
# script Fonction4.py

# définition de fonction
def TirageDe2(valeurMin,valeurMax):
    """ Retourne un nombre entier aléatoire entre valeurMin et valeurMax """
    import random
    return random.randint(valeurMin,valeurMax)

# début du programme
for i in range(5):
    print(TirageDe2(1,10))    # appel de la fonction avec les arguments 1 et 10
>>>
6
7
1
10
2
>>>
```

Exemple n°5

Une fonction qui retourne une liste :

```
# script Fonction5.py

# définition de fonction
def TirageMultipleDe(NbTirage):
    """ Retourne une liste de nombres entiers aléatoires entre 1 et 6 """
    import random
    resultat = [random.randint(1,6) for i in range(NbTirage)]     # compréhension de listes
(Cf. annexe)
    return resultat

# début du programme
print(TirageMultipleDe(10))
>>>
[4, 1, 3, 3, 2, 1, 6, 6, 2, 5]
>>> help(TirageMultipleDe)
Help on function TirageMultipleDe in module __main__:

TirageMultipleDe(NbTirage)
    Retourne une liste de nombres entiers aléatoires entre 1 et 6
```

Exemple n°6

Une fonction qui affiche la parité d'un nombre entier.

Il peut y avoir plusieurs instructions `return` dans une fonction.

L'instruction `return` provoque le retour immédiat de la fonction.

```
# script Fonction6.py

# définition de fonction
def Parite(nombre):
    """ Affiche la parité d'un nombre entier """
    if (nombre % 2) == 1:   # L'opérateur % donne le reste d'une division
        print(nombre,'est impair')
        return
    if (nombre % 2) == 0:
        print(nombre,'est pair')
        return

# début du programme
Parite(13)
Parite(24)
>>>
13 est impair
24 est pair
```

Portée de variables : variables globales et locales

La portée d'une variable est l'endroit du programme où on peut accéder à la variable.

Observons le script suivant :

```
a = 10         # variable globale au programme

def MaFonction():
    a = 20     # variable locale à la fonction
    print(a)
    return
>>> print(a)          # nous sommmes dans l'espace global du programme
10
>>> MaFonction()      # nous sommes dans l'espace local de la fonction
20
>>> print(a)          # de retour dans l'espace global
10
```

La variable `a` de valeur 20 est créée dans la fonction : c'est une variable locale à la fonction.
Elle est détruite dès que l'on sort de la fonction.

L'instruction `global` rend une variable globale :

```
a = 10         # variable globale

def MaFonction():
    global a   # la variable est maintenant globale
    a = 20
    print(a)
    return
>>> print(a)
10
>>> MaFonction()
20
>>> print(a)
20
```

Remarque : il est préférable d'éviter l'utilisation de l'instruction `global` car c'est une source d'erreurs (on peut modifier le contenu d'une variable sans s'en rendre compte, surtout dans les gros programmes).

V – Installer des bibliothèques avec PyPI

PyPI, pour PYthon Package Index, est en quelque sorte un serveur centralisé de paquets Python. Il centralise les différents modules/paquets mis à disposition par la communauté et en gère les différentes versions.

L'outil Pip de Python va vous permettre d'installer des modules/paquets depuis le serveur PyPI officiel, en ligne, depuis un serveur PyPI local, ou encore depuis un dossier spécifique que vous stipulerez.

Il existe plusieurs versions de PyPI, pour Python2 et Python3

Vous pourrez donc les utiliser ainsi dans une invite de commande permettant l'exécution de python
pip install <nom_du_module>
pip3 install <nom_du_module>

Voici les commandes disponibles avec l'outil PyPI :

install	Permet d'installer un paquet
uninstall	Permet de désinstaller un paquet
freeze	Permet d'afficher, ou de créer un fichier contenant l'ensemble des paquets installés, au format requirement
list	Permet de lister les paquets installés sur la machine
show	Permet d'afficher des informations à propos d'un paquet donné
search	Permet de rechercher un paquet, selon un mot clé, sur PyPI (local ou non)
wheel	Crée un fichier Wheel à partir du fichier requirement indiqué (nécessite parfois un paquet supplémentaire)
help	Permet d'afficher l'aide

VI - Introduction au format JSON

Citation : Wikipédia

JSON (*JavaScript Object Notation*) est un format de données textuel, générique, dérivé de la notation des objets du langage ECMAScript. Il permet de représenter de l'information structurée.

Euh… J'ai lu « JavaScript Object Notation *». Pourquoi venir nous parler de JavaScript dans un tutoriel concernant Python ?*

C'est quoi cette histoire ?

En fait, JSON est un format permettant de représenter des données. On peut comparer son usage à celui du XML.

Tout comme le XML, le JSON peut être lu par de nombreux langages de programmation (33 à l'heure actuelle) dont le Python et même… l'humain !

En effet, il s'agit d'un format texte et contrairement au format binaire (**utilisé par *pickle***), vous pourrez voir et modifier facilement des valeurs avec un quelconque éditeur de texte.

Présentation du format JSON par l'exemple

Je vais ici vous présenter deux exemples de fichiers représentant les caractéristiques d'une *playlist* : l'un en XML et l'autre en JSON.

Voici sa représentation « humaine ».

La *playlist* nommée **MeshowRandom** est composée de :

- *Best Improvisation Ever 2* de **David Meshow** avec une note de **5/5** ;

- *My Theory* de **David Meshow** avec une note de **4/5**.

Voyons maintenant sa représentation « informatique » en examinant les fichiers correspondants à cette description.

Tout d'abord, en XML !

Le format de balisage XML est très répandu (le zCode est dérivé du XML !).

```xml
<?xml version="1.0" ?>
<playlist nom="MeshowRandom">
  <chanson>
    <titre>Best Improvisation Ever 2</titre>
    <auteur>David Meshow</auteur>
    <note>5</note>
  </chanson>
  <chanson>
    <titre>My Theory (Bonus)</titre>
     <auteur>David Meshow</auteur>
     <note>4</note>
   </chanson>
</playlist>
```

Et en JSON, ça donne quoi ?

Voici les mêmes informations contenues dans un fichier JSON.

```
{
  "nom" : "MeshowRandom",
  "chansons" : [
    {
      "titre" : "Best Improvisation Ever 2",
      "auteur" : "David Meshow",
      "note" : 5
    },
    {
      "titre" : "My Theory",
      "auteur" : "David Meshow",
      "note" : 4
    }
  ]
}
```

Bilan

À partir de cette comparaison, on peut clairement voir émerger l'avantage principal du format JSON par rapport au format XML : **il est minimaliste**.

En effet, JSON se base plus sur un modèle clé/valeur que sur un format de balisage. Cela permet d'éviter les balises de fermeture, la répétition et cela peut éventuellement représenter un gain de place sur de très gros fichiers (16 caractères économisés ici en comptant les espaces, soit 95% de la taille originale).
Attention cependant : le format JSON ne possède pas de spécification **concernant les commentaires**.
Vous pouvez en rajouter de la même manière qu'en JavaScript avec // Commentaire ou /* Commentaire */ mais rien ne garantit qu'ils seront pris en compte et ils pourraient (dans le pire des cas) faire planter votre programme selon votre parseur.
Dans le doute, **évitez les commentaires dans vos fichiers JSON** histoire de rester conforme à la norme officielle.

Comment créer un objet JSON en python

```
import json

data = { 'a':'A', 'b':(2, 4), 'c':3.0 }
print ('DATA:',data)

data['f'] = 2.4
print ('JSON', json.dumps(data))
```

```
DATA: {'a': 'A', 'c': 3.0, 'b': (2, 4)}
JSON {"a": "A", "c": 3.0, "b": [2, 4], "f": 2.4}
```

Pour aller plus loin :

```
import json

data = {"nom":"MeshowRandom",
        "chansons":[
                    {"Titre":"Best Improvisation Ever 2","auteur":"David Meshow","note":5},
                    {"Titre":"My Theory","auteur":"David Meshow","note":4}
                    ]
        }

print ('DATA:', data)
data["chansons"][1]["Titre"] = "Nouveau Titre !!!"

jsonData = json.dumps(data)
print ('JSON', jsonData)
```

```
DATA: {'nom': 'MeshowRandom', 'chansons': [{'note': 5, 'Titre': 'Best Improvisation Ever 2', 'auteur':
'David Meshow'}, {'note': 4, 'Titre': 'My Theory', 'auteur': 'David Meshow'}]}
JSON {"nom": "MeshowRandom", "chansons": [{"note": 5, "Titre": "Best Improvisation Ever 2", "auteur":
"David Meshow"}, {"note": 4, "Titre": "tutu", "auteur": "David Meshow"}]}
```

VII - Introduction à la manipulation des fichiers Excel

Il existe de nombreux packages Python disponibles pour manipuler les fichiers Excel. Ils fonctionnent sous toutes les plateformes et ne nécessitent pas l'utilisation de Windows ou d'Excel.
Voici un exemple d'utilisation de Openpyxl

```
from openpyxl import Workbook

wb = Workbook()       #Création du classeur
ws = wb.active        #On récupère la première feuille
ws.title = "nomFeuille"
#si on veut créer une nouvelle feuille wb.create_sheet("Mysheet")
irow = 2              #On commence à écrire à partir de la 2ème ligne
for val in range(10):
    ws.cell(row=irow,column=1,value=val)    #Ecriture dans la case
    #ws['A2'] = val
    irow+=1



wb.save("export.xlsx")       #Sauvegarde dans le fichier export.xlsx
```

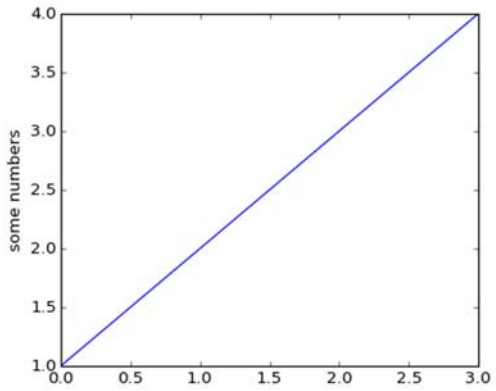
Utilisation des couleurs et bordures :

```
from openpyxl import Workbook
from openpyxl.styles import Border, Side, Font, colors, borders

wb = Workbook()       #Création du classeur
ws = wb.active        #On récupère la première feuille
ws.cell(row=1,column=1,value="coucou")

ft = Font(color=colors.RED, bold=True)

bd = Border(left=Side(border_style=borders.BORDER_THICK),
            right=Side(border_style=borders.BORDER_THICK),
            top=Side(border_style=borders.BORDER_THICK),
            bottom=Side(border_style=borders.BORDER_THICK))

ws.cell(row=1,column=1).font = ft
ws.cell(row=1,column=1).border = bd


wb.save("export.xlsx")       #Sauvegarde dans le fichier export.xlsx
```

VIII – Tracer une courbe avec MatPlotLib

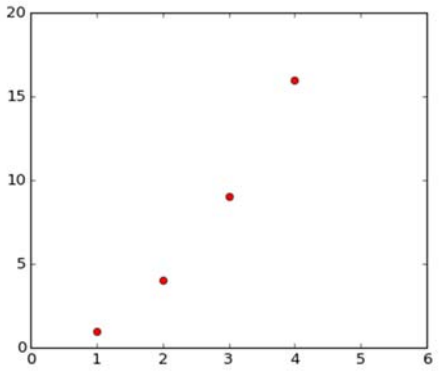
Vous pourrez trouver un tutoriel avec le lien suivant :
http://matplotlib.org/users/pyplot_tutorial.html

Voici un exemple de tracé simple :




```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.savefig('courbe.png')
plt.show()
```

ou si l'on a un tableau pour les axes X et Y :




```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16])
plt.axis([0, 6, 0, 20])
plt.show()
```

IX – Développer une application serveur aves Flask

http://flask.pocoo.org/

Flask est une librairie permettant de développer un site web ou un serveur de requête sans retour graphique.

Voici quelques exemples permettant de comprendre son fonctionnement

Nous allons définir les routes et y associer une fonction qui sera appelée à chaque fois qu'une requête sera envoyée :

Exemple pour la route principale :

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')


if __name__ == '__main__':
    app.run(debug=True)
```

Une route /coucou :

```
@app.route('/coucou')
def coucou():
    return "coucou salut  !"
```

Une route avec deux paramètres :

```
@app.route('/somme/<valeur1>/<valeur2>')
def somme(valeur1,valeur2):
    resultat = valeur1 + valeur 2
    Return str(resultat)
```

Une route qui trace et affiche une courbe :

```
@app.route('/courbe')
def courbe():
    plt.plot([1, 2, 3, 4])
    plt.ylabel('some numbers')
    figfile = BytesIO()
    plt.savefig(figfile, format='png')
    figfile.seek(0)
    figdata_png = base64.b64encode(figfile.getvalue()).decode('ascii')

    return render_template("courbe.html",imgCourbe=figdata_png)
```

X – Envoyer une information sur le réseau (socket UDP)

Les deux principales methodes pour envoyer des informations entre des ordinateurs sont les protocoles TCP (mode connecté) et le protocole UDP (mode non connecté)

Nous allons nous attarder sur le mode UDP.

Voici deux exemples (extrêmement simples et non sécurisés) vous montrant comment envoyer et recevoir des informations en utilisant les sockets UDP

Le client (pour envoyer des données) :

```
import socket
import json

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

host = '127.0.0.1';
port = 8888;
msg = 'Salut !'

data = [ { 'a':'A', 'b':(2, 4), 'c':3.0 } ]

JSonData = json.dumps(data)
s.sendto(JSonData.encode('utf-8'), (host, port))
```

Le serveur (pour recevoir des données) :

```
import socket
import json

UDP_IP = "127.0.0.1"
UDP_PORT = 8888

sock = socket.socket(socket.AF_INET, # Internet
socket.SOCK_DGRAM) # UDP
sock.bind((UDP_IP, UDP_PORT))

data, addr = sock.recvfrom(8096) # buffer size is 8096 bytes
JSonData = json.loads(data.decode('utf-8'))
JSonData[0]['f'] = 2.4
print (JSonData)
```