# **Python**

Python est un langage interprété.

Lors de l’exécution, l’interpréteur va réellement lire, comprendre et effectuer les instructions décrites dans le code.

Par opposition aux langages compilés qui sont traduit dans des langages intermédiaires plus bas niveau avant d’être exécuté. Exemple c et c++ sont compilé en .o (binaire) ou java compilé en .class

## 0. Commentaires

Pour ajouter des commentaires sur une ligne, on utilise #.

Sur plusieurs lignes, on utilise « « « pour ouvrir les commentaires et « « « pour les fermés.

*Exemple* :

*#Commentaire sur une seule ligne*

*"""*

*Ce commentaire peut être*

*sur plusieurs lignes.*

*"""*

## 1. La déclaration des variables.

Une **variable** est une zone de la mémoire de l'ordinateur 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, c'est-à-dire d'une zone particulière de la mémoire.

On utilise des variables pour stocker en mémoire une valeur que l’on réutilisera plus tard (par exemple le résultat, un compteur du nombre d’itération). Il est préférable d’utiliser une variable pour stocké les expressions que l’on calculera.

### 1.1 Déclaration, Assignation, Affectation

*Exemple* :

*var = 3 #déclare une variable nommé var et de type int (entier)*

Cette opération déclare une variable nommé var et lui assigne la valeur 3. La déclaration de la variable et son initialisation sont donc faite en même temps. *Contrairement à d’autre langage (C).*

L’affectation ne peut s’effectuer que dans ce sens, 3=var ne peut pas marcher car cela reviendrait à essayer de créer une variable appelé 3 et lui assigné la valeur de var.

Par convention, un nom de variable commence toujours par une lettre minuscule, mais il peut y avoir des majuscules à l’intérieur du nom.

*nomDeVariable = -5 #int (entier)*

Au moment de l’initialisation, python devine le type que devra avoir notre variable.

*Exemple* :

*var = 1 #int (entier)*

*var = -5 #int (entier)*

*var = 2.2 #float (double)*

*var = "Help" #str (string = chaine de caractère)*

Note on utilise le « . » pour marquer la virgule des flottants.

La virgule quant à elle est un séparateur (entre paramètres d'une fonction, élément d'une liste, variables à déclarer, etc.)

Contrairement à d’autre langage un type n’est pas associé à var. I.e. qu’on peut toujours associer un autre type de valeur a var en faisant var="azerty".

· Affectation multiple :

En utilisant la virgule, on peut déclarer plusieurs variables d’un seul coup.

*var1, var2 = 1 , 8 #var1 vaut 1 et var2 vaut 8*

*Remarque* :

On peut utiliser la notation scientifique pour créer des variables.

*var = 1e6 #var vaut 1000000*

### 1.2 Type

Le **type** d'une variable correspond à la nature de celle-ci. Les trois principaux types dont nous aurons besoin dans un premier temps sont les entiers (*integer* ou *int*), les nombres décimaux que nous appellerons *floats* et les chaînes de caractères (*string* ou *str*). Bien sûr, il existe de nombreux autres types (par exemple, les booléens, les nombres complexes, etc.)

En python, on peut multiplier, additionner ou soustraire des types différents, mais attention au type du résultat.

Pour vérifier le type de votre résultat vous pouvez utiliser la fonction type de la façon suivante type(maVariable).

*Exemple* :

*name='name'*

*var=0*

*other=0.*

*print(type(name)) # donne <class 'str'>*

*print(type(var)) # donne <class 'int'>*

*print(type(other)) # donne <class 'float'>*

## 1.3 Operateurs de base

### 1.3.1 Opérations de base sur les entiers et flottants

Lorsque l’on effectue des opérations entre deux paramètres, il est important de savoir de quel type sera le résultat.

+ (addition) - (soustraction)

\* (multiplication) / division

// quotient % modulo

\*\* puissance ^ (xor binaire)

Attention, parfois le résultat peut surprendre, par exemple, la division avec / renvoi toujours un float. (8/2 à 2.0)

*Exemple* :

*2\*\*4 #donne 16*

*15//4 #donne 3*

*15%6 #donne 3*

|  |  |  |
| --- | --- | --- |
|  | Résultat | Type |
| 8 % 5 |  |  |
| 8 // 5 |  |  |
| 8 / 5 |  |  |
| 8 + 5.0 |  |  |
| 8 / 4 |  |  |

Pour vérifier ce résultat, on peut entrer le code suivant (à compléter) :

*print(type(2 × 3.4)) # donne <class ' ? '>*

*print(type(2 × 2)) # donne <class ' ? '>*

*print(type(2.5 × 2)) # donne <class ' ? '>*

### 1.3.2 Opérateur « raccourci »

Pour simplifier le code, plutôt que d’écrire à chaque fois :

*var = var + 1*

*var = var \* 2*

*var = var / 8*

On peut écrire plus simplement :

*var += 1*

*var \*= 2*

*var /= 8*

*Exemple*:

*print(3\*"Salut") #donne SalutSalutSalut*

*print("Salut"+"ATous") #donne SalutATous*

### 1.3.3 Opérations illicites

Attention également, on ne peut pas faire n’importe quoi.

Par exemple :

*3 + "Salut"*

Va donner l’erreur suivante :

TypeError: unsupported operand type(s) for +: 'int' and 'str'

Cette opération n’est pas reconnue. De même les chaines de caractères ne peuvent pas être multipliées entre elle ni avec des flottant.

### 1.3.3 Opérateurs de comparaison

== égale à

!= différent de

> supérieur à

>= supérieur ou égal à

< inférieur

<= inférieur ou égal

Ces opérations vont renvoyer des booléens, c’est-à-dire un résultat qui vaudra : Vrai ou Faux (*True* or *False*)

***Attention : ne pas confondre = pour l’affectation est == pour la comparaison.***

On peut aussi comparer des chaines de caractères.

‘toto’ < ‘tata’

### 1.3 Portée des variables.

### 1.3.1 Variables locales.

Les variables définies dans une fonction sont appelées variables locales. Elles ne peuvent être utilisées que localement c’est-à-dire qu’à l’intérieur de la fonction qui les a définies. Tenter d’appeler une variable locale depuis l’extérieur de la fonction qui l’a définie provoquera une erreur.

def fonction ():

    var = 10 #variable locale

    return var

print(var) # -> Erreur car var n'est plus définie

Cela est dû au fait que chaque fois qu’une fonction est appelée, Python réserve pour elle (dans la mémoire de l’ordinateur) un nouvel espace de noms (c’est-à-dire une sorte de dossier virtuel). Les contenus des variables locales sont stockés dans cet espace de noms qui est inaccessible depuis l’extérieur de la fonction.

Cet espace de noms est automatiquement détruit dès que la fonction a terminé son travail, ce qui fait que les valeurs des variables sont réinitialisées à chaque nouvel appel de fonction.

### 1.3.2 Variables globales.

Les variables définies dans l’espace global du script, c’est-à-dire en dehors de toute fonction sont appelées des variables globales. Ces variables sont accessibles (= utilisables) à travers l’ensemble du script et accessible en lecture seulement à l’intérieur des fonctions utilisées dans ce script.

var = 10

def fonction ():

    var = 20 #variable locale

fonction()

print(var) # -> var vaut toujours 10

Une fonction va donc pouvoir utiliser la valeur d’une variable définie globalement mais ne va pas pouvoir modifier sa valeur c’est-à-dire la redéfinir. En effet, toute variable définie dans une fonction est par définition locale ce qui fait que si on essaie de redéfinir une variable globale à l’intérieur d’une fonction on ne fera que créer une autre variable de même nom que la variable globale qu’on souhaite redéfinir mais qui sera locale et bien distincte de cette dernière.

## 1.3.3 Modifier une variable globale depuis une fonction

Dans certaines situations, il serait utile de pouvoir modifier la valeur d’une variable globale depuis une fonction, notamment dans le cas où une fonction se sert d’une variable globale et la manipule.

Cela est possible en Python. Pour faire cela, il suffit d’utiliser le mot clef global devant le nom d’une variable globale utilisée localement afin d’indiquer à Python qu’on souhaite bien modifier le contenu de la variable globale et non pas créer une variable locale de même nom.

var = 10

def fonction ():

    global var

    var = 20 #variable locale

fonction()

print(var) # -> Cette fois var vaut bien 20

## 1.3.4 Portée des variables définies dans des boucles.

En Python, les variables définis à l’intérieur d’une boucle sont toujours accessible à la sortie de cette boucle. Elles valent dans ce cas la dernière valeur qui leur a été assignés.

*Exemple*:

*for i in range(3):*

*var=i*

*print(var)*

Ce programme va afficher : 2 car à la fin de la boucle, var vaudra 2.

En revanche les variables définis à l’intérieur d’une fonction ne sont accessible qu'à l'intérieur de celle-ci.

*def somme(a,b) :*

*c = a + b*

*return c*

*print(c)*

Va donner lieu à l’erreur suivante :

NameError: name 'c' is not defined

### 1.4 Nomenclature

Pour les noms de variables. On commence TOUJOURS par une minuscule. Avec une majuscule au "milieu" si la variable correspond à plusieurs mots.

On essaye également de lui donner un nom le plus clair possible pour se souvenir de ce que cette variable contient.

Et si possible en anglais (pas toujours nécessaire pour ce cours).

*Exemple*:

name, result, maxHealth, currentHeatlh, damage, critPercent.

Les abréviations sont acceptés du moment que le nom reste compréhensible, donc essayé d’éviter les isbp pour abréger le nom d’une variable inventorySortedByWeight.

La plupart des IDE proposent de l'auto-complétion donc pas besoin de raccourcir les noms des variables.

### 3.3 La fonction help et autres fonctions

<https://docs.python.org/fr/3.6/library/functions.html>

Ou utiliser la fonction help(input())

### 3.4 Création de fonction

Pour la création de fonctions, on utilise le mot clé def suivi du nom de la fonction puis entre parenthèse des paramètres utilisés d’entré de cette fonction et enfin « : » .

Les sorties de cette fonction seront ce que retournera le return.

*Exemple* :

def function(a) :

instruction...

...

return x

Ou si on veut que notre fonction renvoi plusieurs éléments différents :

def function(a,b,…) :

contenu

…

return x , y , …

### 3.4.1 Paramètres / Inputs

Les paramètres d’une fonction sont ces données d’entrées. Une fonction peut avoir 0 ou plusieurs paramètre.

Si l’on appelle la fonction triple avec en paramètre 10, le résultat de cette fonction sera 30.

### 3.4.2 Return

Return correspond à la sortie d’une fonction, c’est-à-dire le résultat que cette fonction va renvoyer.

Dans le cas de la fonction « triple », le nombre à renvoyer sera le nombre récupéré en entrée multiplié par 3.

On écrira cette fonction de la façon suivante :

*def triple(nombre):*

*return nombre\*3*

## 4. Listes (et Tableaux)

### 4.1 Liste

Une liste est une structure de données dynamique qui contient plusieurs valeurs. En python, les valeurs d’une liste peuvent être de type différent.

La taille d’une liste n’est pas figé, on peut toujours ajouter ou supprimer des éléments (contrairement aux tableaux).

### 4.1.1 Déclaration

On peut déclarer et initialiser une liste de la façon suivante :

*Exemple* :

colors = ['bleu', 'jaune', 'rouge']

Ou initialiser à la liste vide :

colors=[]

On peut également créer une liste en utilisant la fonction range() et la fonction list().

range(a,b,p) va générer une liste d’entier entre a et b espacé d’un pas p.

maList=list(range(0, 5)) # donne [0, 1, 2 ,3 ,4]

[0, 1, 2, 3, 4]

maList=list(range(0,10,2)) # donne [0, 2, 4 ,6 ,8]

Marche également avec un pas négatif:

maList=list(range(10,0,-2)) # donne [10, 8, 6 ,4 ,2]

### 4.1.2 Utilisation

Pour accéder à l’ième élément d’une liste nommé « liste », on peut utiliser liste[i].

Erreur fréquente, si on essaye d’aller chercher le 3ème élément de la liste colors, il faut faire colors[2] et non pas colors[3]

### 4.1.3 Operations sur les listes

#### **4.1.3.1 Append**

Pour ajouter un élément à une liste on utilise la méthode append().

*Exemple* :

*couleurs.append(‘violet’)*

#### **4.1.3.2 Insert**

Pour insérer un élément à une position donné :

liste.insert(1, 22) # insère l'élément 22 à la position 1

#### **4.1.3.3 Extends**

*Pour ajouter une liste à la fin d’une autre liste.*

liste.extend([5, 6]) # concatène la liste [5, 6] à la fin de la

#### **4.1.3.4 Addition / concaténation**

On peut concaténer deux listes avec +.

ages=[3,5]

colors= ['bleu', 'jaune', 'rouge']

colors= colors +ages

*colors contiendra alors ['bleu', 'jaune', 'rouge', 3, 5]*

#### **4.1.3.4 Remove et del**

Del et remove permettent toute les deux de supprimer un élément d’une liste mais n’utilise pas les mêmes informations pour trouver l’élément à supprimer.

del liste[1] # suppression de l'élément d'indice 1

Del supprime un élément à partir de son indice.

l.remove(68)

Remove supprime le premier élément de la liste dont la valeur correspond.

*Attention* :

Erreur si cet élément n’est pas dans la liste.

à ValueError: list.remove(x): x not in list

#### **4.1.3.3 Duplication**

On peut aussi dupliquer la liste avec \* (comme pour les string, en effet, les string sont équivalent à une liste de caractère)

colors = 3\* colors

#### **4.1.3.4 Interval**

Pour accéder à une sous partie d’une liste : *list[i,j]*

Pareil avec un pas différent : *list[i,j,p]*

On peut donc aussi faire *couleurs[::2]*

#### **4.1.3.4 Taille**

Pour connaitre la taille d’une liste, on utilise : *len(list)*

colors= ['bleu', 'jaune', 'rouge']

len(colors) # donne 3

### *Exercice* :

1. Liste des multiples de 7 entre 0 et 100

Affichez la table de multiplication par 7 jusqu’à 100 en une seule commande avec les instructions range() et list().

### 4.1.3 Listes de listes

On peut facilement créer une liste de liste de la manière suivante.

list1=[]

list2=[]

list3=[]

listOfList=[list1,list2,list3]

*Note*:

Dans beaucoup d’autres langages, on différentie les listes des tableaux. Les tableaux ont une taille fixe, on peut accéder à un élément en faisant tab [i]. Contrairement au tableau la taille de la liste peut varier mais on ne peut pas accéder à ses éléments via un indice. Python à fait un mix de ces deux types en un seul donc.

## 5. Les boucles

Les boucles sont utilisées lorsque l’on veut répéter plusieurs fois une instruction.

### 5.1 La boucle Pour (For)

On utilise les boucles for lorsque l’on sait à avance combien d’itération on va devoir faire.

#### **5.1.1 Sur une liste**

On peut itérer directement sur le contenu d’une liste en faisant:

for color in colors:

print(color)

Dans cet exemple, color est la variable d’itération. Elle va prendre successivement l’ensemble de valeur de colors.

Ne pas oublier les « : » à la fin de la première ligne du for ainsi que l’identation pour le contenu de la boucle, sinon -> IdentationError.

#### **5.1.2 Avec la fonction Range**

for i in range(10) :

print(i)

#### **5.1.3 Avec les indices**

Pour faire comme dans 5.1.1 mais en ayant acces à l’indice de l’élément dans la liste, on peut faire :

for i in range(len(colors)) :

print(“La couleur {} est {}”.format(i,colors[i]))

Autre manière, avec enumerate

for i,color in enumerate(colors):

print(“La couleur {} est {}”.format(i,color))

### 5.2 La boucle Tant que (While)

La boucle « tant que » effectue l’instruction tant que la condition sera valide

*Exemple*:

i=0

while i<10 :

i=i+1

print(i)

Attention :

Quand on utilise cette commande, il faut être sûr que notre condition d’arrêt sera bien atteinte.

Sinon, on rentre dans une boucle infinie, donc le programme ne s’arrête jamais.

### 5.3 Mot clé pour les boucles

Pour sortir d’une boucle, on peut utiliser le mot clé break. On sort alors immédiatement de la boucle et on reprend l’exécution du code après le for.

*Exemple*:

for i in range(5):

if i > 2:

break

print(i)

Pour passer à l’itération suivante, sans exécuter le reste du code interne de cette boucle, on utilise le mot clé « continue ».

for i in range(5):

if i == 2:

continue

print(i)

## 6. Tests

### 6.1 If Elif Else

If = Si

Elif =else if = sinon, si …

Else = Sinon

Le *if* sert à effectuer une instruction plutôt qu’une autre (ou que rien) lorsqu’une certaine condition est respecté.

*Exemple* :

x = 2

if x == 2:

print("x vaut 2.")

else:

print("x ne vaut pas 2.")

x = 3

if x == 2:

print("Le test est vrai !")

else:

print("Le test est faux !")

On peut également tester plusieurs conditions les unes à la suite des autres avec elif

*Exemple :*

import random

base = random.choice(["red", "green", "blue", "yellow"])

if base == "red":

print("choix du rouge")

elif base == "blue":

print("choix du bleu")

elif base == "green":

print("choix du vert")

elif base == "yellow":

print("choix du jaune")

### 6.1.2 Opérateurs boolean

#### **And**

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **A and B** |
| TRUE | TRUE | TRUE |
| TRUE | FALSE | FALSE |
| FALSE | TRUE | FALSE |
| FALSE | FALSE | FALSE |

#### **Or**

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **A or B** |
| TRUE | TRUE | TRUE |
| TRUE | FALSE | TRUE |
| FALSE | TRUE | TRUE |
| FALSE | FALSE | FALSE |

#### 

#### **Not**

|  |  |
| --- | --- |
| **A** | **Not (A)** |
| TRUE | FALSE |
| FALSE | TRUE |

### 6.1.3 A éviter

*Comparaison de flottant :*

Attention, il ne fait pas comparer de flottant car les résultats d’une opération renvoyant des flottants peuvent être stocké différemment même s’ils étaient censé être égaux.

*Explication* :

Comme on l’a vu, les nombres décimaux, float, string, sont tous convertis en binaire pour le pc.

Mais il existe une infinité d’entier, on ne peut donc pas tous les représenté sur un nombre fini d’octet. Python a choisi de représenter les entiers sur 4 octets, donc on peut seulement représenter les nombres entre -2 147 483 648 et 2 147 483 647.

De même pour les flottants, il existe une infinité de nombre entre 0 et 1. IL est donc impossible de stocker précisément tous les nombres sur un nombre fini d’octet.

Du coup, on peut avoir le problème suivant :

*print(3-2.7==0.3)* donne False.

*print(3-2.7)* donne 0.29999999998

<https://fr.wikiversity.org/wiki/Python/Les_types_de_base>

*EXERCICE*

1. Faire une table (liste de liste) de 10 par 10 avec le résultat de la multiplication.

0 0 0 0 …

1 2 3 4 …

2 4 6 8 …

3 6 9 12 …

## 7. Itératif et récursif

Pour exécuter plusieurs fois une instruction, on a vu que l’on pouvait utiliser les boucles.

C’est ce qu’on appelle la programmation itérative.

Il existe une deuxième méthode, un peu plus complexe à appréhender, la programmation récursive.

Un fonction ou un programme récursif est un programme qui va s'appeler lui même sur un paramètre “réduit” par rapport au paramètre initial.

*Exercice*:

Factorielle :

a) Ecrire une fonction de calcul de factorielle en programmation itérative.

b) Ecrire une fonction de calcul de factorielle en programmation récursive.

Recherche plus grand élément :

a) Ecrire une fonction de recherche du plus grand élément d’une liste en programmation itérative.

b) Ecrire une fonction de recherche du plus grand élément d’une liste en programmation récursive.

## 8. Les modules

Le langage python, par défaut, ne propose pas des solutions pour tous les problèmes. Par contre il offre la possibilité d'utiliser des modules pour cela.

Pour des besoins hyper spécifiques (ex modules pour Biologie), mais aussi pour des choses plus ordinaire (ex. Module random)

<https://docs.python.org/fr/3/py-modindex.html>

Les module les plus « courants » en Python sont:

[*math*](https://docs.python.org/fr/3/library/math.html#module-math) : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).

[*sys*](https://docs.python.org/fr/3/library/sys.html#module-sys) : interaction avec l'interpréteur Python, passage d'arguments (cf. plus bas).

[*os*](https://docs.python.org/fr/3/library/os.html#module-os) : dialogue avec le système d'exploitation (cf. plus bas).

[*random*](https://docs.python.org/fr/3/library/random.html#module-random) : génération de nombres aléatoires.

[*time*](https://docs.python.org/fr/3/library/time.html#module-time) : accès à l'heure de l'ordinateur et aux fonctions gérant le temps.

[*urllib*](https://docs.python.org/fr/3/library/urllib.html#module-urllib) : récupération de données sur internet depuis Python.

[*Tkinter*](https://docs.python.org/fr/3/library/tkinter.html#module-tkinter) : interface python avec Tk. Création d'objets graphiques (cf. chapitre 20 *Fenêtres graphiques et Tkinter*).

[*re*](https://docs.python.org/fr/3/library/re.html#module-re) : gestion des expressions régulières (cf. chapitre 16 *Expressions régulières et* parsing\*

*PyGame*: module pour faire des jeux vidéo

### 8.1 Utilisation

Pour utiliser un module existant, il suffit d’ajouter au début de son fichier

*Exemple*:

import random

pour ensuite pouvoir utiliser les fonctions définies dans random.

*Exemple:*

import random

var=random.random()

print(var)

Pour voir les fonctions que l'on peut utilisé avec le module random, on peut consulter la doc:

https://docs.python.org/3/library/random.html

### 8.2 Définition

Pour définir soit même un module, il suffit de sauvegarder son programme dans un fichier \*\*\*.py. Et de stocker ce fichier soit dans le répertoire courant, soit dans un répertoire prédéfini.

## 9. Exceptions

Python permet également de définir des exceptions.

Exemple pour renvoyer une exception si on demande à l’utilisateur de saisir un entier mais que celui ci entre une chaine de caractère, on peut faire:

while True:

try:

x = int(input("Please enter a number: "))

break

except ValueError:

print("Oops! That was no valid number. Try again...")