



PROGRAMMER EN  
PYTHON

Bonjour à toi cher/chère lecteur/lectrice !

Dans cette présentation/cours, je vais  
t'apprendre les bases d'un langage  
informatique, celles du langage PYTHON !

# MAIS C'EST QUOI LE LANGAGE PYTHON ?

Bonne question !

Python c'est un langage informatique, dérivé du langage C, un autre langage informatique. En résumé, c'est du langage C simplifié.

Il diffère du C de par sa simplicité, et du non besoin de compiler le script.

# COMPILER LE SCRIPT WTF ?

Compiler oui. C'est une action qui traduit un programme écrit en langage évolué en un programme équivalent en langage machine (tiré de Larousse.fr). Vous vous rappelez du gcc -o script script.c en langage C qu'il fallait taper dans le terminal avant de pouvoir l'exécuter ? Et bah plus besoin ! Parce que ... c'est du C simplifié !

Mais pas seulement ...



You will see many other elements in C that you  
will no longer or almost never use.

## ENFIN BREF !

Tu es prêt(e) ?

On commence !

# AH NON PAS ENCORE !

Avant de débuter je te/vous conseille d'ouvrir un terminal et éditeur de texte (gedit par exemple) pour pouvoir essayer les manipulations qui seront présentées dans les diapos. Je vous proposerai des exercices à la fin de chaque partie pour vous entraîner ! Et je vous conseille vivement de les faire. Je ferais souvent le lien avec le langage C (si cela peut aider).

# Le terminal d'un côté, et l'éditeur de texte d'un autre !

The screenshot shows a dual-pane interface on a Linux desktop. On the left is a terminal window titled "romain@Hp-debian: ~/Documents". The terminal's menu bar includes "Fichier", "Édition", "Affichage", "Rechercher", "Terminal", and "Aide". The command "romain@Hp-debian:~/Documents\$ |" is visible at the bottom of the terminal window. On the right is a code editor window titled "script.py" located at "/Documents". The code editor has a toolbar with "Ouvrir" (Open), a new file icon, and "Enregistrer" (Save). The main pane displays the following Python script:

```
1 #!/usr/bin/python
2 #-*- coding: utf-8 -*-
3
4
5
6
7
8
9
10
```

The status bar at the bottom of the code editor shows "Python" and "Largeur des tabulations : 8". The status bar also indicates "Lig 10, Col 1" and has a "INS" key indicator.

En python, les fichiers scripts possèdent l'extension « .py ». Ajoutez donc « .py » à la fin du nom du fichier dans lequel vous codez.

Vous avez sans doute remarqué les deux lignes tout au début du fichier texte, avec un # devant. C'est ce qu'on appelle des shebangs (un shebang). Il indique au système d'exploitation que le fichier n'est pas un fichier binaire mais un script (ensemble de commandes). Sur la même ligne est précisé l'interpréteur permettant d'exécuter ce script.

Dans l'illustration ci-dessus, le premier shebang permet de spécifier au système d'exploitation que vous codez en python.

```
#!/usr/bin/python
```

Le deuxième shebang permet de ne pas tenir compte des caractères ASCII.

```
-*- coding: utf-8 -*-
```

Note : pour insérer des commentaires, mettre un # devant la ligne, ou 3 guillemets au-dessus puis 3 autres en-dessous du bloc à mettre en commentaires ou de part et d'autre.

# EXÉCUTION DU PROGRAMME

Pour lancer votre programme vous devez vous déplacer dans le dossier dans lequel est votre fichier et taper dans le terminal :

```
python nom_du_fichier.py
```

```
romain@Hp-debian:~/Documents$ python script.py|
```

Grâce au shebang **`#!/usr/bin/python`**, vous pouvez exécuter votre programme en tapant seulement `./nom_du_fichier`. Vous devez auparavant taper `chmod +x nom_du_fichier.py`.

```
romain@Hp-debian:~/Documents$ chmod +x script.py|
```

**chmod** (pour change mode) change les permissions d'accès au fichier.  
**+x** permet l'exécution.

```
romain@Hp-debian:~/Documents$ ./script.py|
```



Ça y est ? Vous avez tout et êtes prêt(e)s ?

C'est parti !

# SOMMAIRE

- Avant de réellement commencer .....	13
- Variables et types .....	15
- Fonctions diverses .....	29
- Listes et tuples .....	70
- Dictionnaires .....	111
- Opérateurs logiques/Conditions .....	133/144
- Boucles .....	160
- Définition de fonction .....	175
- Modules/Bibliothèques .....	208
- Messages d'erreurs fréquents .....	241
- Manipuler les erreurs .....	264
- Pour aller plus loin .....	279

# AVANT DE RÉELLEMENT COMMENCER

Pour coder vous avez deux choix :

- Soit vous codez directement dans le terminal
- Soit vous codez dans un éditeur de texte et dans ce cas, vous pourrez sauvegarder votre script (conseillé)

Pour coder dans le terminal, assurez-vous d'avoir installé python.  
Sinon installez-le, je ne vous dirai pas comment, cherchez un peu !



**Allez on commence !**

**Enfin !**

# LES VARIABLES

C'est quoi une variable ?

C'est un emplacement mémoire que vous réservez ... dans la mémoire pour y mettre quelque chose. Cet emplacement mémoire sera désigné par une variable que vous nommerez bien sûr, pour la manipuler ensuite. C'est la base de la base de tout langage informatique, la variable !

# NOMMER LA VARIABLE ?

Noms possibles :

a, b, ax, trentesix, gti\_izer, truc4, lenomdevariabledelamortquitue,  
to\_ta\_ti, LserLserLd, ...

Noms non possibles :

36quaidesorfevres, ?!\*^§/.?. , 31654, \_\_\_\_-, élk0@, ...

# POUR RÉSUMER

- Vous n'avez pas le droit de mettre de la ponctuation dans vos variables, sauf pour le underscore (ou tiret bas) « \_ ».
- Un nom de variable ne peut pas commencer par un chiffre.
- Les majuscules peuvent faire toute la différence.
- Pas de lettre avec accent.

Exemple : « Truc » et « truc » sont deux variables différentes.

- Dans le cas où le nom de la variable que vous avez choisi n'est pas acceptable, une erreur sera affichée dans le terminal.

# COMPARAISON AVEC LE C

Si vous aviez fait du C auparavant, sachez qu'il fallait la déclarer avant de pouvoir l'utiliser (int truc, float machin, char blabla, ...).

En python plus besoin de déclaration. Il suffit simplement d'entrer le nom de votre variable et d'y mettre quelque chose.

# ON MET QUOI DEDANS ?

Quelque chose, un élément, un objet, quasi n'importe quoi. On peut y mettre en entier, un réel, un caractère, une chaîne de caractères, une liste, un dictionnaire, etc. Bref, quelque chose.

Mais c'est quoi tout ça ?

Bonne question ! Attends un peu !

# AFFECTATION

Quand on met quelque chose dans une variable, on la « crée ». Plus précisément, on « affecte » le quelque chose dans la variable.

## Comment on fait ?

# BONNE QUESTION !

Avec le « = »

Exemple :

JeMetsDansLaVariableLeChiffreUn = 1

Autre exemple :

JeContientUneChaineDeCaracteres = 'tototatatititutu'

```
JeMetsDansLaVariableLeChiffreUn = 1  
JeContientUneChaineDeCaracteres = 'tototatatititutu'
```

# ATTENTION !

Ne pas confondre « = » avec le symbole mathématique  $=$ . On utilise le « = » pour mettre quelque chose dans une variable. Pour le  $=$  signifiant une égalité, ce sera « == ». On le verra dans la partie Opérateurs logiques, ainsi que d'autres trucs utiles et sympas.

## AUTRE FAÇON D'AFFECTER

Dans le cas où vous voudriez affecter plusieurs variables à la fois, en une ligne, vous pouvez séparer les noms des variables et les éléments à affecter par une virgule de chaque côté du « = » comme suit :

`mot1, mot2, mot3 = 'Coucou', 'ça', 'va'`

```
mot1, mot2, mot3 = 'Coucou', 'ça', 'va'
```

Dans le cas où vous décideriez d'affecter plusieurs éléments à une variable, celle-ci se transformera en une liste/tuple. Retenez juste que c'est un ensemble d'éléments. On approfondira ce cas plus loin.

Exemple :

**phrase** = 'coucou', 'ça', 'va'

```
phrase = 'Coucou', 'ça', 'va'
```

# LES TYPES !!!!!

Un type c'est ... un type, de quelque chose (quelques exemples : fille, garçon, monstre, sorcier, robot, virtuel, imaginaire, etc.). Il y en a beaucoup ! En voici quelque-uns en programmation :

`int, float, str, list, dic, tuple, array, ...`

Je ne vais pas tous vous les lister sinon on y passera la nuit !

# COMPARAISON AVEC LE C

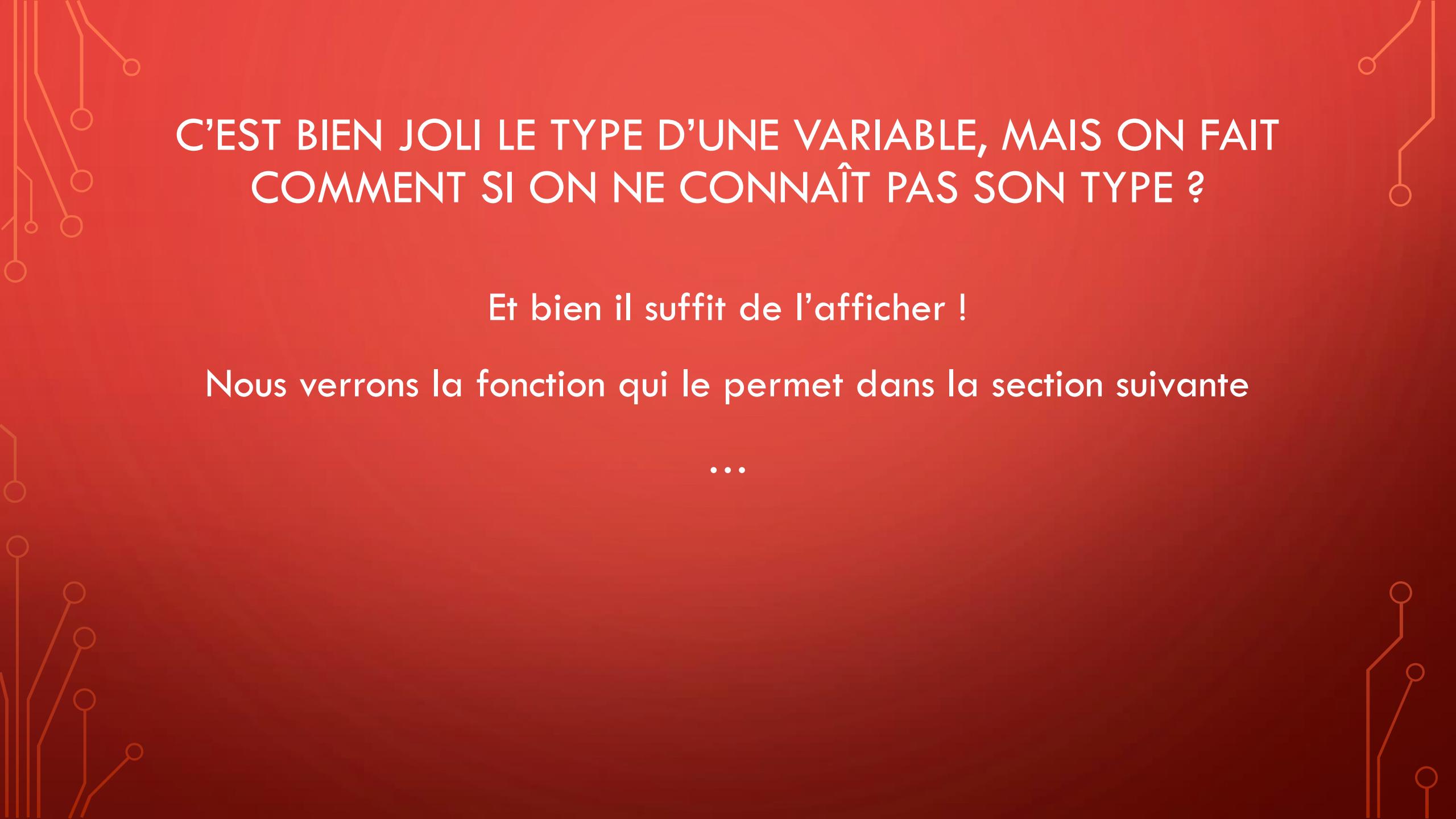
Rappelez-vous qu'en C vous deviez déclarer la variable avant de l'utiliser. Par exemple :

```
int c;
```

Là, vous avez créé la variable `c`, et lui avez donné un type, le type `int` (entier, un nombre entier).

## EN PYTHON

En python plus besoin de faire cette déclaration ! En C lorsque vous avez besoin d'une déclaration, il fallait la déclarer au début du programme (de préférence). En python, lorsque que vous avez besoin d'une variable, il suffit de lui donner un nom et d'y mettre quelque chose, de n'importe quel type, même au beau milieu de votre programme si vous voulez !



C'EST BIEN JOLI LE TYPE D'UNE VARIABLE, MAIS ON FAIT  
COMMENT SI ON NE CONNAÎT PAS SON TYPE ?

Et bien il suffit de l'afficher !

Nous verrons la fonction qui le permet dans la section suivante

...



LA SECTION SUIVANTE !

# Fonctions diverses

# FONCTIONS

C'est quoi une fonction ?

À votre avis ?

$f(x)$ ,  $g(x)$ ,  $h(x)$ , ...

Non toujours pas ?

# C'EST ...

Un truc ou quelque chose qui prend en entrée quelque chose (une variable, une liste, une chaîne de caractères, ...) et effectue des opérations dessus, ou non.

Prenons l'exemple de la station service. Considérons une voiture. C'est l'objet donné entrée de la fonction. Considérons la machine qui possède plusieurs rouleaux qui tournent et nettoient la voiture. C'est la fonction. Vous donnez la voiture à la machine et on aura en sortie la voiture toute propre ! On peut associer à la machine la fonction « nettoie ».

On peut reconnaître une fonction de par la présence des « () » après « un mot » (exemple : la fonction « dormir() »). Le paramètre donné en entrée de la fonction est inséré dans les parenthèses. C'est comme le  $f(x)$ . Pour  $x = 2$ , il suffit de remplacer  $x$  par 2 dans les parenthèses. On obtient :  $f(2)$ .

Voilà ! Maintenant vous savez reconnaître une fonction ! Python a plusieurs petites fonctions déjà toutes prêtes à l'emploi. Pas besoin d'importer un module/bibliothèque (pour l'instant). On traitera les modules/bibliothèques dans la partie correspondante.

## QUELQUES FONCTIONS DE BASE

En C, vous deviez importer une bibliothèque à chaque fois, la bibliothèque **stdio.h** (standard input/output, soit entrée/sortie).

En python, la bibliothèque input/output contenant les fonctions standards est déjà importée par défaut. Pas besoin d'importer quoi que ce soit, on peut utiliser les fonctions directement.

# LA FONCTION POUR AFFICHER QUELQUE CHOSE

**print( ) ou print** (équivalent **printf( )** en C)

La fonction **print** permet d'afficher ce qu'il y a dans une variable ou quelque chose qu'on lui donne en paramètre. Pour afficher plusieurs variables à la suite, les séparer par des « , ».

Cas particulier pour cette fonction : il n'y a pas besoin de parenthèses sur mandriva, ubuntu, debian, ... (sur mac peut être).

Mais essayons-là !

## script.py

```
numerol = 1
mot = 'Bonjour'

print numerol
print mot
print 'Turlututu'
print 3690

print numerol, mot, 9789687, 'dsfkljdslkfj'
```

## Terminal

```
1
Bonjour
Turlututu
3690
1 Bonjour 9789687 dsfkljdslkfj
```

# INTERACTION AVEC L'UTILISATEUR

**input( ) ou raw\_input( )** (équivalent **scanf( )** en C).

Permet de prendre une valeur entrée par l'utilisateur et de la stocker dans une variable.  
Conséquence : il faut l'affecter à une variable.

**input( )** ne prend qu'un élément de type int ou float.

**raw\_input( )** peut prendre un élément de type int, float ou str. Mais la variable en sortie sera de type **string** (str). Il vous faudra la convertir pour changer son type.

Lorsque vous utilisez la fonction, vous pouvez spécifier ce qui doit s'afficher dans le terminal lorsque la fonction est appelé. Et ce, sous forme d'une chaîne de caractères (str), que vous insérerez dans les parenthèses de la fonction, avec des « ».

# EXAMPLE !

script.py

```
untruc = raw_input()
n = input("Un nombre : ")
lettre = raw_input("Une lettre : ")
phrase = raw_input("Une phrase : ")

print untruc
print n
print lettre
print phrase
```

Terminal

```
sqlfkj76
Un nombre : 76
Une lettre : z
Une phrase : Bonjour à toi
sqlfkj76
76
z
Bonjour à toi
```

# AFFICHER LE TYPE D'UNE VARIABLE

`type( )`

Très simple ! Insérez dans les parenthèses la variable dont vous souhaitez connaître le type. Utilisez ensuite un print pour afficher le type, ou affectez le à une variable.

# EXAMPLE !

```
type_int = 14
type_float = 1.2
type_str = 'coucou'
unknown_type = type(1868768681)

print type(type_int)
print type(type_float)
print type(type_str)
print unknown_type
```

```
<type 'int'>
<type 'float'>
<type 'str'>
<type 'int'>
```

# CHANGER LE TYPE D'UNE VARIABLE

`int( ), float( ), str( ), ... ((int)variable, (float)variable, ... en C)`

Encore très simple ! Vous devez mettre dans les parenthèses la variable dont vous souhaitez changer le type, puis affecter tout cela dans la même variable ou une autre.

# ENCORE DES EXEMPLES !

```
nombre = 76
caracteres = 890

print nombre
print type(nombre)
print caracteres
print type(caracteres)

nombre = float(nombre)
caracteres = str(caracteres)

print nombre
print type(nombre)
print caracteres
print type(caracteres)
```

```
76
<type 'int'>
890
<type 'int'>
76.0
<type 'float'>
890
<type 'str'>
```

# LONGUEUR DE CE QU'IL Y A DANS UNE VARIABLE

`len( ) (strlen( ) en C)`

Super simple ! Insérez encore toujours dans les parenthèses la variable dont vous souhaitez connaître la longueur. Utilisez un print pour afficher la longueur, ou l'affecter à une variable (avec le « = » !!!).

# TOUJOURS UN EXEMPLE

```
mot = 'bonjour'  
UneChaine = 's1fkjklcfgdfxkgjdfnk87hbkn546bkn879Vbkjb767'  
  
print len(mot)  
print len(UneChaine)
```

```
7
```

```
43
```

# GÉNÉRER UNE LISTE DE CHIFFRES ORDONNÉES

## `range( )`

Nous n'avons pas encore vu ce qu'était une liste. Retenez juste que c'est un ensemble de variables.

Cette fonction permet de générer une liste de chiffres allant de 0 à au nombre que vous insérerez dans les parenthèses, moins un (exclu).

Vous pouvez également spécifier le pas pour lequel vous voulez générer les nombres.

Par défaut, le pas est de 1, le début est 0.

Syntaxe :

```
range(début, fin (exclue))  
range(début, fin (exclue), pas)
```

# EXEEEEEEEEEAMPLE !!!

```
VingtElements = range(20)
DebutFin = range(10, 20)
AvecUnPasPositif = range(0, 10, 2)
AvecUnPasNegatif = range(10, 0, -1)

print range(10)
print DebutFin
print VingtElements
print AvecUnPasPositif
print AvecUnPasNegatif
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 2, 4, 6, 8]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

# INVERSER

## `reversed( )`

Fonction très pratique si vous voulez faire quelque chose en partant du dernier au lieu du premier. Si vos nombres sont ordonnés par ordre croissant, alors l'application de cette fonction va ranger vos nombres par ordre décroissant. De même pour une chaîne de caractères, va l'inverser.

Cependant, si vous essayez d'afficher le résultat de cette fonction, celle-ci va vous donner son type, et son adresse, et non pas le résultat en lui-même (les pointeurs en C ça vous dit ... ?).

Vous pourrez toutefois manipuler son contenu. Retenez juste ce qu'il y a dedans, cela suffira. Nous verrons son utilisation combinée avec les boucles dans la partie « Boucles ».

# EXAMPLE !

```
print mot
print nombres

print reversed(mot)
print reversed(nombres)

print type(reversed(mot))
print type(reversed(nombres))
```

```
abcdef
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
<reversed object at 0x7fca8a833c10>
<listreverseiterator object at 0x7fca8a833c10>
<type 'reversed'>
<type 'listreverseiterator'>
```

# OUVRIR UN FICHIER

**open( ) et close( ) (fopen( ) et fclose( ) en C)**

Pour lire un fichier, vous devez l'ouvrir, puis le fermer. C'est ce que ces deux fonctions font respectivement. Généralement, on lit un fichier texte (.txt), ou d'un autre type, avec du texte.

Une fois que vous avez ouvert votre fichier, vous pouvez faire trois choses : lire le fichier, écrire dans le fichier, ou ajouter quelque chose dans le fichier.

Pour préciser ce que vous voulez faire, il faut l'ajouter en paramètre dans les parenthèses de la fonction open, sous forme d'une chaîne de caractères, en deuxième, après le nom du fichier à lire.

Syntaxe :

```
open(nom_fichier, mode)  
nom_fichier.close()
```



N'oubliez pas de fermer votre fichier après avoir fini de l'utiliser.  
Dans le cas contraire, il se pourrait que votre fichier soit endommagé.

Lorsque que vous ouvrez un pot de Nutella, et que vous avez fini, vous le refermez. Que se passerait-il si vous ne le fermez pas ? Son contenu risque d'être modifié !

C'est pareil ici !

# LIRE, ÉCRIRE, AJOUTER

**'r'** (**read**) : lit uniquement le fichier. Ne permet pas d'autres opérations.

**'w'** (**write**) : efface ce qu'il y a dans le fichier (s'il y a quelque chose), et écrit dedans.

**'a'** (**append**) : ajoute quelque chose à la suite de ce qu'il y a dans le fichier (s'il y a quelque chose). Sinon ne fait qu'écrire dedans.

# LIRE UN FICHIER

Pour lire un fichier, vous devez l'ouvrir et spécifier le mode dans lequel vous souhaitez traiter ce fichier. Ici, la lecture ('r').

```
nom_fichier = 'texte.txt'  
fichier = open(nom_fichier, 'r')
```

Ou bien

```
fichier = open('texte.txt', 'r')
```

Vous avez sans doute remarqué que j'ai affecté ma fonction `open( )` dans une variable. En effet, lors de l'ouverture du fichier, vous précisez à l'ordinateur que vous voulez lire dedans, mais vous ne l'avez pas encore fait. Le fichier est chargé et est prêt à être lu.

Pour le lire, nous allons utiliser la variable dans laquelle on a affecté le fichier et les fonctions `readline( )` et `readlines( )`.

# READLINE( )

La fonction **readline( )** ne lit « **qu'une** » seule ligne du fichier. Il s'arrête dès qu'il a lu toute la ligne.

Il stocke le résultat sous forme d'une chaîne de caractères.

1 ligne = 1 chaîne de caractères (signe « égal », pas affectation !)

# EXEMPLE

texte.txt

```
1 premiere ligne  
2 deuxieme ligne  
3 87667576575576  
4 quatrieme ligne
```

script.py

```
fichier = open('texte.txt', 'r')  
  
ligne = fichier.readline()  
  
print type(ligne)  
print ligne  
  
fichier.close()
```

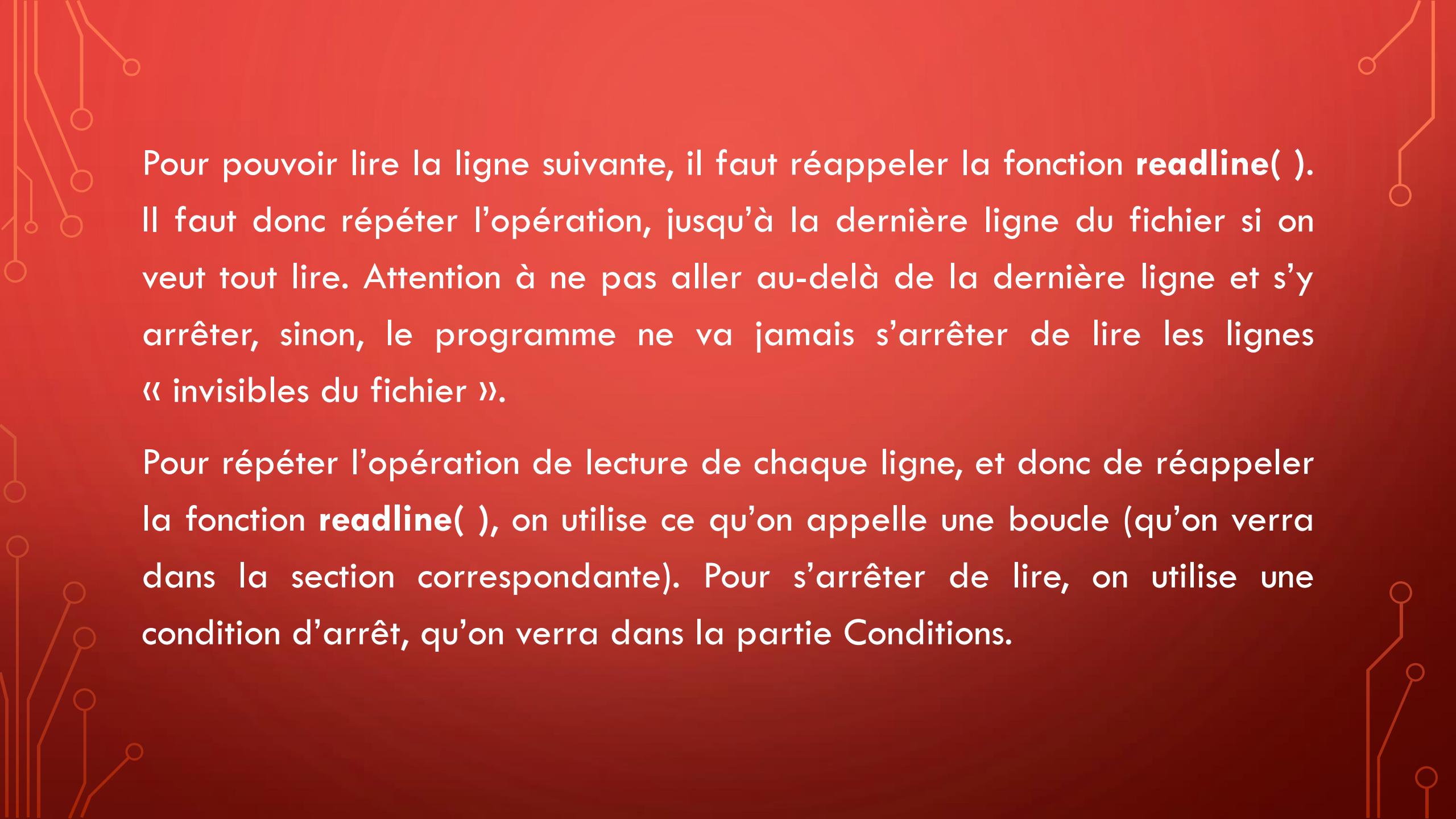
Terminal

```
romain@Hp-debian:~/Documents$ python script.py  
<type 'str'>  
premiere ligne  
  
romain@Hp-debian:~/Documents$
```

Vous avez sans doute remarqué que dans le terminal, il y a une ligne vide après « première ligne ». En effet à la fin de chaque ligne, il y a un retour à la ligne, spécifié par ‘\n’.



```
romain@Hp-debian:~/Documents$ python script.py
<type 'str'>
premiere ligne
romain@Hp-debian:~/Documents$
```



Pour pouvoir lire la ligne suivante, il faut réappeler la fonction **readline()**. Il faut donc répéter l'opération, jusqu'à la dernière ligne du fichier si on veut tout lire. Attention à ne pas aller au-delà de la dernière ligne et s'y arrêter, sinon, le programme ne va jamais s'arrêter de lire les lignes « invisibles du fichier ».

Pour répéter l'opération de lecture de chaque ligne, et donc de réappeler la fonction **readline()**, on utilise ce qu'on appelle une boucle (qu'on verra dans la section correspondante). Pour s'arrêter de lire, on utilise une condition d'arrêt, qu'on verra dans la partie Conditions.

# ENCORE UN EXEMPLE !

```
fichier = open('texte.txt', 'r')

ligne = fichier.readline()
print ligne

fichier.close()
```

```
romain@Hp-debian:~/Documents$ python script.py
premiere ligne

deuxieme ligne

87667576575576

quatrieme ligne

romain@Hp-debian:~/Documents$ |
```

# READLINES( )

Cette fonction lit toutes les lignes du fichier d'un coup. Pas besoin de la réappeler à chaque fois.

Cette fonction est à la fois pratique et désavantageuse par rapport à la fonction **readline( )** qui ne lit qu'une ligne à la fois.

Pratique parce qu'il ne faut pas la réappeler, il stocke chaque ligne dans une liste, les unes à la suite des autres.

Désavantageuse car dans le cas où l'on ne veut que l'information contenue dans une seule ligne, stocker tout l'ensemble du fichier puis chercher ce que l'on veut coûte de la mémoire.

# EXEMPLE !

```
fichier = open('texte.txt', 'r')  
lignes = fichier.readlines()  
print lignes  
fichier.close()
```

```
['premiere ligne\n', 'deuxieme ligne\n', '87667576575576\n', 'quatrieme ligne\n', 'cinquieme ligne']
```

Notez la présence des '\n' à la fin de chaque ligne (sauf la dernière).

# ÉCRIRE DANS UN FICHIER

`write( )`

Assurez-vous d'avoir ouvert votre fichier en mode écriture ('w').

Mettez dans les parenthèses ce que vous souhaitez écrire dans votre fichier.

N'oubliez pas les apostrophes ou guillemets dans le cas des chaînes de caractères.

# EXEMPLE

script.py

```
fichier = open('texte2.txt', 'w')  
fichier.write("Ceci est une chaine de caracteres")  
fichier.close()
```

texte2.txt

```
1 Ceci est une chaine de caracteres
```

# AJOUTER DANS UN FICHIER

Il suffit de spécifier le mode « ajouter » ('a') lors de l'ouverture du fichier, et de réutiliser la fonction **write( )**.

# EXEMPLE

script.py

```
fichier = open('texte.txt', 'a')  
fichier.write("cinquieme ligne")  
fichier.close()
```

texte.txt

```
1 premiere ligne  
2 deuxieme ligne  
3 8766757655576  
4 quatrieme ligne  
5 cinquieme ligne
```

# MAXIMUM ET MINIMUM

**max( ) et min( )**

Ces deux fonctions permettent de trouver la valeur maximale ou minimale entre deux éléments ou plus. Comme les autres fonctions, on peut garder la valeur en l'affectant à une variable, ou l'afficher simplement.

# EXEMPLE

```
nombre1 = 3
nombre2 = 76
nombre3 = 40
nombre4 = max(nombre2, nombre3)
nombre5 = min(nombre2, nombre3)

print nombre4
print max(nombre1, nombre3)
print max(nombre1, nombre2, nombre3)

print nombre5
print min(nombre1, nombre2, nombre3)
print min(nombre2, nombre3)
```

```
76
40
76
40
3
40
```

# TRIER

**sorted( )**

Généralement utilisée pour trier les éléments d'une liste (ensemble d'éléments).

Dans le cas où il n'y a que des entiers ou valeurs décimales, trie selon l'ordre croissant.

Dans le cas où il n'y a que des chaînes de caractères, trie selon l'ordre alphabétique.

Pour trier dans l'ordre décroissant, ou alphabétique inverse ( $Z > A$ ), ajouter :

**reverse = True**

# EXEMPLES

```
nombres = range(10, 0, -1)
lettres = 'dkfhskfhksjh'

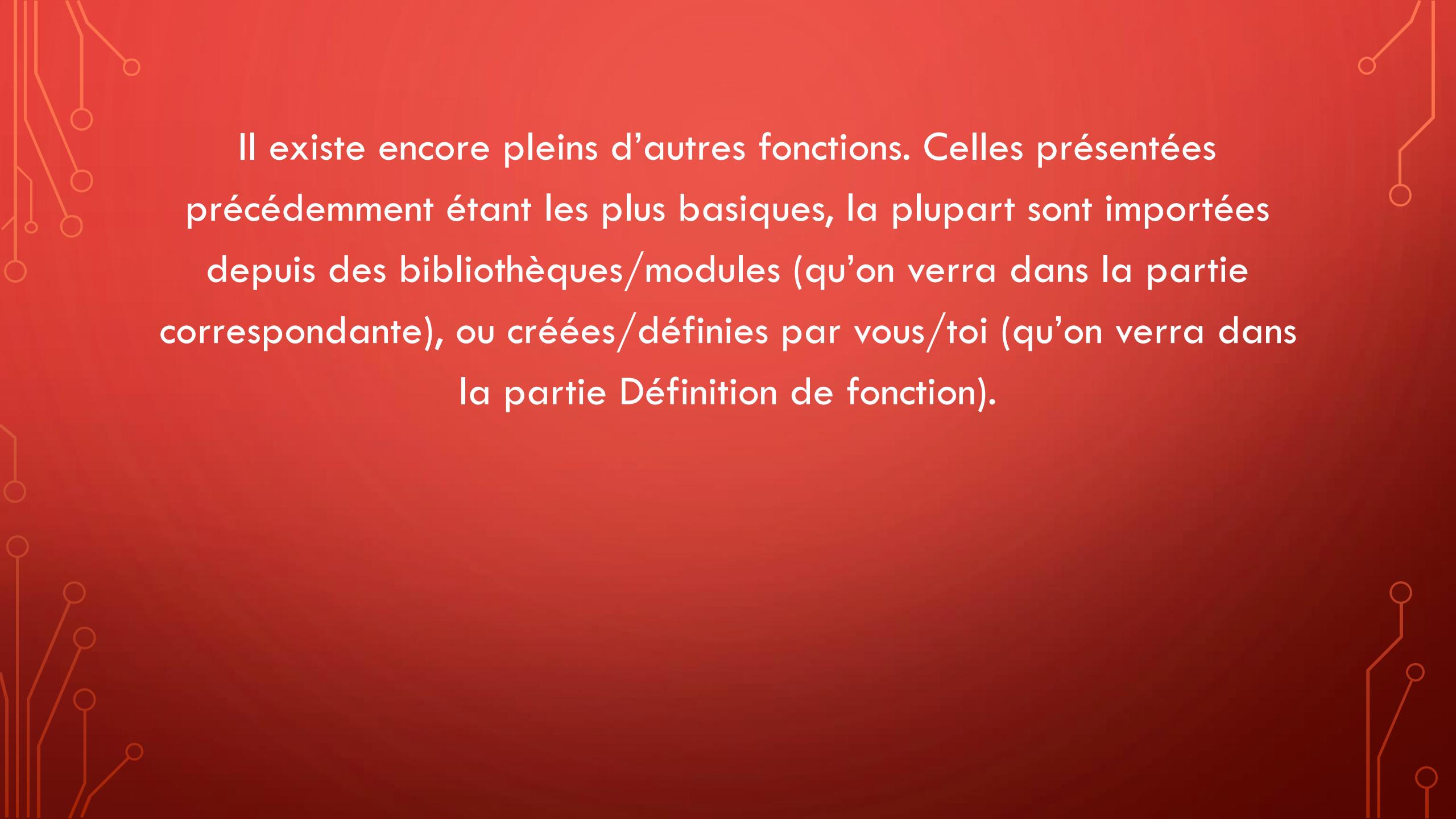
print nombres
print sorted(nombres)
print sorted(lettres)

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
['d', 'f', 'f', 'h', 'h', 'h', 'j', 'k', 'k', 'k', 's', 's']
```

```
nombres = range(10)
lettres = 'kjqshlskvfdkljm'

print nombres
print sorted(nombres, reverse=True)
print sorted(lettres, reverse=True)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
['v', 's', 's', 'q', 'm', 'l', 'l', 'k', 'k', 'k', 'j', 'j', 'h', 'f', 'd']
```



Il existe encore pleins d'autres fonctions. Celles présentées précédemment étant les plus basiques, la plupart sont importées depuis des bibliothèques/modules (qu'on verra dans la partie correspondante), ou créées/définies par vous/toi (qu'on verra dans la partie Définition de fonction).

# EXERCICES !!!!!!!

1) Ecrire un programme qui :

- Demande à l'utilisateur une chaîne de caractères
- Affiche son type
- Affiche sa longueur (en tenant compte des espaces)
- Et écrit la chaîne de caractère 5 fois dans un fichier .txt

2) Soit les deux variables suivantes :

```
nombre1 = 135649872545132168756465761231684.54654216548
```

```
nombre2 = 135649872545132168756495761231684.54654316548
```

Ecrire un programme qui donne le plus grand des deux, et calcule leur différence.

# LISTES ET TUPLES

Les listes et les tuples !

On commence d'abord par les listes si vous le voulez bien !

Pas d'objections possibles !

Dommage !

# LES LISTES

Kézako une liste ?

C'est un ensemble de variables (avec quelque chose dedans bien sûr).

Une liste peut contenir des variables ayant chacun un type différent (int, float, list (une liste dans une liste oui), str, etc.), contrairement au C où vous ne pouviez mettre qu'un seul type.

On peut reconnaître une liste de par la présence des « [ ] » (les crochets, pas les guillemets).

# CRÉER UNE LISTE

Affectez votre liste à une variable, c'est mieux (et conseillé pour la manipuler ensuite). Sinon votre liste sera perdue quelque part dans la mémoire de votre ordinateur et vous ne pourrez plus y accéder (pauvre liste).

Contrairement au C, vous n'avez pas besoin de spécifier sa taille, ni les types des variables que vous allez mettre dedans.

# LA LISTE VIDE

```
liste = []
```

Ou bien

```
liste = list()
```

Une liste qui s'appelle « liste », tout simplement, et vide. Il n'y a rien dedans. Vous voulez y mettre quelque chose dedans ? Bien sûr ! Sinon elle ne sert strictement à rien !

Mais attendez, pas tout de suite. Voyons d'abord d'autres façons de créer une liste sans y ajouter d'autres trucs après (vous pouvez y ajouter des trucs après sa création, bien entendu, ce n'est pas interdit).

# EXEMPLES !

```
liste1 = [0, 45, 16, 78, 100, 65487654]
```

```
liste2 = [1.46, 1.987, 0.465700365487987]
```

```
liste3 = ['sdlfsdjf', 'serskr', 'G']
```

```
liste4 = [1321312, 1.4654, 'sdfsdf']
```

```
liste5 = [liste1, liste3]
```

Essayez de les taper dans votre script, affichez les listes (print) et lancez votre programme !

```
listel = [0, 45, 16, 78, 100, 65557687]
liste2 = [1.46, 1.987, 0.89787667]
liste3 = ['sldfjslfkj', 'HGJK', 'P']
liste4 = [197979, 1.0980, 'lkjUYT']
liste5 = [listel, liste3]

print listel
print liste2
print liste3
print liste4
print liste5
```

```
[0, 45, 16, 78, 100, 65557687]
[1.46, 1.987, 0.89787667]
['sldfjslfkj', 'HGJK', 'P']
[197979, 1.098, 'lkjUYT']
[[0, 45, 16, 78, 100, 65557687], ['sldfjslfkj', 'HGJK', 'P']]
```

Chaque élément d'une liste est séparée par une « , » (la virgule).

La liste « liste1 » ne contient que des entiers (int). Rien de surprenant.

La liste « liste2 » ne contient que des valeurs décimales (float). Remarquez qu'il faut utiliser un « . » à la place d'une « , » pour séparer les décimales du reste. Sinon cela va compter pour 2 éléments. [1,2] différent de [1.2].

La liste « liste3 » ne contient que des chaînes de caractères, et un caractère (le G), mais est aussi compté comme une chaîne de caractères. Les chaînes de caractères sont toujours entourées d'apostrophes « ' », ou de guillemets « « » ».

La liste « liste4 » contient un entier (int), une valeur décimale (float), et une chaîne de caractères (str). Je vous avez dit que vous pouviez mettre plusieurs types dans une liste !

La liste « liste5 » contient deux listes (liste1 et liste3). Essayez de deviner ce qui va s'afficher dans le terminal avant de lancer votre programme (sauf si vous l'aviez déjà fait).

# ACCÉDER À UN ÉLÉMENT D'UNE LISTE

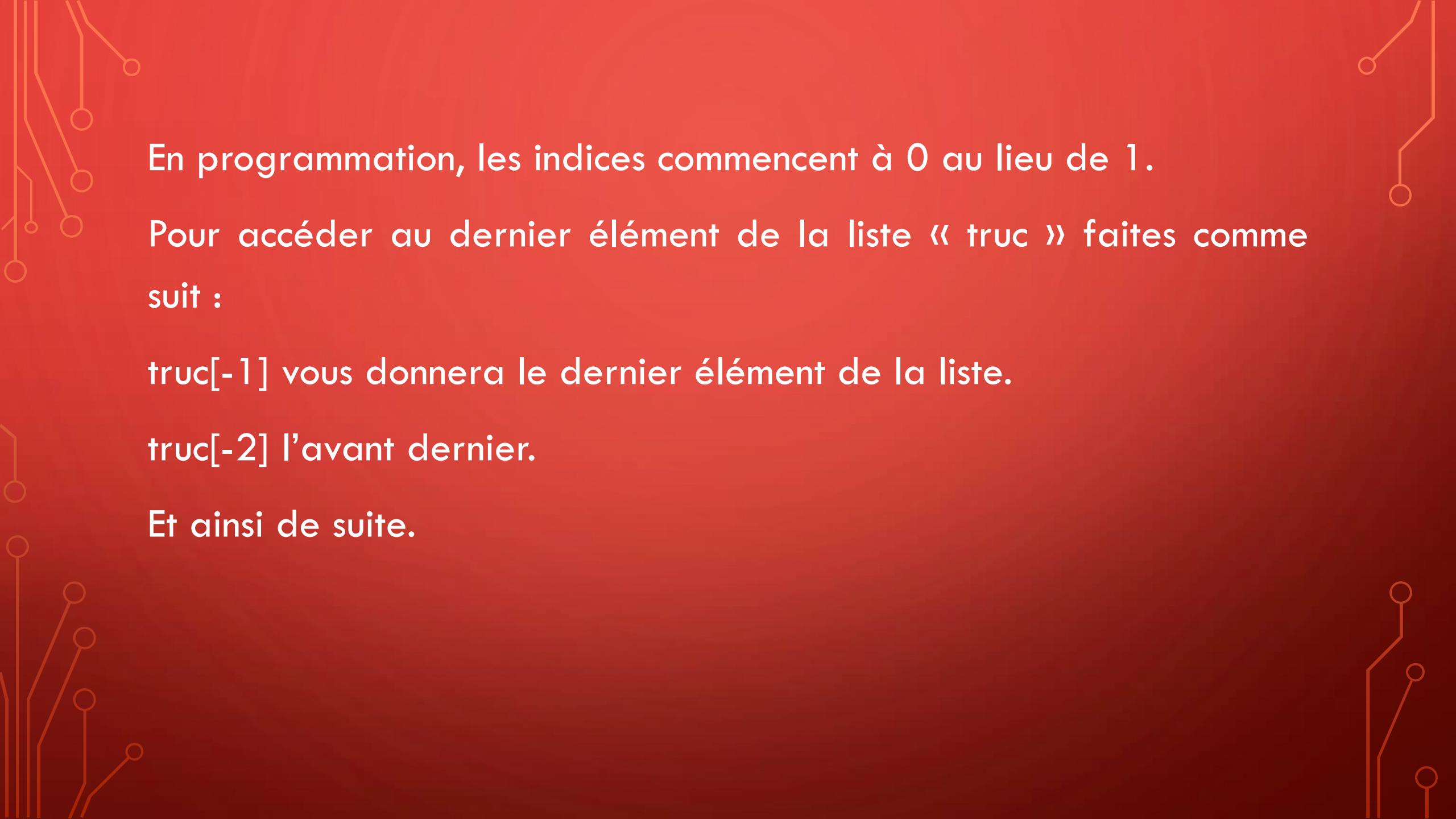
Si vous souhaitez accéder à un élément d'une liste, il faut utiliser des crochets à la suite du nom de votre liste (qui existe !).

Si votre liste s'appelle « truc » :

`truc[0]` vous donne le premier élément de la liste.

`truc[1]` le deuxième.

Et ainsi de suite.



En programmation, les indices commencent à 0 au lieu de 1.

Pour accéder au dernier élément de la liste « truc » faites comme suit :

`truc[-1]` vous donnera le dernier élément de la liste.

`truc[-2]` l'avant dernier.

Et ainsi de suite.

# EXEMPLE

```
truc = [1, 'lkfjxlkfjklf', 'POPO', 1.98]  
  
print truc[0]  
print truc[1]  
print truc[3]  
print truc[-1]
```

```
1  
lkfjxlkfjklf  
1.98  
1.98
```

# ACCÉDER AUX ÉLÉMENTS D'UNE LISTE

Pour extraire un certain nombre d'éléments de votre liste, regroupés dans un intervalle bien précis, faites comme suit :

**nom\_de\_votre\_liste[debut:fin]**

debut et fin représentent les indices de début et fin de l'intervalle de la liste des éléments que vous souhaitez avoir (fin exclue).

# EXEMPLES

```
nombres = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
print nombres  
print nombres[0:5]  
print nombres[3:6]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
[0, 1, 2, 3, 4]  
[3, 4, 5]
```

# AUTRES MANIPULATIONS POSSIBLES

Soit la liste « liste ».

`liste[:]` vous donne toute la liste

`liste[:3]` vous donne les 3 premiers éléments de la liste.

`liste[3:]` vous donne toute la liste sauf les 3 premiers éléments.

# ILLUSTRATION

```
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print liste
print liste[:]
print liste[:3]
print liste[3:]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2]
[3, 4, 5, 6, 7, 8, 9]
```

# AJOUTER UN ÉLÉMENT DANS UNE LISTE

Fonction **append( )**

Mettre dans les parenthèses ce que vous voulez ajouter dans votre liste.

Attention ! Lorsqu'on veut ajouter un élément dans une liste, on ajoute toujours à la fin de la liste. Pour l'insérer à un endroit donné, il faut faire d'autres manipulations.

# EXAMPLE !

Si votre liste s'appelle « TrucMachinChose », et que vous voulez ajouter la chaîne de caractères « Chouette » dans votre liste, faites comme suit :

**TrucMachinChose.append(« Chouette »)**

Vous devez préciser la liste dans laquelle vous souhaitez ajouter un élément. Vous ne pouvez pas ajouter un élément dans rien du tout.

J'ai dans ce cas ajouté une chaîne de caractères dans la liste. Notez la présence des guillemets. J'aurais pu utiliser des apostrophes, mais cela revient au même.

Note : N'oubliez pas de créer votre liste avant !

## AUTRES EXEMPLES

```
UneListeVide = []
UneListePasVide = ['Ceci', 'est', 'une']

print UneListeVide
print UneListePasVide

UneListeVide.append('Bonjour')
UneListePasVide.append('liste')

print UneListeVide
print UneListePasVide
```

```
[]
['Ceci', 'est', 'une']
['Bonjour']
['Ceci', 'est', 'une', 'liste']
```

# AJOUT D'UN ÉLÉMENT À UNE LISTE À UNE POSITION PRÉCISE

## Fonction `insert( )`

La fonction prend deux arguments :

- La position où l'élément doit être inséré
- L'élément à insérer

Syntaxe :

```
liste.insert(position, élément)
```

# EXEMPLE

```
nombres = [0, 1, 2, 4, 5]
phrase = ['Je', 'une', 'phrase']

print nombres
print phrase

nombres.insert(3, 3)
phrase.insert(1, 'contient')

print nombres
print phrase
```

```
[0, 1, 2, 4, 5]
['Je', 'une', 'phrase']
[0, 1, 2, 3, 4, 5]
['Je', 'contient', 'une', 'phrase']
```

# SUPPRIMER UN ÉLÉMENT D'UNE LISTE

## Fonction **remove()**

Spécifier entre les parenthèses l'élément (et non l'indice de l'élément) que vous voulez supprimer.

Lorsque vous supprimez un élément d'une liste, tous les autres éléments derrière votre élément vont se décaler vers la gauche. Ils vont donc changer d'indice.

Si par exemple l'élément 'RTR' est à l'indice 5, et que 'TTT' est à l'indice 6, et que vous supprimez 'RTR', 'TTT' va prendre l'indice 5.

Syntaxe :

```
une_liste.remove(quelque_chose)
```

# EXEMPLE

```
nombres = [0, 1, 2, 47, 3, 4]
phrase = ['Je', 'ne', 'possede', 'pas', 'une', 'phrase']
exemple = ['coucou', 'boujour', 'salut', 'wesh']

print nombres
print phrase
print exemple

nombres.remove(47)
phrase.remove('ne')
phrase.remove('pas')
exemple.remove(exemple[2])

print nombres
print phrase
print exemple
```

```
[0, 1, 2, 47, 3, 4]
['Je', 'ne', 'possede', 'pas', 'une', 'phrase']
['coucou', 'boujour', 'salut', 'wesh']
[0, 1, 2, 3, 4]
['Je', 'possede', 'une', 'phrase']
['coucou', 'boujour', 'wesh']
```

# SUPPRESSION PARTICULIÈRE

## Fonction **pop( )**

La fonction **pop( )** supprime un élément de votre liste et la « sauvegarde » en mémoire. À affecter à une variable pour la récupérer. Dans les parenthèses, précisez l'indice de l'élément à récupérer. Si rien de spécifié, le dernier élément de la liste est récupéré.

Syntaxe :

```
une_liste.pop()  
une_liste.pop(indice)
```

# EXEMPLES

```
UnePhrase = ['Je', 'suis', 'en', 'train', 'de', 'coder', 'dormir']

print UnePhrase

mot = UnePhrase.pop()

print UnePhrase
print mot

['Je', 'suis', 'en', 'train', 'de', 'coder', 'dormir']
['Je', 'suis', 'en', 'train', 'de', 'coder']
dormir
```

```
liste = [1, 13, 16, 10, 20]

p = liste.pop(2)
print liste
print p

p = liste.pop(-1) #equivaut a liste.pop()
print liste
print p

[1, 13, 10, 20]
16
[1, 13, 10]
20
```

# AFFICHER L'INDICE D'UN ÉLÉMENT

Il est possible d'afficher l'indice d'un élément dans une liste grâce à la fonction **index( )**. Dans le cas où l'élément recherché n'est pas trouvé, une erreur est affichée dans le terminal.

Syntaxe :

```
une_liste.index(quelque_chose)
```

Mais voyons un exemple tout de suite !

# EXEMPLE !

```
nombres = [0, 1, 2, 3, 4, 5, 6, 7, 8]  
  
print nombres.index(4)  
print nombres.index(nombres[4])  
print nombres.index(9)
```

```
4  
4  
Traceback (most recent call last):  
  File "script.py", line 9, in <module>  
    print nombres.index(9)  
ValueError: 9 is not in list
```

La valeur 9 n'étant pas trouvée dans la liste, une erreur est affichée dans le terminal. On verra plus en détail les erreurs dans la dernière partie de cette présentation.

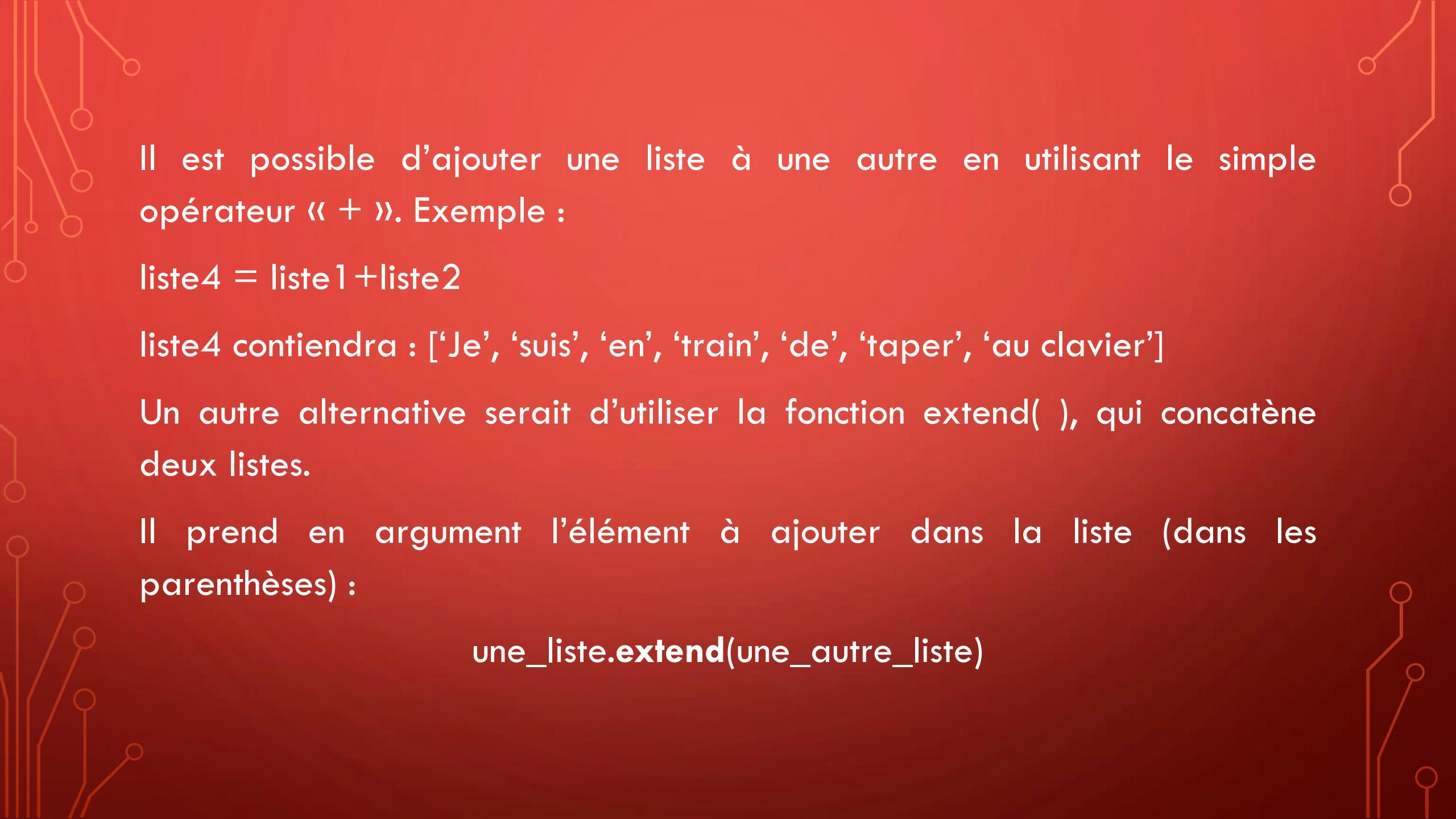
# OPÉRATIONS SUR LES LISTES

Soit les listes suivantes :

```
liste1 = ['Je', 'suis', 'en', 'train']
```

```
liste2 = ['de', 'taper', 'au clavier']
```

```
Liste3 = [36, 12, 6]
```



Il est possible d'ajouter une liste à une autre en utilisant le simple opérateur « + ». Exemple :

```
liste4 = liste1+liste2
```

liste4 contiendra : ['Je', 'suis', 'en', 'train', 'de', 'taper', 'au clavier']

Un autre alternative serait d'utiliser la fonction `extend( )`, qui concatène deux listes.

Il prend en argument l'élément à ajouter dans la liste (dans les parenthèses) :

```
une_liste.extend(une_autre_liste)
```

# ILLUSTRATION

```
listel = ['Je', 'suis', 'en', 'train']
liste2 = ['de', 'taper', 'au clavier']
liste3 = [36, 12, 6]

print listel
print liste2

liste4 = listel+liste2

print liste4
```

Ou bien

```
listel.extend(liste2)

print listel
```

```
['Je', 'suis', 'en', 'train']
['de', 'taper', 'au clavier']
['Je', 'suis', 'en', 'train', 'de', 'taper', 'au clavier']
```

On peut également dédoubler une liste en la multipliant par 2, ou la tripler, etc.

```
listel = ['Je', 'suis', 'en', 'train']
liste2 = ['de', 'taper', 'au clavier']
liste3 = [36, 12, 6]

liste5 = liste3*2

print liste5
```

```
[36, 12, 6, 36, 12, 6]
```

# DÉTERMINER SI UN ÉLÉMENT EST DANS LA LISTE OU NON

Soit la liste suivante :

```
nombres = [1, 2, 3, 4, 5, 6]
```

On souhaite vérifier si le nombre 0 est dans la liste. Il suffit de faire comme suit :

```
0 in nombres
```

Il s'affichera **True** ou **False** selon si l'élément recherché est dans la liste ou non. Dans notre cas **False** sera affiché.

Note : dans le cas où l'élément est trouvé, l'indice de l'élément dans la liste ne sera pas affiché. **True** ou **False** en résultera. Vous pouvez utiliser la fonction **index( )** pour trouver l'indice de l'élément trouvé.

# TEST !

```
nombres = [1, 2, 3, 4, 5, 6]  
Vrai = 1 in nombres  
  
print 0 in nombres  
print Vrai
```

```
False  
True
```

# JOINDRE CHAQUE ÉLÉMENT DE LA LISTE EN UNE CHAINE DE CARACTÈRES

Reprendons notre liste4 :

```
liste4 = ['Je', 'suis', 'en', 'train', 'de', 'taper', 'au clavier']
```

Il est possible de tout concaténer sous forme d'une chaîne de caractères avec la fonction **join( )**. Cette fonction prend en argument la liste dont les éléments sont à concaténer. Le « séparateur » qui sera inséré entre chaque élément de la liste sera spécifié devant la fonction.

```
séparateur.join(une_liste)
```

# ILLUSTRATION

```
liste4 = ['Je', 'suis', 'en', 'train', 'de', 'taper', 'au', 'clavier']

phrase = " ".join(liste4)
phrase1 = "@".join(liste4)
phrase2 = "_".join(liste4)

print liste4
print phrase
print phrase1
print phrase2
```

```
['Je', 'suis', 'en', 'train', 'de', 'taper', 'au', 'clavier']
Je suis en train de taper au clavier
Je@suis@en@train@de@taper@au@clavier
Je_suis_en_train_de_taper_au_clavier
```

# SÉPARER UNE CHAINE DE CARACTÈRES SELON UN SÉPARATEUR

On a vu comment concaténer les éléments d'une liste en une chaîne de caractères. On peut également faire l'inverse : séparer les caractères d'une chaîne de caractères selon un séparateur avec la fonction `split()`. Elle prend en argument le séparateur.

Génère une liste des éléments de la chaîne de caractères séparés par le séparateur spécifié.

```
une_liste.split(séparateur)
```

# EXEMPLE

```
UnePhrase = 'Ceci-est-une-phrase'  
phrase = UnePhrase.split("-")  
print UnePhrase  
print phrase
```

```
Ceci-est-une-phrase  
['Ceci', 'est', 'une', 'phrase']
```

# LISTES 2D, 3D, 4D, ...

Listes 2D : une liste dans une liste

Listes 3D : une liste dans une liste dans une liste

Listes 4D : une liste dans une liste dans une liste dans une liste

Etc.

Pour accéder à chaque élément de votre sous-liste de votre liste, il suffit de doubler, tripler, quadrupler ... votre paire de crochets.

# EXEMPLE

```
liste2D = [[1, 2],[3, 4]]  
liste3D = [[[1, 2, 3],[4, 5, 6]],[[7, 8, 9],[10, 11, 12]]]  
liste4D = [[[[1, 2],[3, 4]],[[5, 6],[7, 8]]],[[[9, 10],[11, 12]],[[13, 14],[15, 16]]]]  
  
print liste2D  
print liste2D[0]  
print liste2D[0][1]  
  
print liste3D  
print liste3D[0]  
print liste3D[0][1]  
print liste3D[0][1][0]  
  
print liste4D  
print liste4D[0]  
print liste4D[0][1]  
print liste4D[0][1][0][1]
```

```
[[1, 2], [3, 4]]  
[1, 2]  
2  
[[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]]  
[[1, 2, 3], [4, 5, 6]]  
[4, 5, 6]  
4  
[[[[1, 2], [3, 4]], [[5, 6], [7, 8]]], [[[9, 10], [11, 12]], [[13, 14], [15, 16]]]]  
[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]  
[[5, 6], [7, 8]]  
6
```

# LES TUPLES

Ce sont des listes non modifiables. Vous ne pouvez ni en supprimer les éléments, ni en ajouter. Mais vous pouvez accéder aux éléments du tuple de la même façon que vous pouvez accéder aux éléments d'une liste.

Il faut donc faire attention au moment de la création du tuple. Souvenez vous que vous ne pourrez plus la modifier.

On peut reconnaître un tuple par la présence de parenthèses autour d'éléments (sans qu'il y ai un nom devant (à la différence des fonctions)).

Vous pouvez également avoir des tuples de plusieurs dimensions.

Tuple vide :

```
UnTuple = ()
```

Ou bien

```
UnTuple = tuple()
```

Si vous avez suivi ce qui est écrit dans la diapositive précédente, créer un tuple vide ne sert à rien. En effet vous ne pourrez pas le modifier par la suite.

# EXEMPLE

```
UnTuple = (12, 8.98, 'slfdkj', 'KLHJ87')
UnTuple2 = ((1, 2), (3, 4))

print UnTuple
print UnTuple[1]

print UnTuple2
print UnTuple2[0]
print UnTuple2[0][1]

UnTuple.remove(12)
```

```
(12, 8.98, 'slfdkj', 'KLHJ87')
8.98
((1, 2), (3, 4))
(1, 2)
2
Traceback (most recent call last):
  File "script.py", line 17, in <module>
    UnTuple.remove(12)
AttributeError: 'tuple' object has no attribute 'remove'
```

On remarque également ici une erreur affichée dans le terminal que l'on verra plus en détail dans la partie correspondante.

# EXERCICES !!!!!!

- 1) En utilisant les fonctions `range( )`, et `remove( )`, obtenir la liste suivante : [0, 1, 3, 5, 7, 8, 9 11, 12, 13, 14, 16]
- 2) Soit la liste suivante : [5, 6, 4, 98, -3, 51, -46, -7, 52]  
Triez-la dans l'ordre décroissant
- 3) Affectez cette chaîne de caractères dans une variable puis mettre chaque mot de cette chaîne dans une liste
- 4) Joindre les mots de la liste précédente avec une \*
- 5) Affichez la liste créée dans 3), puis la « vider », puis la réafficher à nouveau (la liste)
- 6) Obtenez la liste suivante : [['bonjour', 'salut', 'wesh'], [1 ,2 ,3]] en utilisant des fonctions propres aux listes (ne trichez pas !)
- 7) Affichez « wesh » et « 2 » de la liste dans 6)

C'était long !

Passons maintenant aux ...

# Dictionnaires

# QU'EST-CE DONC ?

Un recueil de mots avec leurs définitions ... Mauvaise réponse ! (Du moins ici).

C'est comme une liste, tout en n'étant pas une liste. C'est également un ensemble d'éléments regroupés en un seul élément. Ces éléments sont divisés en deux sous-éléments :

- une **clé** qui peut être une chaîne de caractères, un(e) liste/tuple, un nombre.
- une **valeur** qui est une variable. Ce que je sous-entends par variable, c'est que cela peut être un entier, une liste, un dictionnaire aussi, ou autre.

On peut reconnaître un dictionnaire par la présence d'accolades « { } ».

De la même façon qu'une liste, on peut créer un dictionnaire vide de cette façon :

```
Dico = {}
```

Ou bien

```
dico = dict()
```

On pourra par la suite ajouter et supprimer des éléments du dictionnaire (comme pour une liste). On appelle l'ensemble clé/valeur, un objet (ou item en anglais).

## MAIS VOYONS UN EXEMPLE TOUT DE SUITE

Voici un exemple rapide pour comprendre ce qu'est un dictionnaire.

Tout d'abord créons notre dictionnaire (comme on crée une liste) :

appelons notre dictionnaire Age. Le dictionnaire Age contiendra en clé les noms de plusieurs personnes, et en valeur, leur âge.

Ça y est ? Vous voyez à quoi correspond un dictionnaire à peu près et à quoi ça sert ?

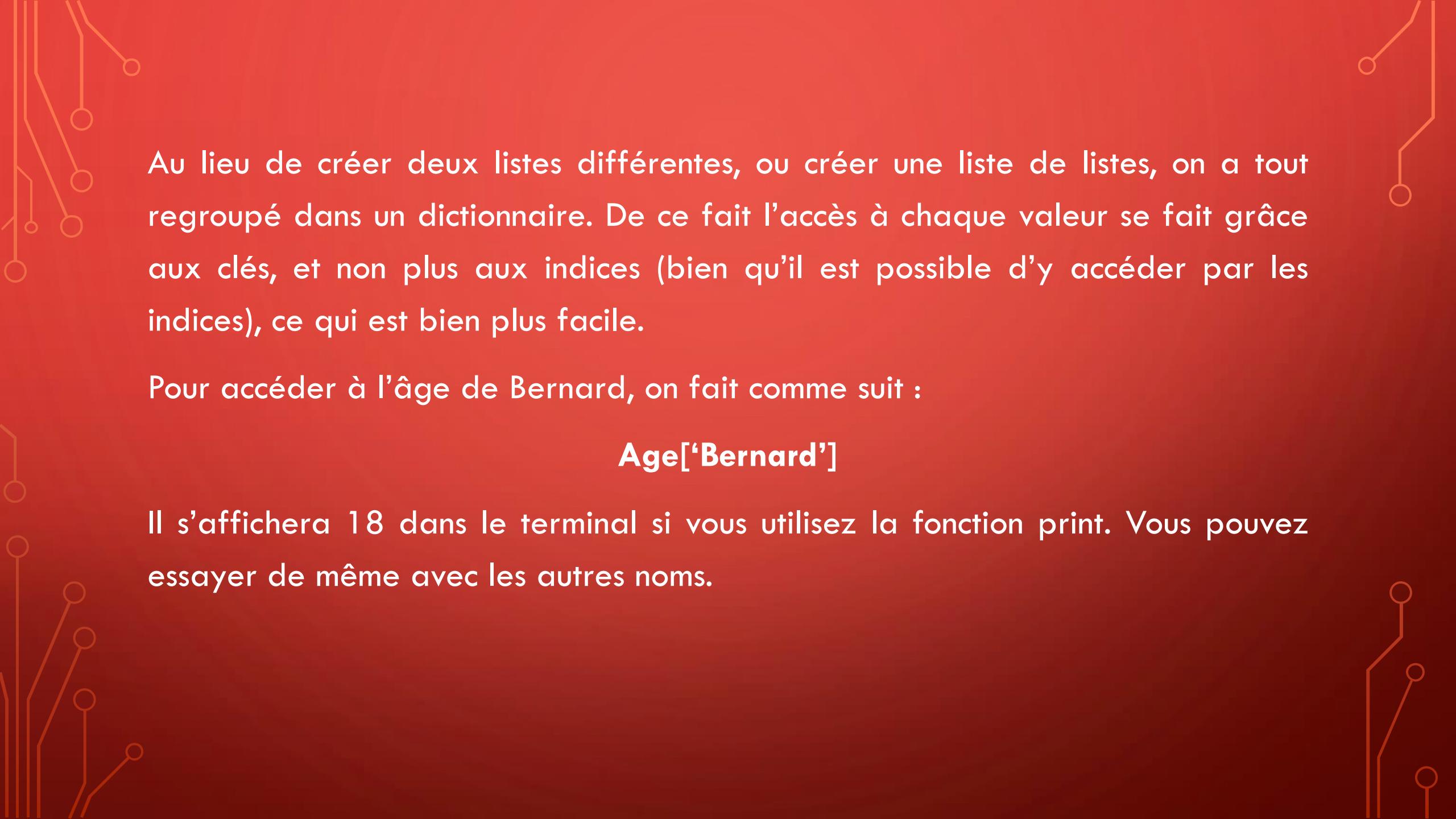
## ILLUSTRATION

Age = {'Bernard' : 18, 'Michel' : 20, 'Lucie' : 23, 'Martine' : 21}

Remarquez que chaque couple clé/valeur est également séparé par une « , ».

Ici les clés sont : Bernard, Michel, Lucie, Martine. Notez que les clés sont entourés d'apostrophes (toujours).

A chaque clé (et donc à chaque personne), on associe une valeur (un âge dans ce cas) qui sont respectivement : 18, 20, 23, 21.



Au lieu de créer deux listes différentes, ou créer une liste de listes, on a tout regroupé dans un dictionnaire. De ce fait l'accès à chaque valeur se fait grâce aux clés, et non plus aux indices (bien qu'il est possible d'y accéder par les indices), ce qui est bien plus facile.

Pour accéder à l'âge de Bernard, on fait comme suit :

**Age['Bernard']**

Il s'affichera 18 dans le terminal si vous utilisez la fonction print. Vous pouvez essayer de même avec les autres noms.

# EXEMPLE

```
Age = {'Bernard' : 18, 'Michel' : 20, 'Lucie' : 23, 'Martine' : 21}  
  
print Age['Bernard']  
print Age['Michel']  
print Age['Lucie']  
print Age['Martine']
```

```
18  
20  
23  
21
```

## CRÉER UNE CLÉ

On peut créer/ajouter une clé dans un dictionnaire vide, ou avec des éléments déjà à l'intérieur. C'est très simple.

Si notre dictionnaire est le suivant :

`capitale = {'France' : 'Paris', 'Angleterre' : 'Londres', 'Espagne' : 'Madrid'}`

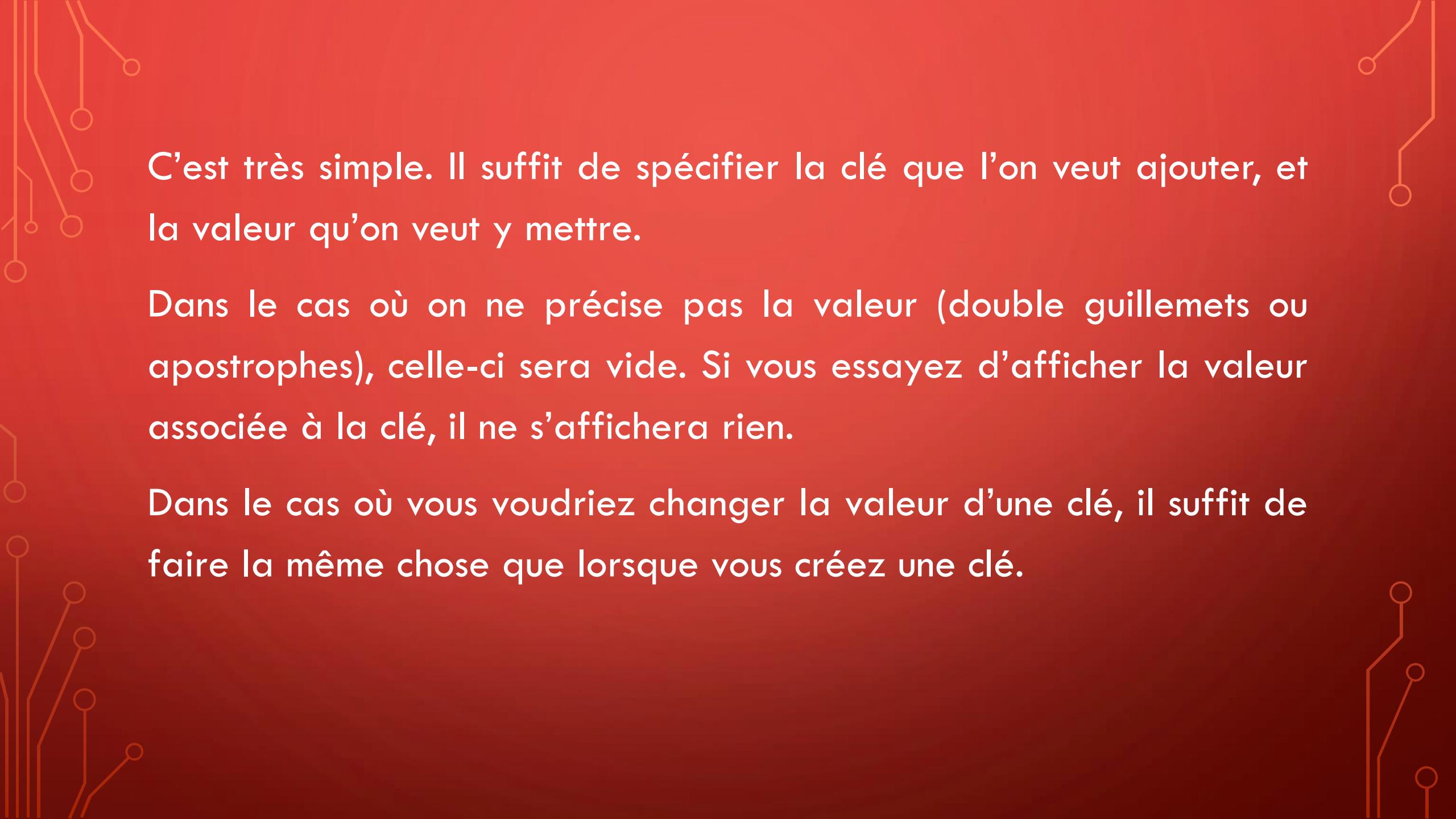
et que l'on souhaite ajouter la clé Italie et la valeur associée Rome, on procède comme suit :

**`capitale['Italie'] = 'Rome'`**

# ILLUSTRATION

```
capitale = {'France' : 'Paris', 'Angleterre' : 'Londres', 'Espagne' : 'Madrid'}  
  
print capitale  
  
capitale['Italie'] = 'Rome'  
  
print capitale  
print capitale['Italie']
```

```
{'Espagne': 'Madrid', 'Angleterre': 'Londres', 'France': 'Paris'}  
{'Espagne': 'Madrid', 'Angleterre': 'Londres', 'Italie': 'Rome', 'France': 'Paris'}  
Rome
```



C'est très simple. Il suffit de spécifier la clé que l'on veut ajouter, et la valeur qu'on veut y mettre.

Dans le cas où on ne précise pas la valeur (double guillemets ou apostrophes), celle-ci sera vide. Si vous essayez d'afficher la valeur associée à la clé, il ne s'affichera rien.

Dans le cas où vous voudriez changer la valeur d'une clé, il suffit de faire la même chose que lorsque vous créez une clé.

# EXEMPLE

```
capitale = {'France' : 'Paris', 'Angleterre' : 'Londres', 'Espagne' : 'Madrid'}  
  
print capitale  
  
capitale['Italie'] = ''  
  
print capitale  
print capitale['Italie']  
  
capitale['Italie'] = 'UneVille'  
  
print capitale  
print capitale['Italie']  
  
capitale['Italie'] = 'Rome'  
  
print capitale  
print capitale['Italie']
```

```
{'Espagne': 'Madrid', 'Angleterre': 'Londres', 'France': 'Paris'}  
{'Espagne': 'Madrid', 'Angleterre': 'Londres', 'Italie': '', 'France': 'Paris'}  
  
{'Espagne': 'Madrid', 'Angleterre': 'Londres', 'Italie': 'UneVille', 'France': 'Paris'}  
UneVille  
{'Espagne': 'Madrid', 'Angleterre': 'Londres', 'Italie': 'Rome', 'France': 'Paris'}  
Rome
```

# SUPPRIMER UN ÉLÉMENT D'UN DICTIONNAIRE

## Fonction `del`

Reprenons notre dictionnaire capitale précédent.

```
capitale = {'France' : Paris, 'Angleterre' : Londres, 'Espagne' : Madrid, 'Italie' :  
Rome}
```

Pour supprimer la clé Italie (et sa valeur par conséquent), on procède comme suit :

```
del capitale['Italie']
```

# MAIS ESSAYONS !

```
capitale = {'France' : 'Paris', 'Angleterre' : 'Londres', 'Espagne' : 'Madrid'}  
  
print capitale  
  
capitale['Italie'] = 'Rome'  
  
print capitale  
  
del capitale['Italie']  
  
print capitale
```

```
{'Espagne': 'Madrid', 'Angleterre': 'Londres', 'France': 'Paris'}  
{'Espagne': 'Madrid', 'Angleterre': 'Londres', 'Italie': 'Rome', 'France': 'Paris'}  
{'Espagne': 'Madrid', 'Angleterre': 'Londres', 'France': 'Paris'}
```

# EFFACER LE DICTIONNAIRE

Pour effacer une dictionnaire, on peut utiliser la fonction **del** vue précédemment sur chaque élément du dictionnaire, ou simplement utiliser la fonction **clear()** comme suit :

**capitale.clear()**

On peut également affecter des {} vides à notre dictionnaire, qui l'efface entièrement :

**capitale = {}**

# ESSAYONS ENCORE !

```
capitale = {'France' : 'Paris', 'Angleterre' : 'Londres', 'Espagne' : 'Madrid'}
```

```
print capitale
```

```
capitale.clear()
```

```
print capitale
```

```
capitale = {}
```

```
{'Espagne': 'Madrid', 'Angleterre': 'Londres', 'France': 'Paris'}  
{}
```

# FONCTIONS UTILES SUR LES DICTIONNAIRES

Soit le dictionnaire suivant :

```
Maison = {'Salon' : 1, 'Chambre' : 2, 'Salle de bain' : 2, 'Cuisine' : 1}
```

# RÉCUPÉRER LES CLÉS ET/OU LES VALEURS

Il est possible de récupérer uniquement les clés et/ou les valeurs du dictionnaire à l'aide de 3 fonctions simples :

**keys()** : génère une liste qui ne contient que les clés du dictionnaire

**values()** : génère une liste qui ne contient que les valeurs du dictionnaire

**items()** : génère une liste de tuples qui contient les couples clé/valeurs du dictionnaire

# VOYONS CELA

```
Maison = {'Salon' : 1, 'Chambre' : 2, 'Salle de bain' : 2, 'Cuisine' : 1}

Pieces = Maison.keys()
NombrePieces = Maison.values()
LesDeux = Maison.items()

print Pieces
print NombrePieces
print LesDeux
```

```
['Salle de bain', 'Salon', 'Chambre', 'Cuisine']
[2, 1, 2, 1]
[('Salle de bain', 2), ('Salon', 1), ('Chambre', 2), ('Cuisine', 1)]
```

# AUTRE FONCTION UTILE

## Fonction `has_key( )`

Cette fonction permet de vérifier dans un dictionnaire si la clé (précisée dans les parenthèses) existe ou non.

```
un_dictionnaire.has_key(une_clé)
```

Elle renvoie la valeur **True** si la clé existe. **False** sinon.

Note : le terme « renvoie » peu paraître flou. Il sera abordé dans la partie Définition des fonctions. Retenez juste que si vous décidez de mettre un **print** devant, il va d'afficher **True** ou **False**.

# TESTONS !

```
Maison = {'Salon' : 1, 'Chambre' : 2, 'Salle de bain' : 2, 'Cuisine' : 1}  
  
print Maison.has_key('Salon')  
print Maison.has_key('Garage')
```

```
True  
False
```

## AUTRE MANIÈRE DE VÉRIFIER L'EXISTENCE D'UNE CLÉ

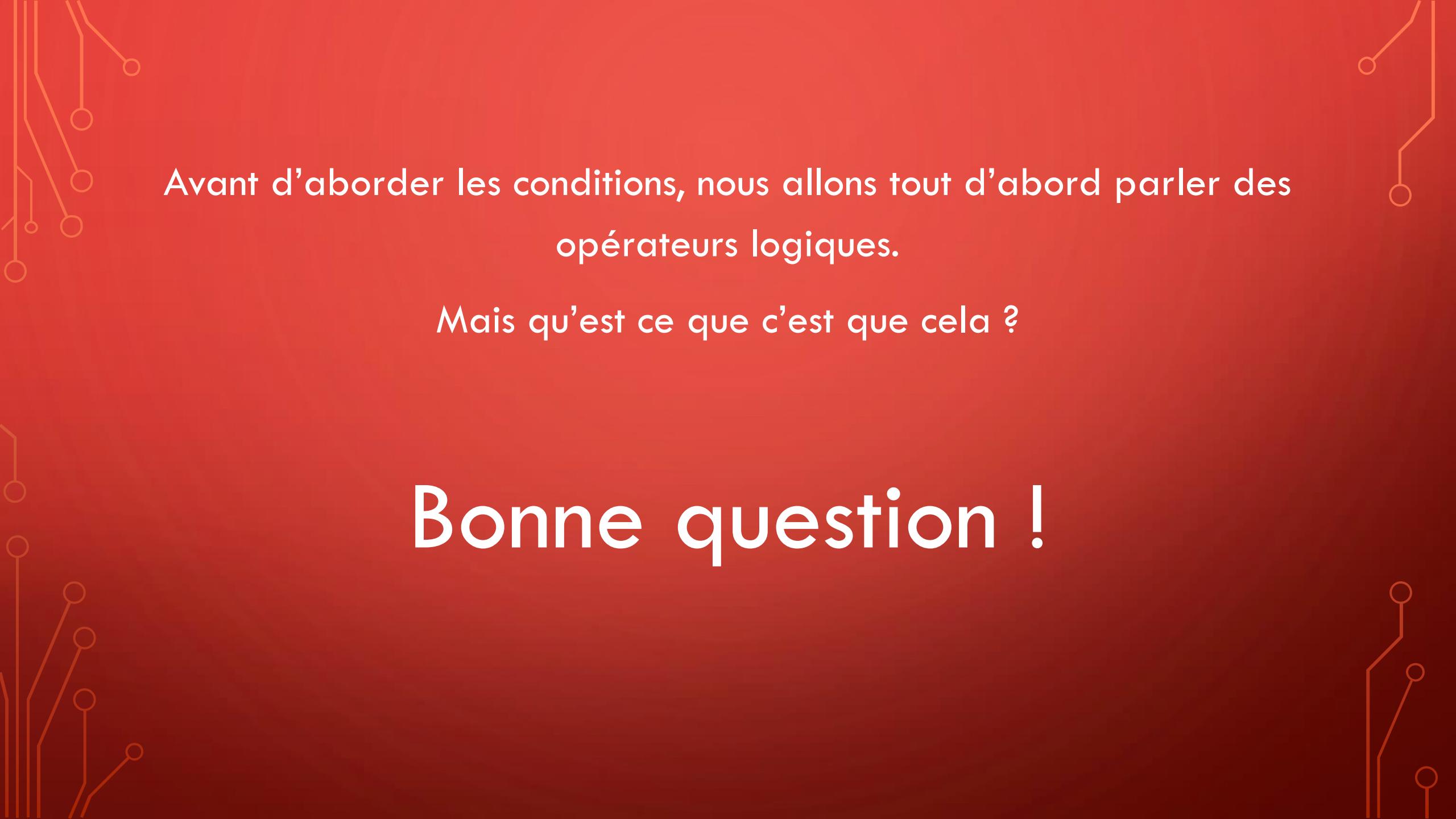
```
capitale = {'France' : 'Paris', 'Angleterre' : 'Londres', 'Espagne' : 'Madrid'}  
  
print 'France' in capitale  
print 'Italie' in capitale
```

```
True  
False
```

# EXERCICES

- 1) Soit votre ordinateur : `{'ecran' : 1, 'clavier' : 1, 'souris' : 1, 'unite centrale' : 1}`  
Possédez-vous des « enceintes » ?  
Combien de « souris » avez-vous ?
- 2) Ajouter 1 « webcam » et 2 « enceintes » à votre ordinateur
- 3) Enlever votre « ecran » et ajouter 1 « super ecran »
- 4) Débarrassez-vous de (effacer) votre ordinateur, et en créer un nouveau
- 5) De quoi est composé votre nouvel ordinateur ? Combien d'objets composent votre ordinateur ?

# Conditions et Opérateurs logiques



Avant d'aborder les conditions, nous allons tout d'abord parler des opérateurs logiques.

Mais qu'est ce que c'est que cela ?

Bonne question !

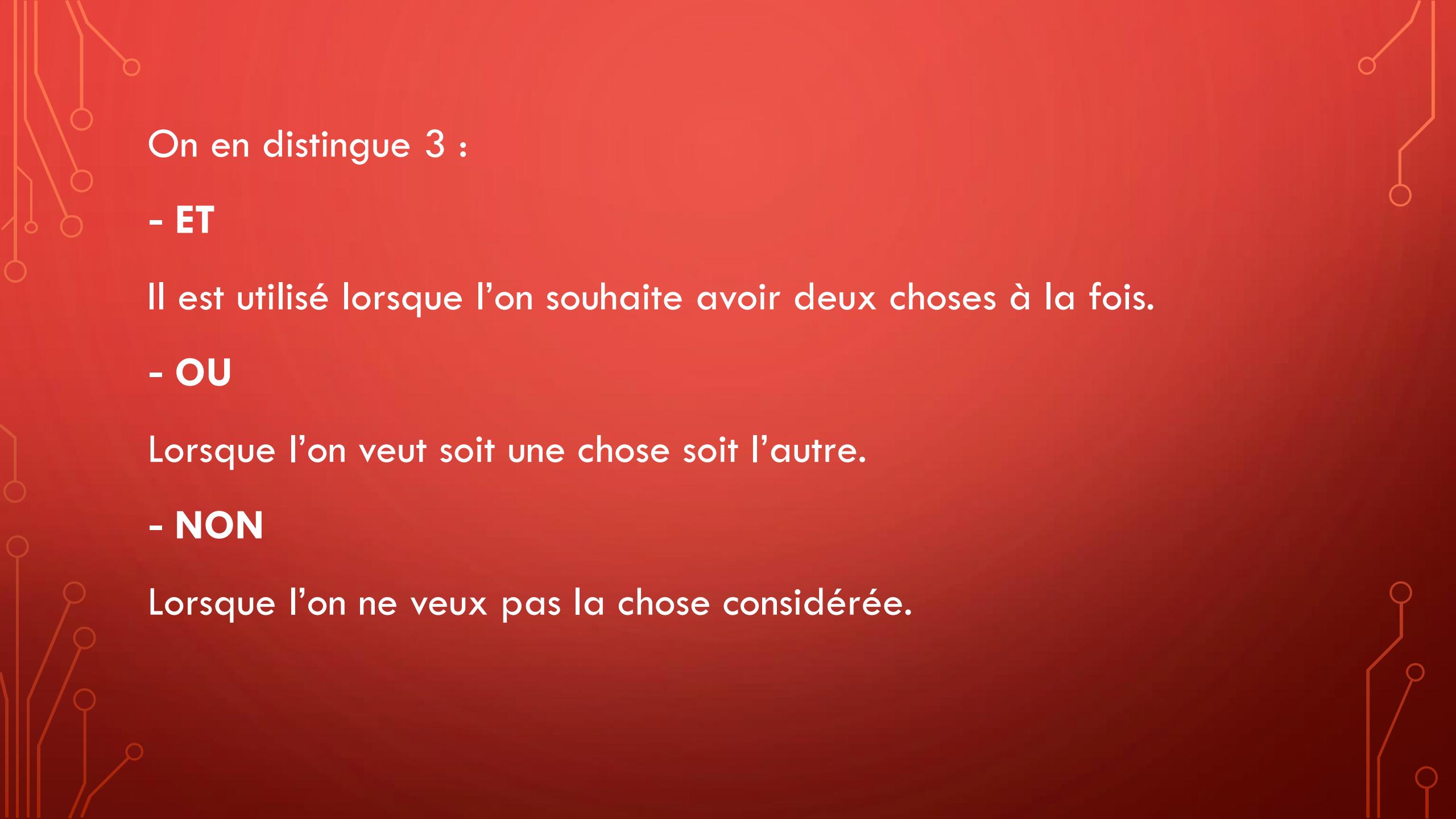
# OPÉRATEURS LOGIQUES

« Je veux ça **ET** ça. Ah **ET** je veux ça, **OU** sinon ça ».

« Par contre je **NE VEUX PAS** ça. »

Ça y est ?

Vous avez compris ?



On en distingue 3 :

- **ET**

Il est utilisé lorsque l'on souhaite avoir deux choses à la fois.

- **OU**

Lorsque l'on veut soit une chose soit l'autre.

- **NON**

Lorsque l'on ne veux pas la chose considérée.

# EXEMPLES !

Bleu et blanc

Le truc doit être bleu **ET** blanc.

Rouge ou noir

Le truc doit être **soit** rouge **soit** noir.

Non difficile

Le truc **ne doit pas** être difficile (équivalent à être facile).

# EN PYTHON

Rappelez vous qu'en C, vous écriviez les opérateurs logiques de cette manière :

**ET : &&**

**OU : ||**

**NON : !**

En python on les écriera de cette manière :

**ET : and**

**OU : or**

**NON : not ou !**

Avouez que c'est plus simple à retenir !

À quoi cela va-t-il vous servir ?

Vous le verrez quelques diapositives plus loin, que les opérateurs logiques sont très utiles dans les Conditions. Lorsque vous devez faire un choix entre ça **OU** ça, ou forcément ça **ET** ça, connaître ces éléments vous seront très utiles.

# ÉLÉMENTS DE COMPARAISONS

C'est comme des opérateurs logiques sans être des opérateurs logiques. On s'en sert beaucoup en mathématique lorsque l'on veut comparer deux éléments, savoir si l'un est plus grand que l'autre, ou s'ils sont égaux, ...

Vous avez trouvé ?

Et bien ces éléments de comparaisons sont les suivants :

`>, <, >=, <=, ==, !=`

À première vue, je viens de taper ce qui semble être une suite d'étranges symboles. Ici, ils sont écrits tels que vous devez les écrire en langage informatique. On y regardant de plus près on en reconnaît quelques-uns.

Mais voyons les de plus près.

< : strictement inférieur à  
> : strictement supérieur à

Rien de nouveau, on connaît tous.

$\geq$  : inférieur ou égal à

$\leq$  : supérieur ou égal à

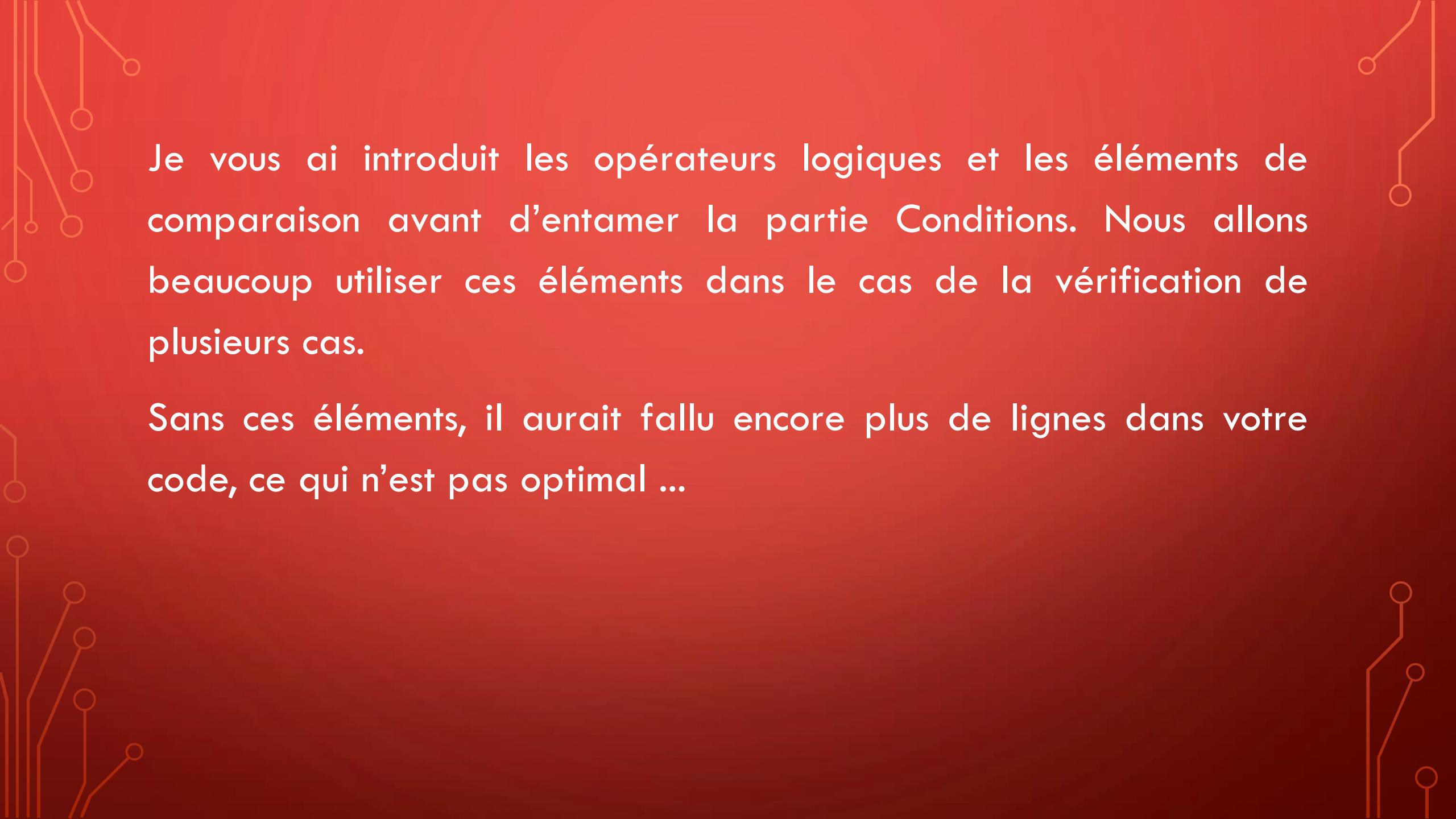
Facile à deviner ! Un signe égal juste après les symboles que vous êtes censé connaître. C'est pas sorcier !

$=$  : égal

Si vous remontez aux diapos **Affectation** (20 et 21), je vous ai précisé que le « = » signifiait une affectation et non une égalité. Si vous voulez vérifier une égalité, alors il faut utiliser le symbole « == ».

$\neq$  : différent de

Un opérateur logique NON devant un signe égal ... Ah ça veut dire que cela ne doit pas être égal, mais différent ! Et oui c'est comme ça ! Point ! Attention ce n'est pas « != ».



Je vous ai introduit les opérateurs logiques et les éléments de comparaison avant d'entamer la partie Conditions. Nous allons beaucoup utiliser ces éléments dans le cas de la vérification de plusieurs cas.

Sans ces éléments, il aurait fallu encore plus de lignes dans votre code, ce qui n'est pas optimal ...

# Les conditions

Si demain je réussи mon examen, alors je vais m'acheter une glace !

Si je gagne 20 euros demain, je t'en donnerai la moitié.

S'il n'est pas âgé de plus de 16, il ne conduit pas,

Voici 3 conditions !

Un peu plus d'explications

...

# LES CONDITIONS

En programmation, on impose des conditions partout. C'est ce qui nous permet d'avancer dans notre programme.

Il faut que telle chose soit vraie (ou fausse) pour pouvoir avancer. Sinon, on n'avance pas.

Sans conditions, le programme serait linéaire, et assez vide.

# COMMENT ÇA MARCHE ?

Syntaxe :

```
if condition:  
    fais ce qui est ici  
    fais ce qui est également ici
```

Exemple de base de ce qu'est une condition. Mais analysons tout de suite les différentes parties de ce qui est présenté ci-dessus.

```
if condition:
```

Vous l'avez sans doute deviné, le « if » signifie « si ». Cette ligne nous permet de vérifier la condition « condition ». Si la condition « condition » est vraie, alors on effectue l'instruction située juste en-dessous. N'oubliez pas les « : » !

```
fais ce qui est ici  
fais ce qui est également ici
```

Si la condition est vérifiée, ces instructions sont alors exécutés. Sinon, on ne fait rien, on passe.

# ATTENTION !

Vous avez sans doute remarqué que les deux éléments ne sont pas alignés. Il y a ce qu'on appelle une « indentation » au « bloc » des instructions. Tout ce qui est sous la ligne du « if » et qui est indenté sera exécuté. Si votre commande est aligné avec le « if », elle sera exécutée sans qu'il y ai vérification de la condition.

Ok pour l'instant c'est cool. Mais lorsqu'il y a un « si », il y a toujours un « sinon » (ou pas). En python nous l'écrivons ainsi :

```
if condition:  
    fais ceci  
  
else:  
    fais cela
```

L'instruction figurée sous le « **else** » (qui veut dire « **sinon** »), et indenté sera exécutée dans le cas où la condition du « **if** » n'est pas valide. C'est tout simple !

# MAIS VOYONS TOUT DE SUITE UN EXEMPLE

```
nombre = 7  
  
if nombre > 5:  
    print "Le nombre est supérieur à 5"  
  
else:  
    print "Le nombre est inférieur à 5"
```

```
Le nombre est supérieur à 5
```

## DANS LE CAS DE PLUSIEURS CONDITIONS

« Si tu es sage et que tu m'apporte une bonne note, tu pourras avoir une glace ! »

« Si tu as des chaussures ou des tong, tu peux sortir ! »

Dans le cas où plusieurs conditions sont à vérifier, on peut procéder de deux façons.

- Soit on utilise deux conditions l'une sous l'autre :

```
if condition1:  
    if condition2:  
        fais ça
```

Ce qui à la fin revient à ajouter un « if » en plus.

- Soit on utilise les opérateurs logiques vu précédemment :

```
if condition1 and condition2:  
    fais ça
```

Le deuxième cas nous fait économiser une ligne et est beaucoup plus simple.

# VOYONS UN EXEMPLE

```
nombre1 = 7
nombre2 = 5

if nombre1 > 3:
    if nombre2 > 3:
        print "Les nombres sont supérieurs à 3"

else:
    print "Au moins un des deux nombres est inférieur à 3"
```

Ou bien

```
if nombre1 > 3 and nombre2 > 3:
    print "Les deux nombres sont supérieurs à 3"

else:
    print "Au moins un des deux nombres est inférieur à 3"
```

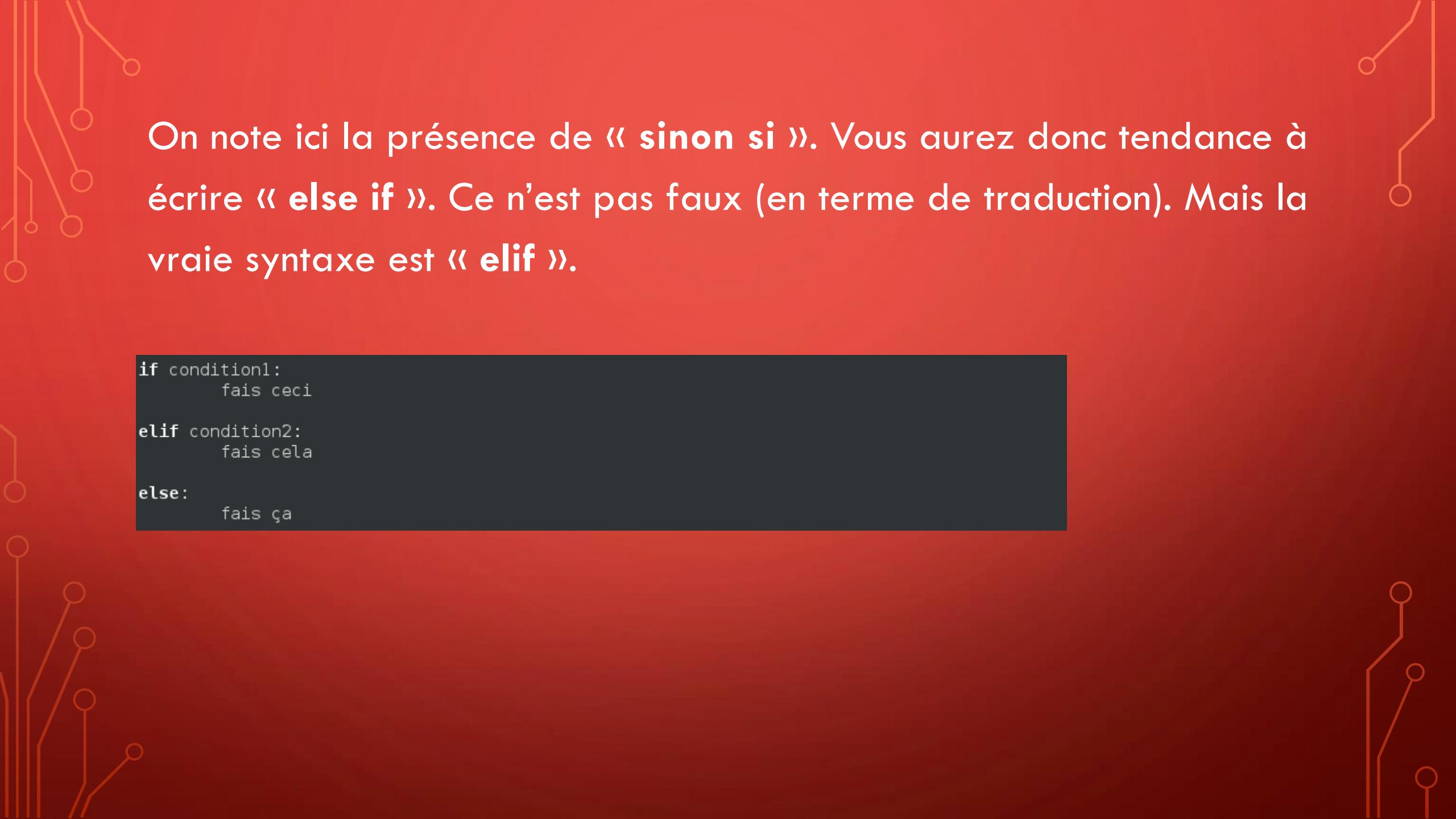
Les nombres sont supérieurs à 3

## DANS LE CAS DE PLUSIEURS CONDITIONS À LA SUITE

« Si tu es sage tu auras un bonbon. Sinon si tu es méchant tu auras une fessée. Sinon, tu auras la moitié d'un bonbon, et une demi-fessée. »

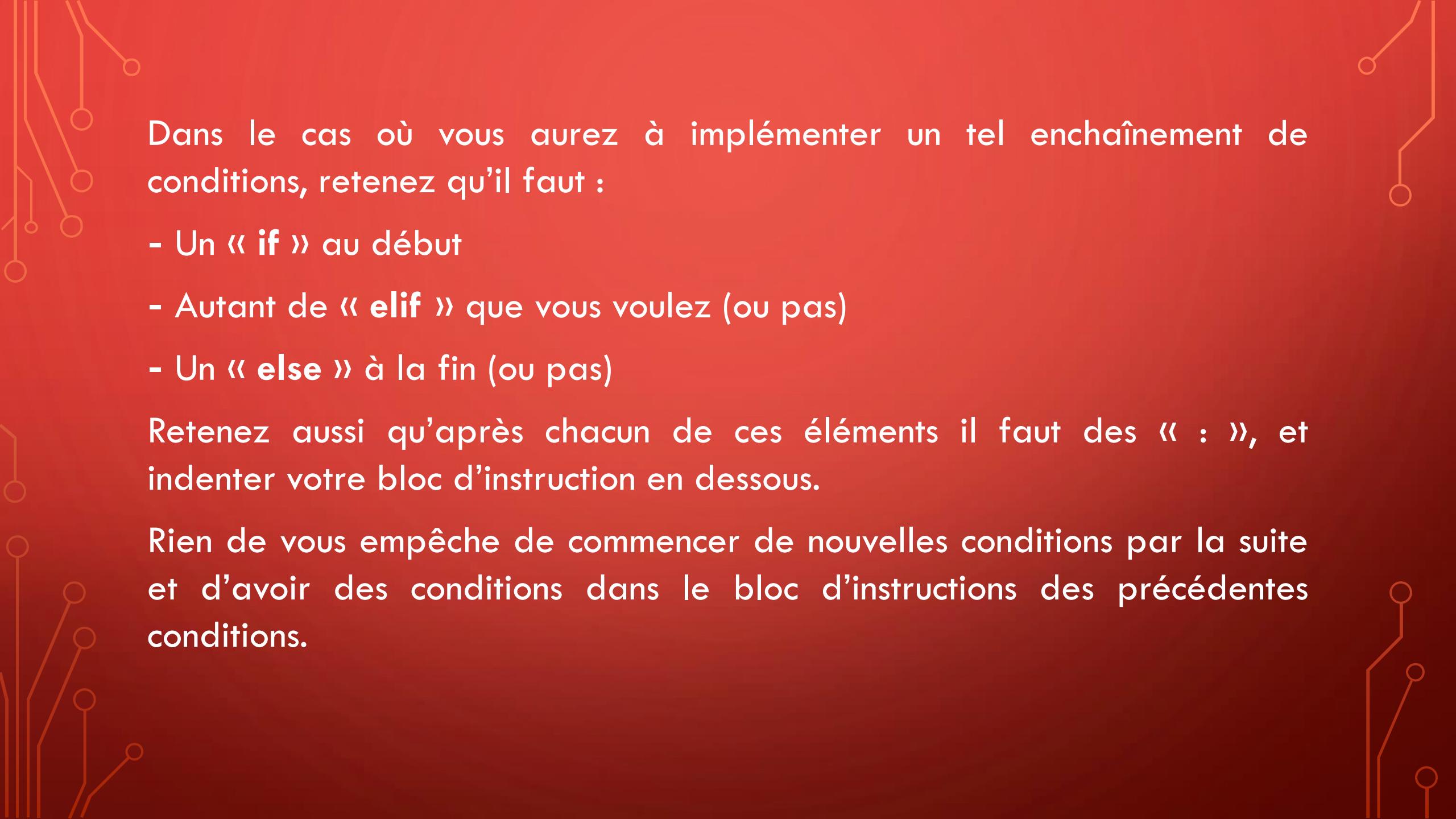
Notez l'enchainement des conditions.

Si condition1 : instructions1, sinon si condition2 : instructions2, sinon : instructions3.



On note ici la présence de « **sinon si** ». Vous aurez donc tendance à écrire « **else if** ». Ce n'est pas faux (en terme de traduction). Mais la vraie syntaxe est « **elif** ».

```
if condition1:  
    fais ceci  
  
elif condition2:  
    fais cela  
  
else:  
    fais ça
```



Dans le cas où vous aurez à implémenter un tel enchaînement de conditions, retenez qu'il faut :

- Un « **if** » au début
- Autant de « **elif** » que vous voulez (ou pas)
- Un « **else** » à la fin (ou pas)

Retenez aussi qu'après chacun de ces éléments il faut des « **:**  », et indenter votre bloc d'instruction en dessous.

Rien de vous empêche de commencer de nouvelles conditions par la suite et d'avoir des conditions dans le bloc d'instructions des précédentes conditions.

# EXEMPLE !

```
nombre1 = 7
nombre2 = 5

if nombre1+nombre2 > 20:
    print "La somme des deux nombres est supérieure à 20"

elif nombre1+nombre2 == 12:
    print "La somme des deux nombres est égale à 12"

else:
    print "La somme des deux nombres est inférieure à 20 et n'est pas égale à 12"
```

```
La somme des deux nombres est égale à 12
```

# EXERCICES

1) Ecrire un programme qui demande à l'utilisateur 3 nombres (entiers ou réels), affiche « OK » si la somme des trois nombres est supérieure à 100, sinon affiche « PAS OK ».

2) Voici vos notes : {‘maths’ : 10, ‘histoire’ : 12, ‘français’ : 14, ‘langues’ : 11, ‘biologie’ : 5, ‘physique chimie’ : 2}

Affichez « Tu passes » si votre moyenne est supérieure ou égale à 10. Si elle est située entre 8 et 10 (inclus), affichez « Seconde chance accordée ». Sinon affichez « recalé ».

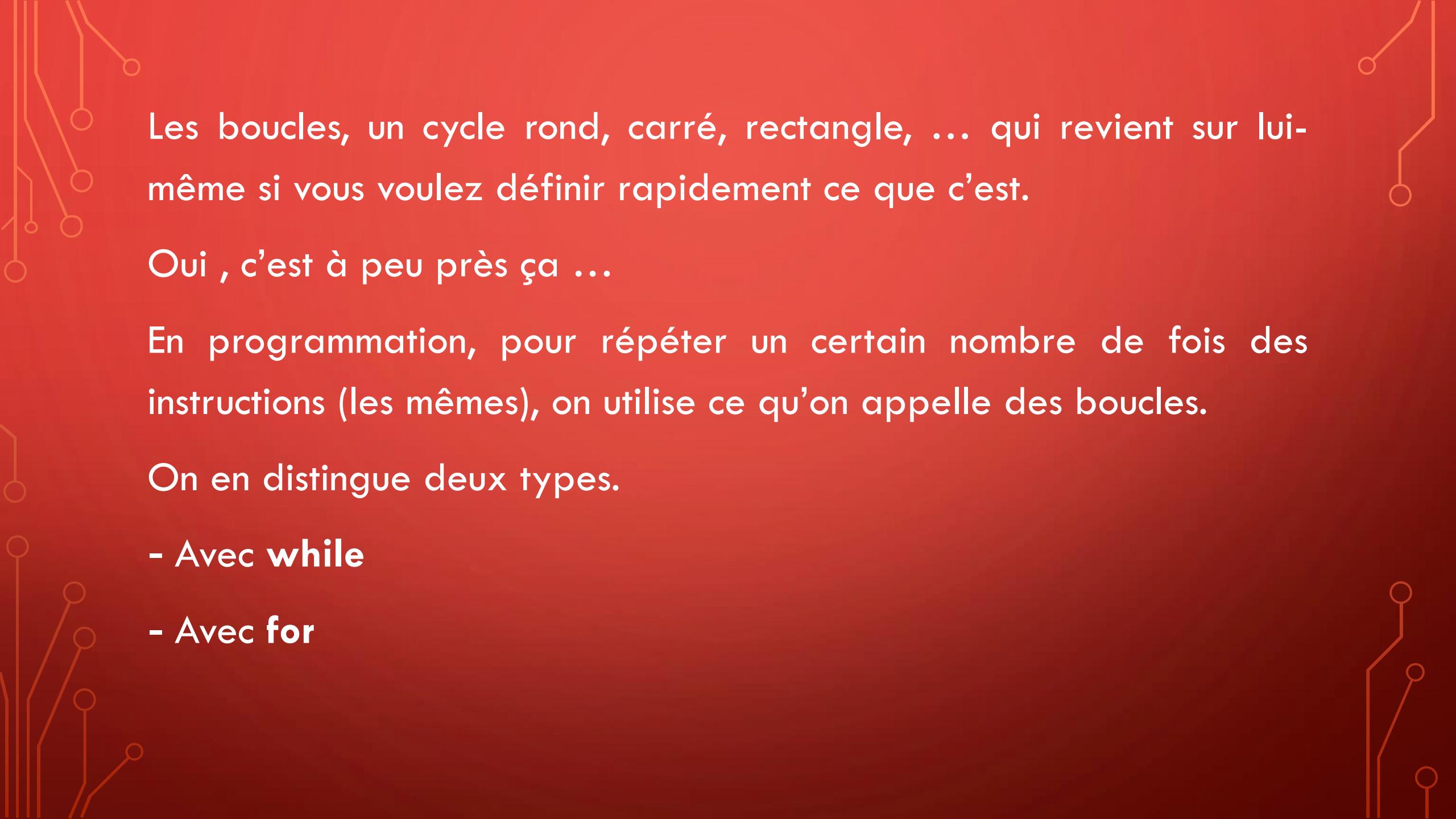
3) Soit votre classe d'élèves : [‘Maxime’, Benjamin’, ‘Pierre’, ‘Marianne’, ‘Manon’, ‘Denis’]

Votre classe d'élèves est composé de 7 personnes.

Ecrire un programme qui affiche « Le cours commence » si tout le monde est là. Sinon affichez « On attend ».

Faites rentrer ‘Toto’ dans la classe. Est-ce que le cours va commencer ?

# Les boucles



Les boucles, un cycle rond, carré, rectangle, ... qui revient sur lui-même si vous voulez définir rapidement ce que c'est.

Oui , c'est à peu près ça ...

En programmation, pour répéter un certain nombre de fois des instructions (les mêmes), on utilise ce qu'on appelle des boucles.

On en distingue deux types.

- Avec **while**
- Avec **for**

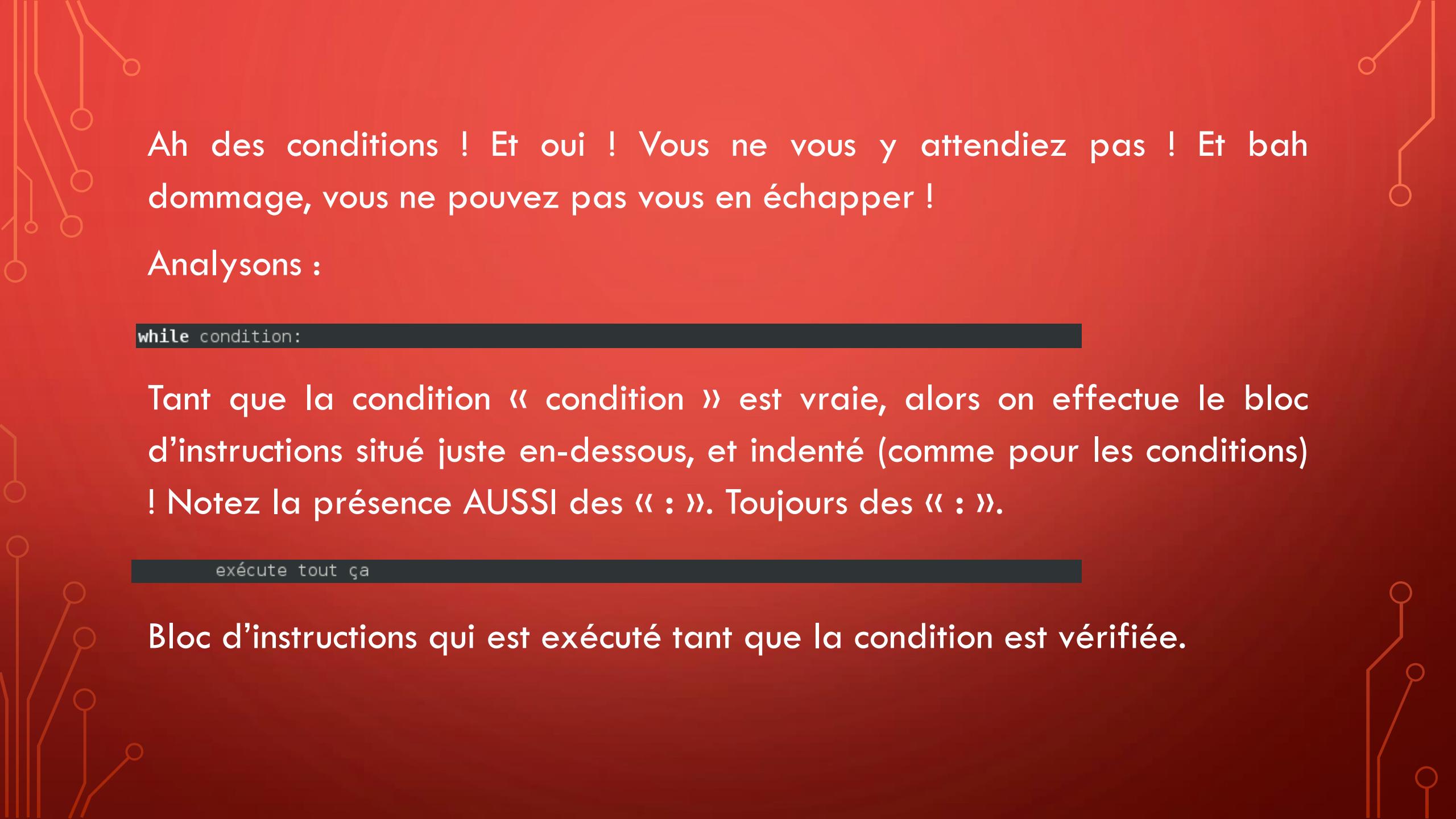
# WHILE

Qui veut dire « **tant que** ».

La boucle **while** est généralement utilisé dans le cas où le nombre de répétitions que l'on fait n'est pas déterminé à l'avance. On ne sait donc pas quand on s'arrête.

Syntaxe :

```
while condition:  
    exécute tout ça
```



Ah des conditions ! Et oui ! Vous ne vous y attendiez pas ! Et bah dommage, vous ne pouvez pas vous en échapper !

Analysons :

```
while condition:
```

Tant que la condition « condition » est vraie, alors on effectue le bloc d'instructions situé juste en-dessous, et indenté (comme pour les conditions) ! Notez la présence AUSSI des « : ». Toujours des « : ».

```
    exécute tout ça
```

Bloc d'instructions qui est exécuté tant que la condition est vérifiée.

# EXEMPLE

```
nombre = 2  
  
while nombre < 5:  
    print nombre  
    nombre = nombre+1  
  
print "Fin while, nombre :", nombre
```

```
2  
3  
4  
Fin while, nombre : 5
```

## ATTENTION !

Je vous ai dit qu'on utilise la boucle **while** lorsqu'on ne sait pas quand on veut s'arrêter. Tant que la condition est vérifiée, on continue nos boucles.

Toutefois, il se peut que des boucles « infinies » apparaissent par maladresse. C'est ce qui arrive lorsque la condition est tout le temps vraie. Le programme ne s'arrêtera donc jamais de tourner.

Pas de panique ! Pour interrompre la boucle, il suffit d'arrêter le programme ! Un petit CTRL+Z ou CTRL+C devrait suffire.

# EXEMPLE

```
nombre = 2

while nombre < 5:
    print nombre
```

Dans ce cas, `nombre` vaut toujours 2 et ne change pas. L'incrémentation « `nombre = nombre+1` » permet d'augmenter de 1 la variable `nombre` à chaque tour de boucle, et lorsque `nombre` vaudra 5, on ne rentrera plus dans la boucle.

```
nombre = nombre+1
```

# FOR

Littéralement « **pour** ». La boucle for est utilisée lorsque l'on sait quand on s'arrête. L'avantage de cette boucle est qu'il n'y a pas de conditions à vérifier pour répéter le bloc d'instructions.

Syntaxe:

```
for un_truc in une_liste:  
    fais ça
```

Littéralement :

Pour un\_truc dans une\_liste:

fais ça

# ANALYSONS

`un_truc` : c'est « le nom d'une variable ».

Vous pouvez choisir n'importe quoi (srlfsdf, dsfsdf45, i, x\_qs\_d, TTTT, ...). Il faut juste que ça respecte le format du nom d'une variable.

`une_liste` : une liste d'éléments.

`un_truc` va prendre toutes les valeurs situées dans la liste « `une_liste` », et ce un par un, jusqu'à la fin.

De ce fait, la boucle `for` s'arrête lorsque `un_truc` a atteint le dernier élément de la liste `une_liste`.

NOTEZ LA PRÉSENCE DES « : » ET DU  
BLOC D'INSTRUCTIONS INDENTÉ !

# COMBINAISON AVEC RANGE( )

On a vu à quoi servait la fonction **range( )**. Si vous ne vous en souvenez plus, remontez un peu (diapo 44).

Généralement pour exécuter un nombre défini de fois un bloc d'instructions, on utilisera la boucle `for` combinée avec la fonction **range( )**.

# EXAMPLE

```
for i in range(10):  
    print i
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

## AUTRE EXEMPLE

```
mots = ['Ceci', 'est', 'une', 'suite', 'de', 'mots', 'affichés', 'en', 'colonne']

for i in mots:
    print i
```

```
Ceci
est
une
suite
de
mots
affichés
en
colonne
```

# EXERCICES

1) Une pièce ne peut contenir que 30 personnes. Le nombre de personnes au départ est de 2 (minute 0). Toutes les 5 minutes, 2 personnes entrent dans la pièce. Dès que la pièce est remplie, plus personne n'entre.

Ecrire le programme qui simule cette situation. Affichez le temps au bout duquel la salle est remplie.

2) Ecrire un programme qui demande à l'utilisateur un premier nombre (entier), puis d'autres entiers qui vont s'additionner au premier. S'arrêter lorsque la somme des valeurs est supérieure à 100. Affichez le nombre de nombre qui a permis d'obtenir au moins 100.

# Définition de fonctions

# UNE FONCTION C'EST QUOI ?

Rappelez-vous, une fonction est reconnaissable de par la présence de « ( ) » après un mot (exemple : **casse( )**, n'existe pas en python, dommage). Vous pouvez vous référer à la partie « Fonctions diverses » pour plus d'informations sur ce qu'est une fonction.

Celles que l'on a utilisé précédemment (**range( )**, **print( )**, **max( )**, ...) sont tous des fonctions déjà « définies ».

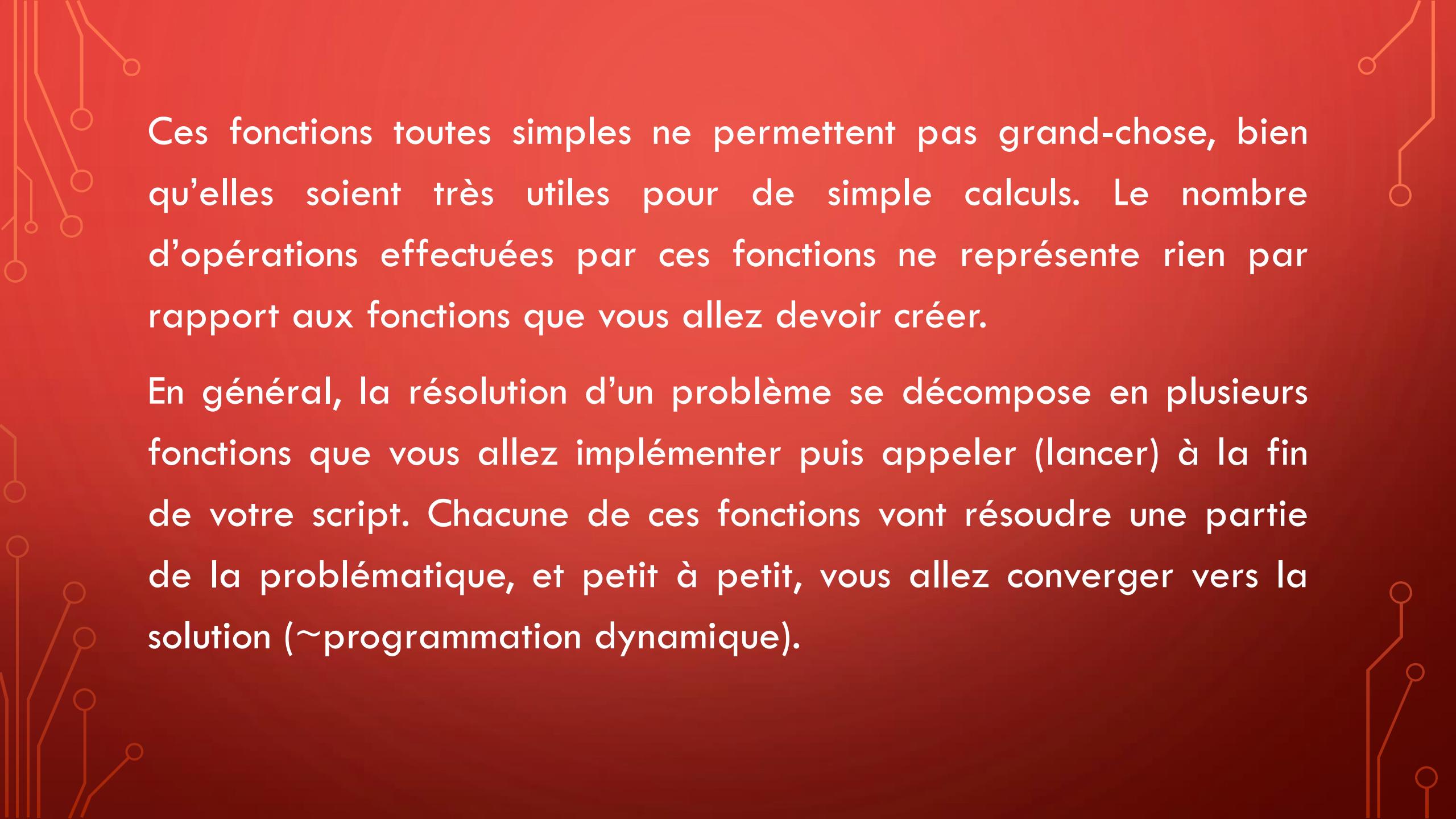
Ce que je veux dire par là c'est que ces fonctions existent déjà, et que vous n'aviez donc pas à les créer, puisqu'elles étaient déjà tout prêtes à l'emploi.

# DÉFINIR UNE FONCTION

Ce qu'on appelle par « **définir une fonction** » c'est « **créer** » une fonction.

C'est beau les fonctions déjà toutes faites. Mais derrière ces jolies petites fonctions que vous utilisez sans cesse se cachent des lignes de code secrètement gardés hors de vue de vos yeux.

Non je rigole. Les fonction comme `range( )`, `max( )`, ... sont importées depuis une des bibliothèques de python (je ne sais pas laquelle), installée par défaut, et possède à elles-mêmes des lignes de codes.



Ces fonctions toutes simples ne permettent pas grand-chose, bien qu'elles soient très utiles pour de simple calculs. Le nombre d'opérations effectuées par ces fonctions ne représente rien par rapport aux fonctions que vous allez devoir créer.

En général, la résolution d'un problème se décompose en plusieurs fonctions que vous allez implémenter puis appeler (lancer) à la fin de votre script. Chacune de ces fonctions vont résoudre une partie de la problématique, et petit à petit, vous allez converger vers la solution (~programmation dynamique).

# DÉFINIR UNE FONCTION

Pour créer/définir une fonction, c'est très simple ! Voici la syntaxe :

```
def nom_de_la_fonction(parametres):  
    instructions  
    return quelque_chose
```

# ANALYSONS

- nom\_de\_la\_fonction `nom_de_la_fonction`

C'est le nom de votre fonction, rien de surprenant, attention à ne pas utiliser les caractères interdis tels que les accents, la ponctuation, etc. Ne soyez pas surpris si le nom de votre fonction n'apparaît pas en bleu. Suivant les systèmes d'exploitation, le mode de coloration n'est pas le même.

- def `def`

Pour « define », définir donc. Toute création de fonction commence par le mot « def », devant le nom de votre fonction.

- () (   )

Vous avez sans doute reconnu les typiques parenthèses propres aux fonctions ! Bravo ! C'est dedans qu'on y insérera les paramètres. La fonction va alors les prendre et les utiliser.

- **paramètre(s)** parametres

Un ou plusieurs paramètres, selon vos besoin et comment vous allez coder votre fonction (ou arguments si vous voulez, ce serait plus logique).

- **Instructions** instructions

Les instructions que votre fonction exécutera.

- Toujours les « : » !

# RETURN

Toute fonction (presque) « retourne » quelque chose ou rien. Ce qu'on appelle par retourner, c'est ce que la fonction affichera éventuellement en sortie lorsque vous mettez un « **print** ». C'est le résultat donné par la fonction : un nombre, un mot, une liste, etc.

Si par exemple la variable **x** contient votre résultat à la fin de la fonction il faut la retourner en tapant :

```
return x
```

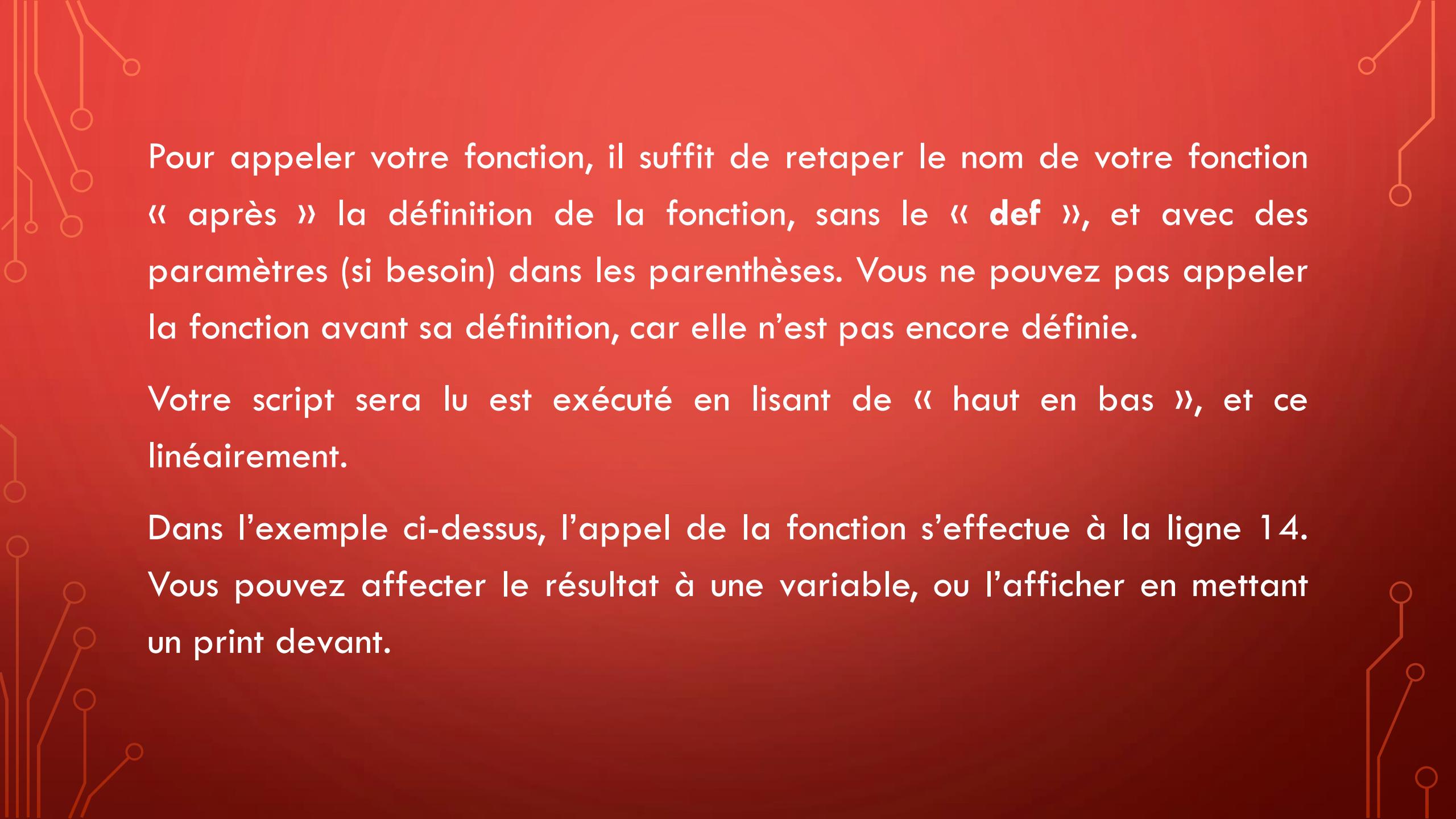
```
return x
```

La fonction va ainsi vous donner **x** comme résultat.

# UN PREMIER EXEMPLE

```
6 def multiplie_par_deux(nombre):  
7  
8     nombre2 = nombre*2  
9  
10    return nombre2  
11  
12  
13  
14 print multiplie_par_deux(4)
```

```
8
```



Pour appeler votre fonction, il suffit de retaper le nom de votre fonction « après » la définition de la fonction, sans le « **def** », et avec des paramètres (si besoin) dans les parenthèses. Vous ne pouvez pas appeler la fonction avant sa définition, car elle n'est pas encore définie.

Votre script sera lu est exécuté en lisant de « haut en bas », et ce linéairement.

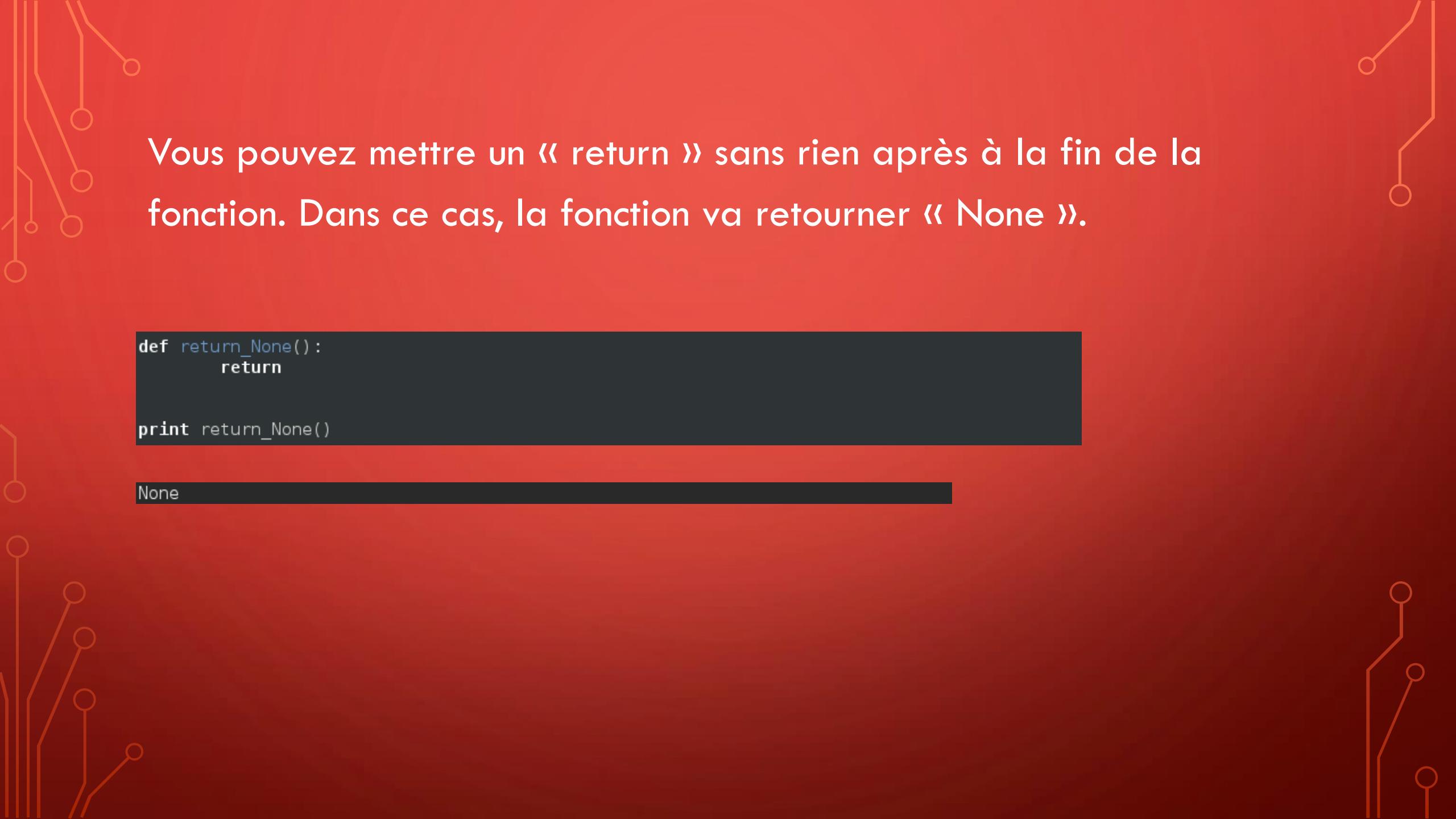
Dans l'exemple ci-dessus, l'appel de la fonction s'effectue à la ligne 14. Vous pouvez affecter le résultat à une variable, ou l'afficher en mettant un **print** devant.

Dans le cas où la fonction ne retourne pas de résultat (fonction d'affichage par exemple), la présence ou non d'un « return » à la fin de votre fonction n'est pas nécessaire.

```
def affiche(nombre):
    print nombre
    print nombre
    print nombre
    print nombre
    print nombre
    print nombre
```

```
affiche(4)
```

```
4
4
4
4
4
4
```



Vous pouvez mettre un « return » sans rien après à la fin de la fonction. Dans ce cas, la fonction va retourner « None ».

```
def return_None():
    return

print return_None()
```

```
None
```

## REMARQUE

Lorsque vous appelez la fonction « **print** » dans votre fonction, nul besoin de mettre un **print** devant l'appel de votre fonction. En effet, en appelant votre fonction, toutes les instructions vont s'exécuter. Les fonctions « **print** » dans votre fonction vont afficher dans le terminal ce qu'elles sont censées afficher.

Mettre un **print** devant l'appel de votre fonction ne permettra que d'afficher le résultat renvoyé par le « **return** ».

# ILLUSTRATION

```
def UneFonction():
    print "Un truc"
    print "Un autre truc"

    return "Un dernier truc"

UneFonction()
```

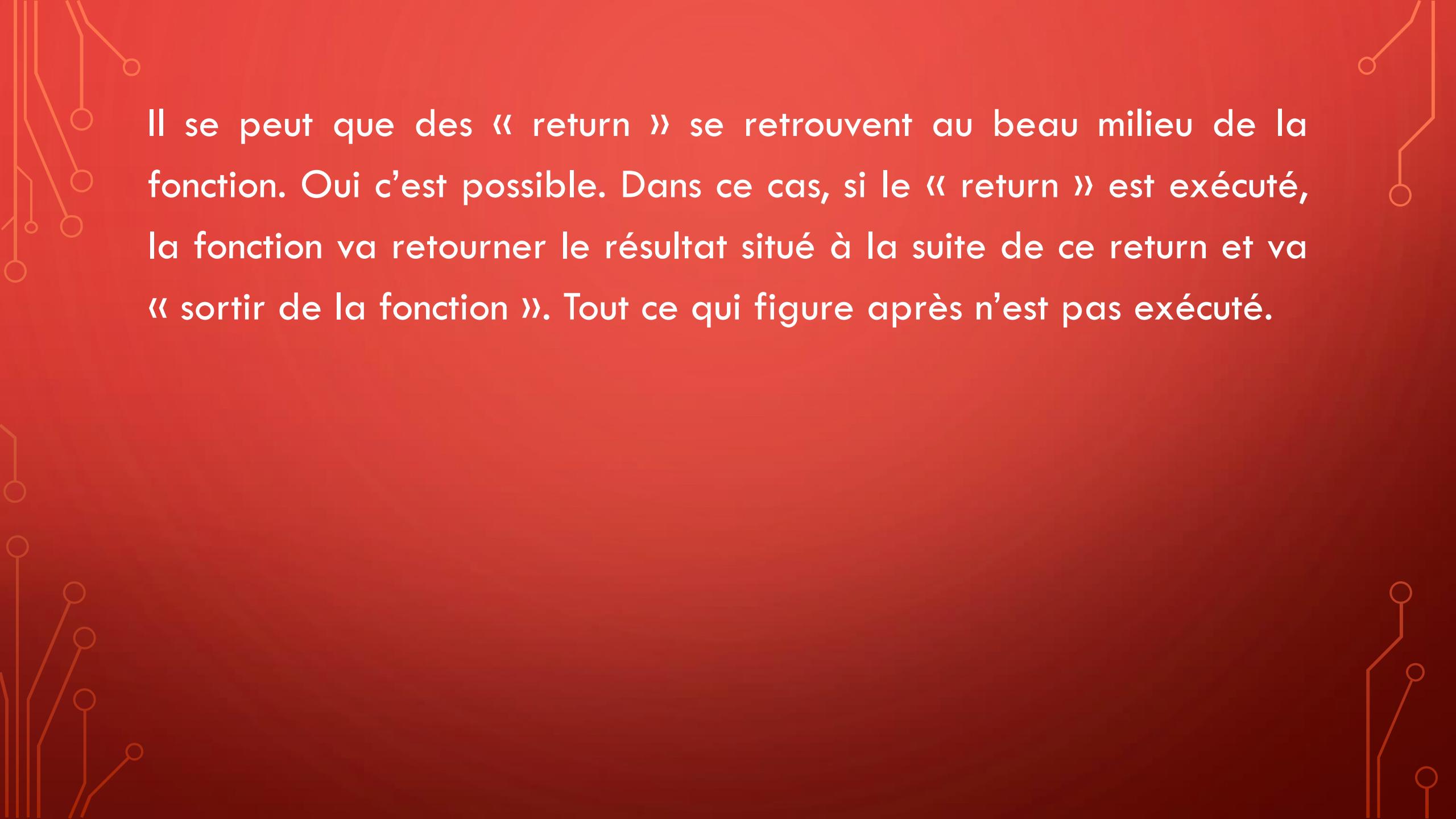
```
Un truc
Un autre truc
```

```
def UneFonction():
    print "Un truc"
    print "Un autre truc"

    return "Un dernier truc"

print UneFonction()
```

```
Un truc
Un autre truc
Un dernier truc
```



Il se peut que des « return » se retrouvent au beau milieu de la fonction. Oui c'est possible. Dans ce cas, si le « return » est exécuté, la fonction va retourner le résultat situé à la suite de ce return et va « sortir de la fonction ». Tout ce qui figure après n'est pas exécuté.

# ILLUSTRATION

```
def fonction(nombre):
    if nombre == 1:
        return "On s'arrête ici"

    else:
        return "On s'arrête là"

    print "OleyOleyOley"

    return "Pourquoi pas ici ?"

print fonction(1)
```

On s'arrête ici

```
def fonction(nombre):
    if nombre == 1:
        return "On s'arrête ici"

    else:
        return "On s'arrête là"

    print "OleyOleyOley"

    return "Pourquoi pas ici ?"

print fonction(98765)
```

On s'arrête là

# RETOUR SUR LES PARAMÈTRES

Paramètres ou arguments, comme vous voulez. Mais on dirait plus « arguments » que paramètres, bien que cela porte à confusion. C'est ce que vous mettez dans les parenthèses lors de la création de votre fonction et lors de son appel.

Lorsque vous créez une fonction, vous créez des variables « locales » à la fonction. Ces variables seront donc « propres » à la fonction et en aucun cas ne pourraient avoir « d'interaction » avec « l'extérieur ».

# REGARDONS CET EXEMPLE

```
def met_au_carre(x):  
    p = x*x  
    return p  
  
nombre = 2  
  
print met_au_carre(nombre)
```

4



Les variables `x` et `p` sont propres à la fonction. Si vous essayez d'afficher `x` et `p` en dehors de la fonction, votre ordinateur vous signifiera que ces variables « n'existent pas ».

De même si vous essayez d'afficher « `nombre` » dans la fonction, cela aura le même effet.

Dans cette fonction, la variable « `nombre` » est passée en argument de la fonction `met_au_carre()`, et lors de son appel, va prendre la place de la variable `x`.



# ILLUSTRATION

```
def met_au_carre(x):  
    p = x*x  
    return p  
  
nombre = 2  
  
print met_au_carre(nombre)
```

Ce qui se passe dans la fonction :

```
p = nombre*nombre  
return p
```

Équivaut à

```
p = 2*2  
return p
```

Notez que vous pouvez votre fonction peut prendre plusieurs arguments. Exemple :

```
def affiche_les_arguments(x, y, z):
    print x
    print y
    print z

    return x, y ,z

nombre1 = 1
nombre2 = 2
nombre3 = 3

N = affiche_les_arguments(nombre1, nombre2, nombre3)
```

```
1
2
3
```

En C nous ne pouvions, à la fin d'une fonction ne retourner qu'une seule variable. En python, nous pouvons en retourner plusieurs. Il suffit de les séparer par une virgule.

Dans l'exemple ci-dessus, nombre1 va prendre la place de x dans la fonction, nombre2 va prendre la place de y, et nombre3 va prendre la place de z dans la fonction.

Nous avons vu dans la partie affectation/variables que vous pouvez affecter vos variables comme suit :

A, B, C = 1, 2, 3

Dans l'exemple ci-dessus, N contient les 3 variables. Devinez quel sera son type ?

## Une liste/tuple !

Pour vérifier, affichez la variable N avec un print, ou afficher son type avec la fonction **type( )**. Pour accéder aux valeurs, utilisez les « [ ] » comme vu précédemment dans la section « listes ».

Dans le cas où vous voudriez affecter les résultats de la fonction à 3 variables différentes, suivez l'exemple dans la diapositive précédente ou comme suit :

```
N1, N2, N3 = affiche_les_arguments(nombre1, nombre2, nombre3)
```

Vous pouvez ensuite les afficher un à un.

# ILLUSTRATION

```
N = affiche_les_arguments(nombre1, nombre2, nombre3)

print type(N)
print N
```

```
1
2
3
<type 'tuple'>
(1, 2, 3)
```

```
N1, N2, N3 = affiche_les_arguments(nombre1, nombre2, nombre3)

print N1
print N2
print N3
```

```
1
2
3
1
2
3
```

# APPEL DE FONCTION DANS UNE FONCTION

Lors de la création de votre fonction, rien ne vous empêche d'appeler une autre fonction. Respectez seulement l'ordre hiérarchique d'exécution de votre script. Si vous appelez une fonction avant de la définir, et bien ça marche pas !

# EXEMPLE

```
def met_au_carre(nombre):  
    return nombre*nombre  
  
def appelle_la_fontion_avant(nombre):  
    resultat = met_au_carre(nombre)  
    return resultat  
  
print appelle_la_fontion_avant(4)
```

16

# AUTRE EXEMPLE

```
8 print appelle_la_fontion_avant(4)
9
10
11 def met_au_carre(nombre):
12
13     return nombre*nombre
14
15
16 def appelle_la_fontion_avant(nombre):
17
18     resultat = met_au_carre(nombre)
19
20     return resultat
```

```
Traceback (most recent call last):
  File "script.py", line 8, in <module>
    print appelle_la_fontion_avant(4)
NameError: name 'appelle_la_fontion_avant' is not defined
```

## REMARQUE

Lorsque vous appelez une fonction dans une autre, l'ordre de définition de vos fonctions n'importe pas. En effet, si vousappelez une fonction1 dans une fonction2 et que la fonction1 est définie après la fonction2, et bien ça fonctionne !

# ILLUSTRATION

```
def appelle_la_fontion_avant(nombre):  
    resultat = met_au_carre(nombre)  
    return resultat  
  
def met_au_carre(nombre):  
    return nombre*nombre  
  
print appelle_la_fontion_avant(4)
```

16

# AUTRE FAÇON DE CRÉER UNE FONCTION

Il est possible de définir une fonction autre qu'avec le mot clé **def**. Comment me direz-vous ? Simplement avec le mot clé **lambda**.

Mais pourquoi deux façons de créer des fonctions ?

Utiliser **def** permet de réaliser des grosses fonctions techniques et élaborées alors que **lambda** s'utilise pour des fonctions petites et simples qui tiennent en une ligne (pas une ligne de 1000000000000 de caractères bien sûr).

La syntaxe est la suivante :

```
une_fonction = lambda paramètre(s) : résultat
```

La fonction **lambda** est affectée à une variable, variable qui elle-même deviendra la fonction, soit la fonction **une\_fonction()**. Tout de suite après le mot clé **lambda** sont placées un ou plusieurs paramètres, « **lambda x :** » équivaudra à « **def ...(x) :** ». Ce qui suit après les « **:** » est la valeur renournée par la fonction.

Pour utiliser la fonction, il suffit d'appeler **une\_fonction()**, tout comme vous appelleriez une fonction créée avec **def**.

```
fois_deux = lambda x : x*2
print fois_deux(4)

add = lambda x,y : x+y
print add(3,8)
```



```
def fois_deux(x) :
    return x*2

def add(x,y) :
    return x+y
```

# EXERCICES

- 1) Ecrire une fonction qui prend en arguments 5 entiers et retourne leur produit.
- 2) Soit la liste suivante : [1, 5.1, 65, 4.98, 1234, 98.7]  
Ecrire une fonction qui prend en entrée une liste, et supprime toutes valeurs réelles de la liste, et renvoie la liste.
- 3) Ecrire une fonction qui demande à l'utilisateur un nombre non défini de variables et les mets dans une liste. Si la variable est un entier ou un réel, la convertir comme il le faut. Si l'utilisateur entre -1, la fonction d'arrête et renvoie la liste.
- 4) Ecrire une fonction avec le mot clé lambda, prenant en paramètres 2 variables x et y et retourne la variable la plus grande. Si elles sont égales, retourne une des deux variables.

# Modules/Bibliothèques

# QU'EST-CE QUE C'EST ?

Ce n'est pas un lieu regroupant tout plein de livres, non bien au contraire. En programmation, une bibliothèque ou un module est un fichier qui contient plusieurs fonctions déjà définies. Nous n'avons donc pas à les créer. Pour les utiliser il suffit juste d'aller les chercher, et de les appeler, comme on l'a vu dans la section « Définition de fonctions », dans le fichier dans lequel on en a besoin.

# COMMENT FAIT-ON ?

Bonne question !

Si vous essayez d'appeler une fonction sans avoir précisé « où » elle se trouve, cela ne fonctionnera pas.

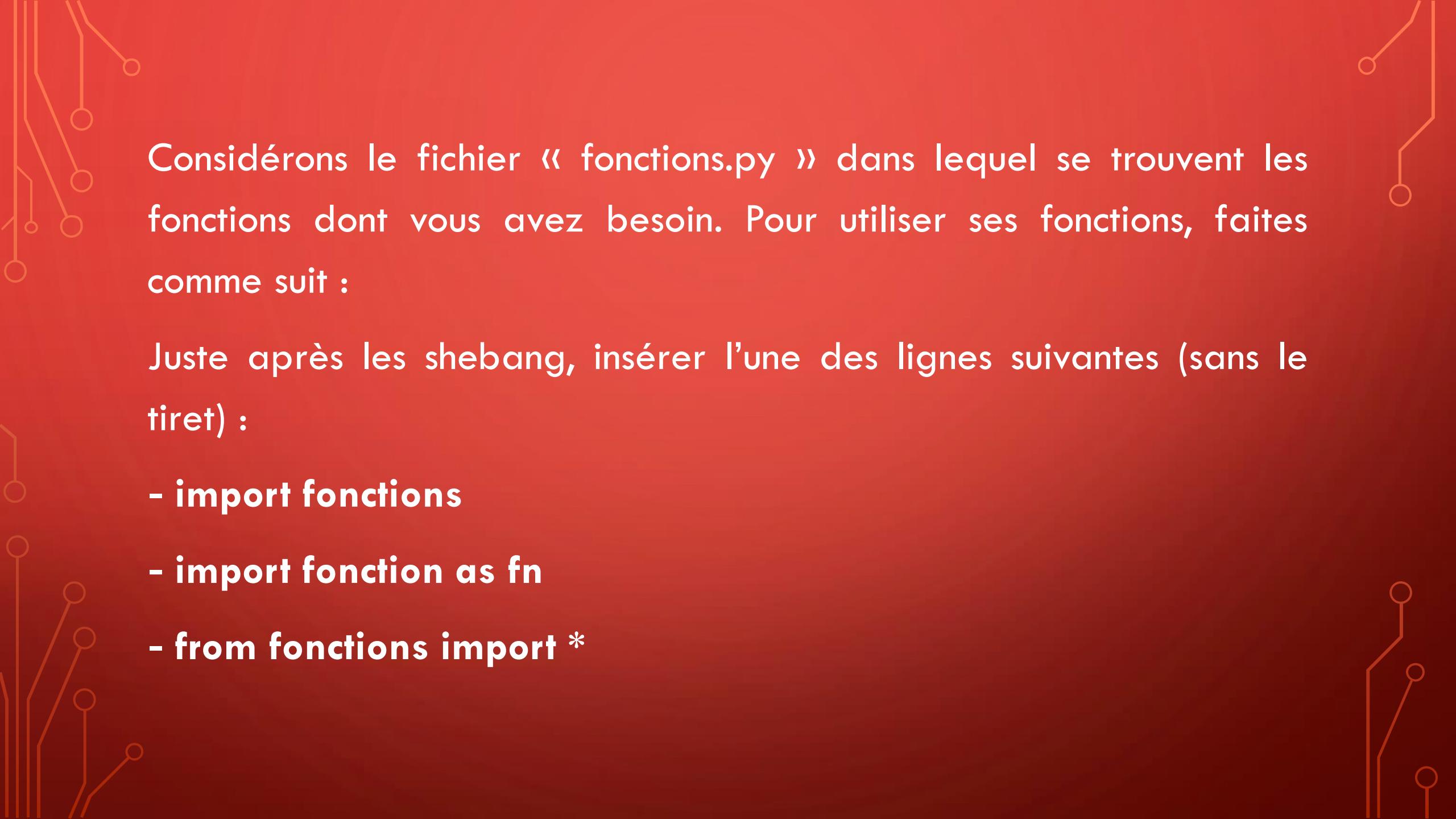
Exception faite pour les fonctions de base telles que **print( )**,  
**range( )**, **max( )**, etc.

# IMPORTER UN MODULE

Pour utiliser une fonction issuz d'un autre fichier, il suffit d'importer le fichier dans lequel sont situées les fonctions. Ce/ces fichier(s) portent également l'extension « `.py` ».

Lorsque vous créerez votre propre module, enregistrez-le en « `.py` ».

Il y a plusieurs manières d'importer un module. On en distingue 3.



Considérons le fichier « fonctions.py » dans lequel se trouvent les fonctions dont vous avez besoin. Pour utiliser ses fonctions, faites comme suit :

Juste après les shebang, insérer l'une des lignes suivantes (sans le tiret) :

- **import fonctions**
- **import fonction as fn**
- **from fonctions import \***

# IMPORT FONCTIONS

C'est le classique (attention : en minuscules). On « importe » le fichier dans lequel sont les fonctions qu'on veut utiliser dans le fichier dans lequel on code. Si dans le fichier « **fonctions.py** » est définie la fonction **CARRE()**, alors pour pouvoir l'utiliser il faudra écrire :

**fonctions.CARRE()**

En effet, cette manière d'importer le module nécessite la présence du nom du module devant la fonction appartenant au module que vous voulez utiliser. Et ce pour toutes les fonctions du fichier.

# IMPORT FONCTIONS AS FN

Même principe que précédemment sauf qu'au lieu d'écrire le nom du module devant chaque fonction, vous écrirez :

**fn.CARRE()**

En effet, le « **as** » permet de donner un nom (simplifié) à la place du nom de votre module. C'est très utile si votre module a un nom super long. Vous pouvez choisir n'importe quel nom, du moment que vous n'utilisez pas les caractères « interdits » (accents, ponctuation, ...), en un mot, sans espace.

# FROM FONCTIONS IMPORT UNE\_FONCTION

En important votre module de cette manière, vous n'avez pas besoin de préciser le nom du module lorsque vous appelez la fonction « une\_fonction() ».

## FROM FONCTIONS IMPORT \*

En utilisant cette manière d'importer votre module, il n'y a plus besoin de préciser le nom du module devant chaque fonction.

En effet le « \* » signifie que vous importez toutes les fonctions présentes dans la bibliothèque. Vous appellerez votre fonction ainsi :

**CARRE()**

Attention tout de même. Si vous importez plusieurs modules de cette manière, il se peut que vos modules comportent des fonctions dont le nom est le même. Cela peut donc entraîner des erreurs. Lorsque vous importez plusieurs modules à la fois, préférez les deux premières manières d'importer vos modules.

# EXEMPLE

Fichier fonctions.py

```
1 #!/usr/bin/python
2 #-*- coding: utf-8 -*-  
3  
4  
5  
6 def CARRE(nombre):  
7     return nombre*nombre  
8  
9  
10 def CUBE(nombre):  
11     return nombre*nombre*nombre  
12
```

```
import fonctions  
  
print fonctions.CARRE(2)  
import fonctions as fn  
  
print fn.CARRE(2)  
from fonctions import CARRE  
  
print CARRE(2)
```

```
4
```

```
from fonctions import *  
  
print CARRE(2)  
print CUBE(2)
```

```
4  
8
```

# REMARQUES

Vous pouvez importer votre module n'importe où dans votre script, et pas forcément tout au début. Importez-le juste avant d'utiliser une fonction issue du module.

Lorsque vous importez plusieurs modules dans un script, vous pouvez le faire en une seule ligne :

```
import module1, module2, module3
```

Ne fonctionne pas lorsque vous mettez un « **as** » ou une « **\*** ». Importez-les sur des lignes différentes.

# EXEMPLE

```
print 'Bonjour'  
print 'Voici'  
print 'un exemple'  
  
import fonctions  
  
print fonctions.CARRE(2)
```

```
Bonjour  
Voici  
un exemple  
4
```

```
4 print fonctions.CARRE(2)  
5  
6 import fonctions  
  
Traceback (most recent call last):  
  File "script.py", line 4, in <module>  
    print fonctions.CARRE(2)  
NameError: name 'fonctions' is not defined
```

# CRÉER VOTRE PROPRE MODULE

Rien de plus simple !

Il suffit de créer un simple fichier « .py », dans lequel vous définissez vos fonctions (sans les appeler bien sûr).

Si des « `print` » par exemple figurent dans le fichier que vous importez, ceux-ci seront exécutés.

# EXERCICES

Recréer le fichier `fonctions.py`. Importez-le dans un autre fichier script dans lequel une variable « `nombre` » est déjà définie et contient 5.

Affichez la somme du cube et du carré de `nombre`.

Affichez la différence entre le produit du cube du `nombre` par le cube du `nombre` et la somme du carré du `nombre` et du cube du `nombre`.

# Quelques bibliothèques utiles

# MATH

- **import math**

Cette bibliothèque comporte des fonctions mathématiques :

Cosinus, sinus, puissance, racine carré, logarithme, ...

<https://docs.python.org/2/library/math.html>

# EXEMPLES

```
import math

racine_carre = math.sqrt(9)
logarithme = math.log(0.87687)
exponentielle = math.exp(1)
cosinus = math.cos(math.pi)

print racine_carre
print logarithme
print exponentielle
print cosinus
```

```
3.0
-0.131396530209
2.71828182846
-1.0
```

# RANDOM

- **import random**

Très utile pour générer des nombres aléatoires.

<https://docs.python.org/2/library/random.html>

# EXEMPLE

```
import random

liste = ['untruc', 87687, 1.9]

entier_aleatoire = random.randint(10, 20)
entre_zero_et_un = random.random()
reel_aleatoire = random.uniform(0, 30)
dans_une_liste = random.choice(liste)

print entier_aleatoire
print entre_zero_et_un
print reel_aleatoire
print dans_une_liste
```

```
16
0.808332207626
1.60701783564
87687
```

# COPY

- **import copy (as cp)**

La fonction `copy` du module `copy` permet de créer une « réplique » d'un élément. Très utile lors de la manipulation de listes par exemple.

<https://docs.python.org/2/library/copy.html>

# OBSERVONS

```
import copy  
  
liste = [1, 2, 3, 4, 5]  
liste2 = liste  
  
print liste  
print liste2  
  
liste[0] = 0  
  
print liste  
print liste2
```

```
[1, 2, 3, 4, 5]  
[1, 2, 3, 4, 5]  
[0, 2, 3, 4, 5]  
[0, 2, 3, 4, 5]
```

# EXPLICATIONS

liste2 ne fait que « pointer » sur liste. Il ne possède pas lui-même sa « propre » liste. Lorsque l'on modifiera liste, on modifie la même liste sur laquelle liste2 pointe. D'où les résultats observés.

Pour corriger cela, on utilise la fonction `copy` du module `copy`.

# ILLUSTRATION

```
import copy

liste = [1, 2, 3, 4, 5]
liste2 = copy.copy(liste)

print liste
print liste2

liste[0] = 0

print liste
print liste2
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[0, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

# NUMPY

- **import numpy (as np)**

Bibliothèque remplie de fonctions multiples et variés, telles que des fonctions mathématiques, des fonctions random, etc. Sa principale caractéristique réside dans le fait qu'elle permet de créer et manipuler les « matrices ». Ce sont des listes de listes, mais de format différent.

<https://docs.scipy.org/doc/numpy-dev/user/quickstart.html>

# EXEMPLE

```
import numpy

matrice = numpy.array([[1, 2], [3, 4]])

liste = [[5, 6], [7, 8]]

print matrice
print liste
print numpy.array(liste)

zeros = numpy.zeros((2, 3))
uns = numpy.ones((4, 2))

print zeros
print uns
```

```
[[1 2]
 [3 4]]
[[5, 6], [7, 8]]
[[5 6]
 [7 8]]
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
[[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]
```

## REMARQUE

Les matrices ne peuvent contenir que des valeurs numériques. Si vous essayez d'y mettre autre chose, vous ne pouvez pas.

```
import numpy

matrice = numpy.array([[1, 2], [3, 4]])

print matrice

matrice[0][0] = 387686

print matrice

matrice[0][0] = 'UnMot'

[[1 2]
 [3 4]]
[[387686      2]
 [      3      4]]
Traceback (most recent call last):
  File "script.py", line 16, in <module>
    matrice[0][0] = 'UnMot'
ValueError: invalid literal for long() with base 10: 'UnMot'
```

# MATPLOTLIB

- **import matplotlib**

Cette bibliothèque permet principalement de créer des graphes 2D ou 3D. Il permet de générer des courbes, des histogrammes, ...

<http://matplotlib.org/>

# MATPLOTLIB.PY PLOT

- **import matplotlib.pyplot (as plt)**

Pyplot est un sous-ensemble du module matplotlib. Il contient les principales fonctions qui permettent de tracer un graphique. Les autres modules peuvent venir compléter celui-ci (légendes, effets, couleurs, etc.).

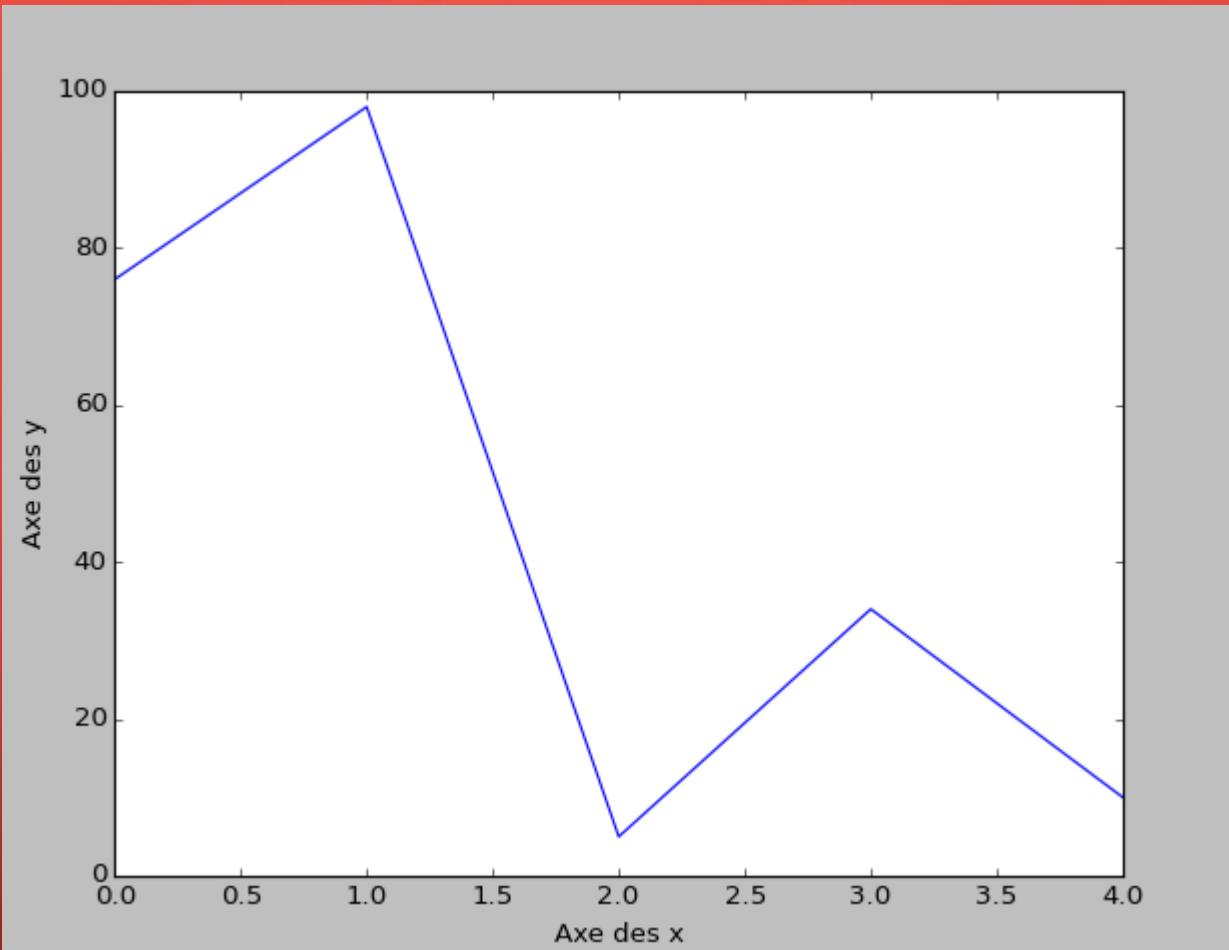
[http://matplotlib.org/api/pyplot\\_api.html](http://matplotlib.org/api/pyplot_api.html)

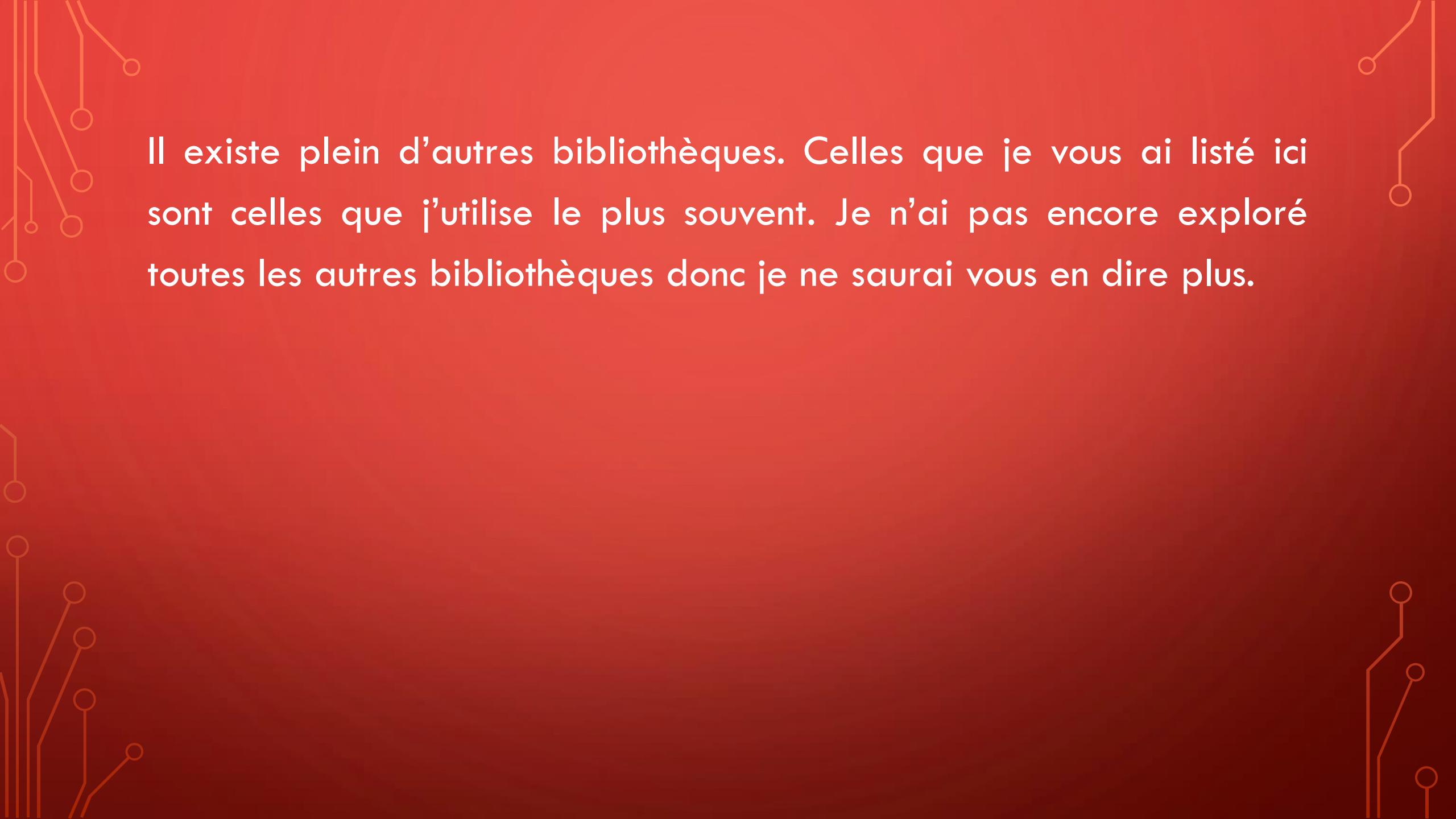
# UN GRAPHIQUE SIMPLE

```
import matplotlib.pyplot

liste_des_x = [0, 1, 2, 3, 4]
liste_des_y = [76, 98, 5, 34, 10]

matplotlib.pyplot.plot(liste_des_x, liste_des_y)
matplotlib.pyplot.xlabel('Axe des x')
matplotlib.pyplot.ylabel('Axe des y')
matplotlib.pyplot.show()
```



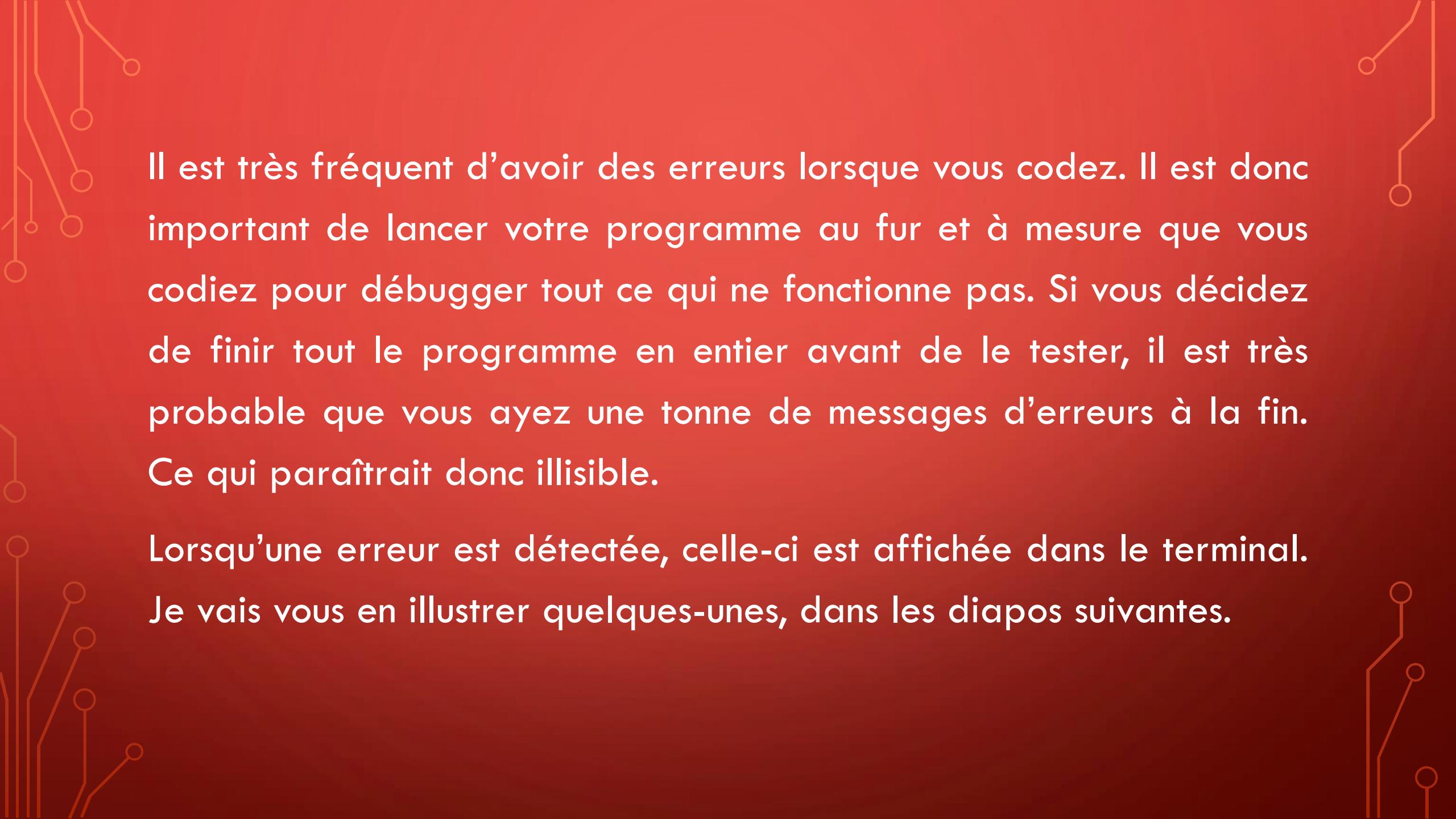


Il existe plein d'autres bibliothèques. Celles que je vous ai listé ici sont celles que j'utilise le plus souvent. Je n'ai pas encore exploré toutes les autres bibliothèques donc je ne saurai vous en dire plus.

# EXERCICES

- 1) Créer une matrice 9 lignes 9 colonnes remplie de 0.
- 2) Remplissez chaque case de la matrice par un entier aléatoire entre 0 et 100 (inclu)
- 3) Récupérer la deuxième ligne de la matrice dans une variable
- 4) Changer la valeur au milieu par un nouvel entier aléatoire
- 5) Affichez la matrice et la variable qui contient la deuxième ligne modifiée.
- 6) Créer un graphique : première ligne de la matrice en fonction de la dernière ligne de la matrice.

# Messages d'erreurs fréquents



Il est très fréquent d'avoir des erreurs lorsque vous codez. Il est donc important de lancer votre programme au fur et à mesure que vous codiez pour débugger tout ce qui ne fonctionne pas. Si vous décidez de finir tout le programme en entier avant de le tester, il est très probable que vous ayez une tonne de messages d'erreurs à la fin. Ce qui paraîtrait donc illisible.

Lorsqu'une erreur est détectée, celle-ci est affichée dans le terminal. Je vais vous en illustrer quelques-unes, dans les diapos suivantes.

Voyons tout d'abord la structure d'une erreur :

Fichier d'où provient l'erreur

Numéro de la ligne de l'erreur

Ligne d'où l'erreur provient

Type de l'erreur

```
File "script.py", line 6  
nombre = (1+1)*2)/10  
          ^  
  
SyntaxError: invalid syntax
```

La plupart des erreurs sont structurés comme illustré dans la diapositive d'avant.

- Nom du fichier où l'erreur se trouve
- Numéro de la ligne d'où l'erreur provient
- Ligne d'où l'erreur provient
- Type de l'erreur

Avant tout ceci, vous devez être capable de « tracer » l'erreur, d'identifier ce qui ne va pas, et de la corriger. Rien de plus simple.

# Quelques exemples

# ZERO DIVISION ERROR (INTEGER DIVISION OR MODULO BY ZERO)

```
1 print 12/0
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    print 12/0
ZeroDivisionError: integer division or modulo by zero
```

Une division par 0 n'est pas possible.

# NAME ERROR (NOT DEFINED)

```
1 #!/usr/bin/python
2 -*- coding: utf-8 -*-
3
4
5
6 print a

Traceback (most recent call last):
  File "script.py", line 6, in <module>
    print a
NameError: name 'a' is not defined
```

L'erreur indique ici que la variable « a » n'est pas définie. Elle n'existe donc pas. Pensez à affecter quelque chose dedans, ou mettre des guillemets ou apostrophes si vous voulez afficher la lettre « a ».

# NAME ERROR (NOT DEFINED 2)

```
1 #!/usr/bin/python
2 -*- coding: utf-8 -*-
3
4
5
6
7 math.log(67)
```

```
File "script.py", line 7, in <module>
    math.log(67)
NameError: name 'math' is not defined
```

Vous essayez d'utiliser la fonction log (logarithme) du module math. Toutefois, vous n'avez pas importé le module correspondant. De ce fait, le programme ne connaît pas la fonction log, ni d'où il doit la tirer. Rajouter un import math juste avant la ligne 7.

# INDENTATION ERROR (EXPECTED AN INDENTED BLOCK)

```
6 nombre = 9
7
8 if nombre == 9:
9 print "oui"
```

```
File "script.py", line 9
    print "oui"
          ^
```

```
IndentationError: expected an indented block
```

L'erreur indiquée ici indique qu'un bloc indenté est attendu à la ligne 9. Le « if » n'ayant pas d'instructions à exécuter, il faut lui passer des instructions en indentant le bloc situé dans la ligne juste après.

# INDENTATION ERROR (UNEXPECTED INDENT)

```
6 variable = 378687689
7
8     print variable
```

```
File "script.py", line 8
    print variable
^
```

```
IndentationError: unexpected indent
```

Contrairement à l'erreur précédente, il y a ici une indentation en trop. Voyez que le « `print` » à la ligne 8 est indenté alors qu'il ne le devrait pas.

# SYNTAX ERROR (INVALID SYNTAX)

```
6 nombre = (1+1)*2)/10
```

```
File "script.py", line 6
    nombre = (1+1)*2)/10
               ^
SyntaxError: invalid syntax
```

Erreur très fréquente, la syntaxe invalide. En effet cette erreur indique que vous n'avez pas bien écrit ce qu'il faut. Dans l'exemple ci-dessus, remarquez qu'il manque une parenthèse ouvrante « ( » tout au début de la ligne.

# SYNTAX ERROR (EOL WHILE SCANNING STRING LITERAL)

```
7 mot = "bonjour"  
  
File "script.py", line 7  
    mot = "bonjour"  
          ^  
SyntaxError: EOL while scanning string literal
```

Moins fréquent que les erreurs précédentes, mais tout de même observable. Ici, l'erreur est que vous entourez une chaîne de caractères avec un guillemet et une apostrophe. Vous devez soit mettre deux apostrophes soit deux guillemets mais pas les deux.

# SYNTAX ERROR (NON-ASCII CHARACTER)

```
1 #!/usr/bin/python
2
3 print "dkfkdhfèss"
```

```
File "script.py", line 3
SyntaxError: Non-ASCII character '\xc3' in file script.py on line 3, but no encoding declared; see http://python.org/dev/peps/pep-0263/ for details
```

Les caractères ASCII sont des caractères particuliers. Je ne vais pas m'approfondir dessus. Dans l'exemple ci-dessus, le caractère non désirable est le « è ». Pour passer outre cette erreur, rajoutez le shebang manquant.

```
2 #-*- coding: utf-8 -*-
```

```
1 #!/usr/bin/python
2 #-*- coding: utf-8 -*-
3
4 print "dkfkdhfèss"
```

```
dkfkdhfèss
```

# TYPE ERROR (CANNOT CONCATENATE)

```
7 mot = 'mot'  
8 nombre = 8  
9  
10 tout = mot+nombre
```

```
Traceback (most recent call last):  
  File "script.py", line 10, in <module>  
    tout = mot+nombre  
TypeError: cannot concatenate 'str' and 'int' objects
```

Vous ne pouvez pas concaténer (additionner) deux variables de type différent. C'est ce qu'indique l'erreur ici.

# TYPE ERROR (FUNCTION() TAKES EXACTLY X ARGUMENT(S))

```
7 def carre(nombre):  
8     return nombre*nombre  
9  
10  
11 print carre(1, 2)
```

```
Traceback (most recent call last):  
  File "script.py", line 11, in <module>  
    print carre(1, 2)  
TypeError: carre() takes exactly 1 argument (2 given)
```

La fonction `carre()` ne prend qu'un seul argument. Or vous lui en passer 2 lorsque vous appelez la fonction.

# INDEX ERROR (LIST INDEX OUT OF RANGE)

```
7 liste = [1, 2, 3, 4, 5]
8
9 print liste[5]
```

```
Traceback (most recent call last):
  File "script.py", line 9, in <module>
    print liste[5]
IndexError: list index out of range
```

Ici, vous essayez d'afficher ce qui se trouve dans la case d'indice 5 de la liste. Cependant, voyez que la liste n'a que 5 cases, et par conséquent, ne va que jusqu'à l'indice 4 (commence à 0 pas à 1).

# IO ERROR (NO SUCH FILE OR DIRECTORY)

```
7 fichier = open('truc.txt', 'r')  
  
romain@Hp-debian:~/Documents$ python script.py  
Traceback (most recent call last):  
  File "script.py", line 7, in <module>  
    fichier = open('truc.txt', 'r')  
IOError: [Errno 2] No such file or directory: 'truc.txt'
```

Le fichier que vous tentez d'ouvrir n'existe pas, ou ne se trouve pas dans le répertoire dans lequel vous êtes.

# VALUE ERROR (X NOT IN LIST)

```
6 liste = [1, 2, 3]
7
8 liste.remove(4)

Traceback (most recent call last):
  File "script.py", line 8, in <module>
    liste.remove(4)
ValueError: list.remove(x): x not in list
```

Cette erreur indique que l'élément que vous essayez de supprimer n'est pas dans la liste. Ici, 4 n'est pas dans la liste.

# ATTRIBUTE ERROR (OBJECT HAS NO ATTRIBUTE)

```
7 UnTuple = (1, 2, 3)
8
9 UnTuple.remove(1)
```

```
Traceback (most recent call last):
  File "script.py", line 9, in <module>
    UnTuple.remove(1)
AttributeError: 'tuple' object has no attribute 'remove'
```

Si vous vous rappelez de ce qu'est un tuple, c'est une liste non modifiable. Ce qu'on veut signifier par non modifiable, c'est qu'il n'existe pas de fonctions qui permettent de manipuler les tuples.

Dans cet exemple, on veut supprimer la valeur 1 du tuple avec la fonction `remove( )`. Toutefois cette fonction ne fonctionne qu'avec des listes et non de tuples.

# TRAÇAGE DE L'ERREUR

```
6 def CARRE(nombre):
7     truc
8     return nombre*nombre
9
10
11 def CUBE(nombre):
12     return CARRE(nombre)*nombre
13
14
15 def main(nombre):
16     return CUBE(nombre)
17
18
19 main(2)
```

```
Traceback (most recent call last):
  File "script.py", line 19, in <module>
    main(2)
  File "script.py", line 16, in main
    return CUBE(nombre)
  File "script.py", line 12, in CUBE
    return CARRE(nombre)*nombre
  File "script.py", line 7, in CARRE
    truc
NameError: global name 'truc' is not defined
```

Ce dernier cas est un bon exemple de « traçage » de l'erreur. Il suffit de lire ce qui est écrit dans le terminal de haut en bas et de « chercher » l'endroit où est l'erreur dans le script en lisant de bas en haut.

L'erreur est située dans l'appel de fonction **main()** ligne 19, dans la fonction **main()** ligne 16, dans la fonction **CUBE()** ligne 12, et enfin dans la fonction **CARRE()** ligne 7.

« truc » n'est pas défini. D'où l'erreur « not defined ».

Vous aurez souvent à faire à ce cas.

# EXERCICE

```
1#!/usr/bin/python
2#-*- coding: utf-8 -*-
3
4import math
5
6
7variableé = 100
8        variable2 = 500
9        variable3 = 13.8
10
11print math.log(variable1, 10)
12
13for i in range(20)
14    variable1 = variable+10
15
16    if variable1 == 200:
17        print "Supérieur à 200"
18
19
20if variable2 > 300:
21    print 'Oh bien !!!'
22    variable3 = variable2+variable3
23
24else:
25    print "Peut mieux faire"
26
27valeur_aleatoire = random.random()
28print valeur_aleatoire
29
30print Vous avez réussi !!!!"
```

Ce programme comporte plusieurs erreurs. Trouvez-les, essayez de deviner quelle est le type de chaque erreur et corriger les.

# MANIPULER LES ERREURS

Lorsque que vous lancez un programme et qu'il n'y a pas d'erreur(s), celui s'exécute jusqu'au bout, ou jusqu'à ce qu'il rencontre un **return**. Lorsqu'une erreur survient, le programme s'arrête, sans exécuter les instructions qui suivent. Il est possible de contourner cela et permettre au programme de lancer les instructions qui suivent. Pour cela, nous allons utiliser les mots clés **try** et **except** qui fonctionnent comme **if** et **else** (une erreur est aussi appelé « exception »).

Littéralement, le **if** et **else** donnent : « si ..., fait cela. Sinon fait cela. »  
Et le **try** et **except** : « essaye cela. Sinon fait cela. »

Voyez ? Pratiquement la même chose.

# TRY ... EXCEPT ...

Syntaxe :

```
try :  
    instructions  
  
except :  
    instructions
```

Je ne vais pas tout re-détailler, c'est comme les conditions (diapos 144 et suivantes).

# EXEMPLE

```
n = 12  
try :  
    a = n/0  
    print a  
except :  
    print "il y a une erreur"  
  
il y a une erreur
```

Dans cet exemple, nous avons essayé de faire une division par 0, ce qui est impossible. Sans le bloc **try/except**, le programme se serait arrêté mais ici, l'instruction du bloc **except** est exécutée. Dans le terminal, les éléments qui figurent à la diapo 243 ne sont pas affichés, notamment le type de l'erreur. Saurez-vous deviner que est le type de l'erreur ici ?

Il s'agit du **ZeroDivisionError** (attention aux majuscules), diapo 246.

Le mot clé **except** utilisé seul s'exécute pour toute erreur rencontrée, quelque soit son type.

Connaissant le type de l'erreur, il est possible de la traiter au cas par cas, en le précisant à la suite de **except**.

```
n = 12  
  
try :  
    a = n/0  
    print a ] ←  
  
except ZeroDivisionError :  
    print "Bip ! Division par zero"  
  
except NameError :  
    print "Bip ! Variable inexistante"  
  
except :  
    print "il y a une erreur"
```

Bip ! Division par zero

```
n = 12  
  
try :  
    print a ] ←  
    a = n/0  
  
except ZeroDivisionError :  
    print "Bip ! Division par zero"  
  
except NameError :  
    print "Bip ! Variable inexistante"  
  
except :  
    print "il y a une erreur"
```

Bip ! Variable inexistante

Dans le deuxième cas, j'ai essayé d'afficher la variable « a » avant de la créer, d'où le **NameError**.

Si n'y a ni le **ZeroDivisionError**, ni le **NameError**, les instructions du bloc **except** tout seul sont exécutés (si ce bloc existe) ...

```
n = 12
l = [0, 1]

try :
    l[2] = 2
    print a
    a = n/0

except ZeroDivisionError :
    print "Bip ! Division par zero"

except NameError :
    print "Bip ! Variable inexistante"

except :
    print "il y a une erreur"

il y a une erreur
```

... sinon les éléments de la diapo 243 sont affichés.

Le type de l'erreur ici est **IndexError**.



```
1 n = 12
2 l = [0, 1]
3
4 try :
5     l[2] = 2
6     print a
7     a = n/0
8
9 except ZeroDivisionError :
10    print "Bip ! Division par zero"
11
12 except NameError :
13    print "Bip ! Variable inexistante"

Traceback (most recent call last):
  File "test.py", line 5, in <module>
    l[2] = 2
IndexError: list assignment index out of range
```

# LEVER ET PERSONNALISER VOTRE ERREUR

Il est possible de lever une erreur manuellement, et de la personnaliser.

Comment ? Avec le mot clé **raise**. Littéralement « lever ». On parle de « lever une erreur ».

Il fonctionne comme **return**. C'est-à-dire qu'après avoir levé l'erreur, le programme s'arrête.

Syntaxe :

```
raise type_erreur(« description »)
```

```
1 print 1
2
3 raise ZeroDivisionError("On ne divise pas par 0 !")
4
5 print 2
```

```
1
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    raise ZeroDivisionError("On ne divise pas par 0 !")
ZeroDivisionError: On ne divise pas par 0 !
```

Notez que le « `print 2` » situé après l'instruction `raise` ne s'est pas exécuté.

En général, on lève une exception dans un bloc **try/except**, et non pas tout seul comme fait précédemment.

```
1 a = 1
2 b = 2
3
4 try :
5     c = a+b
6     if c < 4 :
7         raise ValueError("le resultat est inferieur a 4 !")
8
9 except ValueError : # ou juste except :
10    print "a ou b est trop petit"
```

```
a ou b est trop petit
```

# COMBINAISON CONDITION ET ERREUR

Il est possible de tester une condition et de lever une erreur si la condition n'est pas vérifiée plus simplement qu'en écrivant `if ... raise ...`

Pour cela, il faut utiliser `assert`.

Syntaxe :

`assert condition`

C'est une façon de vérifier une condition et de lever une erreur avant d'aller plus loin dans le programme. Si la condition est vérifiée, rien ne se passe, les instructions qui suivent s'exécutent. Si elle n'est pas vérifiée, l'erreur levée est de type `AssertionError`, et comme vous l'avez peut-être deviné, le programme s'arrête. Vous pouvez l'inclure dans un `try/except` (ou pas).

```
1 a = 1
2 b = 2
3
4 assert a == b
5
6 print "ici"
```

```
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    assert a == b
AssertionError
```

```
1 a = 1
2 b = 2
3 c = 4
4
5 try :
6     d = a+b
7     assert c == d
8     print "OK"
9
10 except AssertionError : # ou juste except :
11     print "ce n'est pas bon"
```

ce n'est pas bon

# PASS

Lorsqu'une erreur est levée, il est possible de l'ignorer et de continuer à exécuter les instructions qui suivent. Il suffit pour cela de mettre le mot clé **pass** dans le bloc **except** tout simplement.

```
a = 1
b = 2
c = 4

try :
    d = a+b
    assert c == d
    print "OK"

except AssertionError : # ou juste except :
    pass

print "fin du programme"
fin du programme
```

# FINALLY

Non ce n'est pas la fin des diapos. C'est un autre mot clé (et oui il y en a beaucoup). Il s'utilise après un bloc **try/except**. On l'utilise généralement pour contourner un éventuel **return** (fin du programme) dans le bloc **except**. Le bloc **finally** est ainsi exécuté bien qu'il y ai un **return** avant.

```
def fonz(a,b):
    try :
        assert a == b
        print "OK"

    except :
        print "pas OK"
        return

    print "fin du programme"

fonz(1,2)
pas OK
```

```
def fonz(a,b):
    try :
        assert a == b
        print "OK"

    except :
        print "pas OK"
        return

    finally :
        print "fin du programme"

fonz(1,2)
pas OK
fin du programme
```

# EXERCICES

Il n'y en a pas ! Vous arrivez à la fin des diapos, vous avez le droit à un repos mérité.

En réalité je n'ai pas trouvé et/ou n'ai pas d'idée d'exercice pertinent pour mettre en valeur les erreurs. Peut être dans une prochaine mise à jour !

Essayez de créer vous-même vos exercices afin de bien assimilez ces notions.



## POUR ALLER PLUS LOIN

Si vous souhaitez d'avantage maîtriser le langage de programmation python, je vous invite à vous tourner autour de l'utilisation de bibliothèques. Elles contiennent une multitudes de fonctions qui vous permettront au mieux d'optimiser votre travail.

Je vous conseille ainsi le module **matplotlib** qui est très bien pour la représentation de données, ainsi que **numpy** pour l'utilisation de matrices.

Vous connaissez déjà tout pleins de bibliothèques ? Je vous invite donc à essayer la **programmation orientée objet (POO)**. Vous ne le saviez pas, mais depuis le début vous en utilisiez. Où cela me dites-vous ? La réponse dans de prochaines diapositives !