

# LES VARIABLES

## Rappel définition d'une variable

Le travail d'un ordinateur, c'est de manipuler des données qui sont stockées dans sa mémoire. Or, pour cela, il faut un minimum d'organisation. C'est la raison pour laquelle les ordinateurs utilisent des variables, lesquelles correspondent à des adresses mémoire bien précis où sont stockées les données.

Une variable, c'est donc avant tout une adresse mémoire mais c'est également un endroit que l'on réserve dans la mémoire de l'ordinateur. Lorsqu'on déclare une variable, on crée un petit tiroir sur lequel on colle une étiquette (qui correspond au nom de la variable) et dans lequel on stocke une valeur.

## Les différents types de variables

Les variables peuvent stocker des données de différents types :

- des nombres entiers qui correspondent au type **int** (integer en anglais)
- des nombres décimaux qui correspondent au type **float**.
- des strings (c'est-à dire des chaînes de caractères en français) qui correspondent au type **str**.
- des booléens qui correspondent au type **bool**.
- des listes qui correspondent au type **list**.
- des tuples qui correspondent au type **tuple**.
- des dictionnaires qui correspondent au type **dict**.

Dans un premier temps, découvrons ensemble les trois premiers types, c'est-à dire les nombres entiers, les nombres décimaux et les chaînes de caractères.

## Déclarer une variable et lui affecter une valeur

Pour déclarer une variable, rien de plus simple, il suffit de lui donner un nom et de lui affecter une valeur. Par exemple, je déclare la variable **n** et je lui affecte le nombre entier **11** :

```
[code language= « python » light= »true »]  
n = 11  
[/code]
```

Dans les entrailles de votre ordinateur, un tiroir s'est ouvert et un lutin numérique y a glissé le nombre entier **11**. Maintenant, à chaque fois que l'on appelle la variable **n**, le nombre entier **11** apparaît. Notez bien que le signe **=** n'est pas un symbole d'égalité arithmétique mais un signe d'affectation, c'est-à dire que si l'on a envie d'affecter une autre valeur à la variable **n**, nous pouvons le faire comme bon nous semble.

n = 7 nouvelle affectation

Le langage Python est sensible à la casse, c'est-à dire que **n** est différent de **N**. Une variable que vous avez baptisée **nombre\_7**, est différente de la variable **Nombre\_7**. Dans vos noms de variables, vous pouvez utiliser toutes les lettres combinées à tous les chiffres mais vous ne pouvez pas utiliser d'accents ni de caractères spéciaux! Vous ne pouvez utiliser que le tiret « Under scoré » : **\_** En outre, vous ne pouvez pas commencer par un chiffre. Par exemple, **7nombre** va vous retourner une exception, c'est à dire une erreur, et le programme ne s'exécutera pas.

# LES CONDITIONS

Dans un programme écrit en langage Python, les instructions s'exécutent les unes après les autres. Par exemple, dans ce code, on commence par déclarer une variable **a** à laquelle on affecte le nombre entier 5. On fait un print qui m'affiche 5 puis à la ligne suivante, on modifie la variable **a** avant de refaire un print. Enfin, on déclare une variable **b** à laquelle on affecte le nombre entier 10 et on fait un print qui nous affiche l'addition de **a + b**. Chaque instruction s'exécute l'une après l'autre.

```
[code language= « python »]
#!/usr/bin/python3
# -*- coding: utf8 -*-
a = 5
print(a)
a = a + 6
print(a)
b = 10
print (a + b)
[/code]
```

Seulement parfois, il est nécessaire d'orienter le programme par rapport aux données qu'un utilisateur va renseigner. Et c'est là que les conditions entrent en jeu. Prenons l'exemple de la fonction **input ()**. Cette fonction met le programme en attente jusqu'à ce que l'utilisateur ait renseigné le champ d'entrée. C'est le cas dans le programme ci-dessous on demande à l'utilisateur de renseigner son âge. Selon sa réponse, l'une des quatre conditions va s'afficher.

```
[code language= »python » wraplines= »false » collapse= »false »]
#!/usr/bin/python3
# -*- coding: utf8 -*-
age = input("Veuillez entrer votre âge : ") # Fonction input()
# age contient une chaîne de caractères
# que je transforme en nombre entier grâce à la fonction int()
age = int(age)
if age > 0 and age < 20 : # 1ère condition
print("Tu es jeune!") # Indentation obligatoire en Python
elif age > 20 and age < 60 : # 2ème condition
print("Vous êtes un adulte.") # Bloc d'instructions indenté
elif age > 60 : # 3ème condition
print ("Vous avez acquis une certaine expérience de la vie.")
else : # Dernière condition
print ("Vous n'êtes pas encore né puisque vous avez entré un âge négatif!")
```

Pour écrire une condition, il faut utiliser le mot-clé **if** (« si » en français) suivi d'un bloc d'instructions indenté.

C'est obligatoire en Python! Qu'est-ce donc que **l'indentation**? C'est le décalage de quatre espaces qui précède le bloc d'instructions. On déconseille d'utiliser les tabulations pour effectuer le décalage. Utilisez plutôt quatre espaces. Vous pouvez configurer votre éditeur de texte de telle sorte que la touche **->|** crée un décalage de quatre espaces au lieu d'une tabulation. Surtout, ne mélangez pas les tabulations et les décalages de quatre espaces sous peine de voir Python vous retourner une exception (error)!

Si le premier **if** ne s'exécute pas, c'est la deuxième condition ou la troisième condition qui entre en action grâce au mot-clé **elif**. **elif** est la contraction de **else if**. Cela signifie **sinon si**.

Enfin, si aucune condition ne s'applique à la valeur entrée par l'utilisateur, c'est la dernière condition qui entre en action : **else** signifie **sinon**.

```
• • • #[codelanguage= »python » wraplines= »false » collapse= »false »]
• • • #!/usr/bin/python3
• • • # -*- coding: utf8 -*-
• 19 age = input("Veuillez entrer votre âge : ") # Fonction input()
• • # age contient une chaîne de caractères
• • # que je transforme en nombre entier grâce à la fonction int()
• • age = int(age)
• • if age > 0 and age < 20 : # 1ère condition
• •     print("Tu es jeune!") # Indentation obligatoire en Python
• • elif age > 20 and age < 60 : # 2ème condition
• •     print("Vous êtes un adulte.") # Bloc d'instructions indenté
• • elif age > 60 : # 3ème condition
• •     print("Vous avez acquis une certaine expérience de la vie.")
• 29 else : # Dernière condition
• 30     print("Vous n'êtes pas encore né puisque vous avez entré un âge négatif!")
```

**POUR RESUMER :**

```
si + condition n° 1 : # Ne pas oublier les deux petits points!
    "Bloc d'instructions indenté"
sinon si + condition n° 2 :
    "Bloc d'instructions indenté"
sinon si + condition n° 3 :
    "Bloc d'instructions indenté"
sinon :
    "Bloc d'instructions indenté"
```

## Les expressions conditionnelles

Il existe une autre façon plus compacte d'écrire une condition. Dans certains cas, il est possible d'utiliser une expression conditionnelle qui se résume à une seule ligne d'instructions. Voici un exemple avec ce code.

```

1#!/usr/bin/python3
2# -*- coding: utf8 -*-
3
4# Je déclare deux variables auxquelles j'affecte respectivement les valeurs 5 et 10 :
5var_1, var_2 = 5, 10
6# Je déclare une troisième variable dont la valeur est tributaire
7# d'une expression conditionnelle
8var_3 = var_1 * var_2 if var_1 < var_2 else var_1 + var_2
9print(var_3)
10
11# Cette expression conditionnelle est très compacte. Elle tient sur une ligne.
12# Voici exactement à quoi elle correspond avec une condition multilignes :
13if var_1 < var_2 : # Si var_1 est inférieur à var_2 :
14    var_3 = var_1 * var_2 # var_3 est égal à la multiplication de var_1 par var_2.
15else : # sinon :
16    var_3 = var_1 + var_2 # var_3 est égal à la somme de var_1 et var_2
17print(var_3)
18
19

```

On l'exécute dans un terminal interactif IDLE-3. Que l'on utilise une expression conditionnelle ou bien une instruction conditionnelle, on constate que le résultat retourné est identique : 50. Mais l'expression conditionnelle tient sur une seule ligne.

## La boucle while

Prenons un exemple concret : une punition qu'un écolier doit recopier deux cents fois. Il y a quarante ans, le pauvre cancre pris par la patrouille devait patiemment recopier la phrase qui lui permettait d'expier sa faute : « Je ne collerai plus mon vieux chewing-gum sous mon pupitre ».

Aujourd'hui, ce genre de punition est une formalité pour celui qui maîtrise les boucles et en Python, pour en créer une, on utilise le mot-clé **while**. Cela signifie **Tant que...** C'est à dire **Tant que la condition est vraie, exécute cette instruction**.

[code language= »python »]

```

#!/usr/bin/python3
# -*- coding: utf8 -*-

```

```

# Phrase à recopier:
chaine = "Je ne copierai plus sur mon voisin.\n"

```

```

# Variables d'incréméntation.
# À chaque tour de boucle, sa valeur augmente de 1
i = 1

```

```

# BOUCLE:
while i <= 10: #Tant que i est inférieur ou égal à 10:
    print(i, chaine) # Imprimer i et chaine
    i += 1 # Incréméntation de la variable (i = i + 1)

```

```

# Lorsque i vaut 10, la boucle s'interrompt.
[/code]

```

Je l'exécute dans un terminal interactif IDLE-3 :

1 Je ne copierai plus sur mon voisin

2 Je ne copierai plus sur mon voisin

3 Je ne copierai plus sur mon voisin

4 Je ne copierai plus sur mon voisin

5 Je ne copierai plus sur mon voisin

6 Je ne copierai plus sur mon voisin

7 Je ne copierai plus sur mon voisin

8 Je ne copierai plus sur mon voisin

9 Je ne copierai plus sur mon voisin

10 Je ne copierai plus sur mon voisin

Il faut prévoir un moyen d'interrompre la boucle après un nombre défini d'[itérations](#) sous peine de créer une boucle infinie qui finirait par épuiser les ressources mémoire de votre ordinateur. Pour cela, on déclare une variable souvent nommée **i** (par convention) que l'on incrémente à chaque tour de boucle, c'est-à-dire à laquelle on ajoute la valeur 1 à chaque tour de boucle (**i += 1**) . Quand la valeur contenue dans la variable **i** est égale à la condition définie en début de boucle (dans notre exemple, c'est la ligne 9 : **tant que i est inférieur ou égal à 10**), alors la boucle s'interrompt.

Voici un exemple de boucle infinie. La condition est toujours vraie, la boucle ne s'interrompt jamais. Le seul moyen de l'interrompre avant qu'elle ne dévore toute la mémoire de votre ordinateur, tel un trou noir digital, est de presser les touches **ctrl + C**.

```
[code language= »python »]
```

```
#!/usr/bin/python3
```

```
# -*- coding: utf8 -*-
```

```
line_number = 0
```

```
# Tant que 3 est inférieur à 4 (C'est toujours vrai!)
```

```
while 3 < 4 :
```

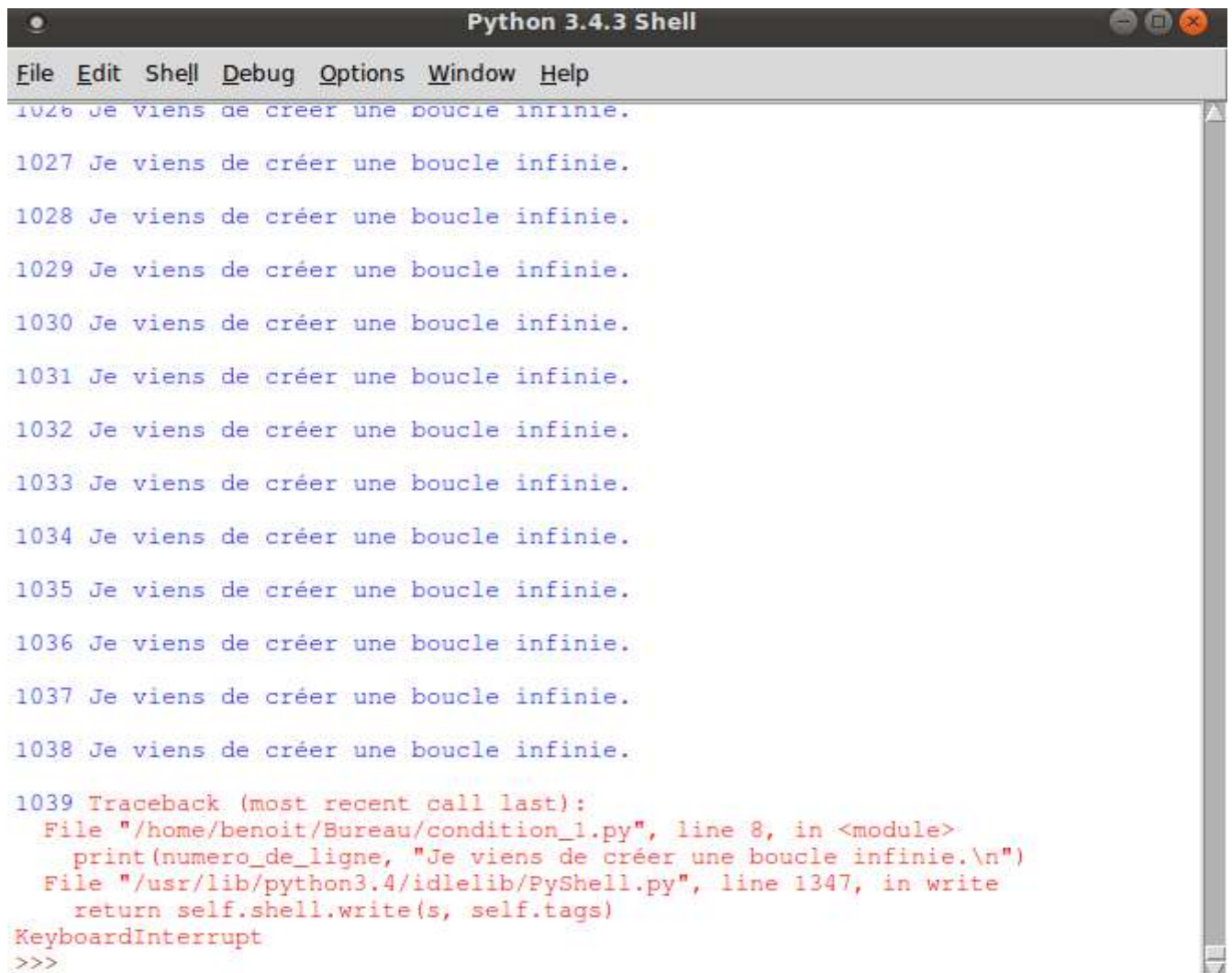
```
    line_number += 1
```

```
    print (line_number, "Je viens de créer une boucle infinie.\n")
```

```
# Rien n'est prévu pour sortir de la boucle.
```

```
[/code]
```

J'interromps la boucle après 1038 itérations :



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
1026 Je viens de créer une boucle infinie.
1027 Je viens de créer une boucle infinie.
1028 Je viens de créer une boucle infinie.
1029 Je viens de créer une boucle infinie.
1030 Je viens de créer une boucle infinie.
1031 Je viens de créer une boucle infinie.
1032 Je viens de créer une boucle infinie.
1033 Je viens de créer une boucle infinie.
1034 Je viens de créer une boucle infinie.
1035 Je viens de créer une boucle infinie.
1036 Je viens de créer une boucle infinie.
1037 Je viens de créer une boucle infinie.
1038 Je viens de créer une boucle infinie.
1039 Traceback (most recent call last):
  File "/home/benoit/Bureau/condition_1.py", line 8, in <module>
    print(numero_de_ligne, "Je viens de créer une boucle infinie.\n")
  File "/usr/lib/python3.4/idlelib/PyShell.py", line 1347, in write
    return self.shell.write(s, self.tags)
KeyboardInterrupt
>>>
```

## La boucle for

Il existe une autre manière de créer une boucle, en utilisant les mots-clés **for... in range**. Comme je l'écris en commentaire, il est inutile d'incrémenter la variable **i**, la boucle **for** s'en charge.

[code language= »python »]

```
#!/usr/bin/python3
# -*- coding: utf8 -*-
```

```
for i in range (1, 51):
    print(i, "Je dois recopier 50 fois cette phrase.")
```

```
# Il est inutile d'incrémenter la variable i.
# La boucle for s'en charge.
```

[/code]

## Parcourir une liste

Avec **for**, il est possible de parcourir une liste pour, par exemple, en extraire les éléments.

Avec **for... in**:

```
[code language= »python »]
#!/usr/bin/python3
# -*- coding: utf8 -*-
```

```
liste_fruits = ["pommes", "poires", "oranges"]
```

```
for fruit in liste_fruits:
    print (fruit)
[/code]
```

**pommes**

**poires**

**oranges**

Avec **for... in enumerate**

```
[code language= »python »]
#!/usr/bin/python3
# -*- coding: utf8 -*-
```

```
liste_fruits = ["pommes", "poires", "oranges"]
```

```
for i, fruit in enumerate (liste_fruits):
    print(i + 1, fruit)
[/code]
```

Résultat identique !



# Un peu d'histoire avec George Boole

Figurez-vous que celui qui a inventé l'**algèbre booléenne** s'appelle **George Boole**. Ce sujet anglais est né en 1815 et vous ne serez pas étonné d'apprendre qu'il est mort. C'est une pneumonie qui l'a emporté à l'âge de 49 ans, après que son épouse l'eût aspergé d'eau pour soigner le mal par le mal! (source: Wikipédia).

Il est donc le père de l'**algèbre de Boole**.

## Définition

En Python, un booléen est un type de variables qui possède deux états :

- L'état « vrai » : **True** en anglais
- Ou bien l'état « faux » : **False** en anglais

**False == 0** et **True == 1**. Comment le prouver? C'est très simple. Nous allons créer deux variables (**faux** et **vrai**) et leur affecter respectivement les valeurs **False** et **True**. Ensuite, nous allons créer une liste de deux éléments indicés **0** et **1**. Nous allons nous servir des deux variables comme indice pour retourner chacune des deux valeurs dans un **print ()**. C'est parti:

```
[code language= « python »]
faux = False
vrai = True
liste = ["Corée du Nord", "États-Unis"]
print ("1er pays:", liste[faux])
print("2ème pays:", liste[vrai])
[/code]
```

On rappelle que l'indice d'une liste ne peut être qu'un nombre entier. Le fait que j'aie réussi à retourner les deux éléments de la liste en utilisant comme indice les variables **faux** et **vrai** prouvent que ces dernières stockent des valeurs qui sont respectivement égales aux nombres entiers 0 et 1.

## Amusons-nous avec un exemple un peu tordu

J'ai dit que **faux** était égal à **0**, c'est-à-dire **faux == 0**. Attention! J'ai bien écrit **faux == 0** et non pas **faux = 0**. En effet, La variable **faux** ne stocke pas le nombre entier **0** mais bien le booléen **False** qui est égal à la valeur **0**. En Python, je rappelle que le signe **=** est un signe d'affectation tandis que le signe **==** est le signe d'égalité.

Je vais donc évaluer l'expression (**faux == 0**) pour savoir si elle est vraie ou fausse et je vais stocker la valeur retournée dans cette même variable **faux**.

```
[code language= »python »]
faux = False
faux = (faux == 0) # Évaluation d'une expression (vraie ou fausse?)
print (faux)
[/code]
```

**True**

L'expression **faux == 0** est bien vraie (**True**). J'ai stocké le résultat dans la variable **faux** et maintenant **faux = True**!



## Évaluation d'une expression

Bon, il s'agissait d'un exemple qui n'était pas du tout pédagogique. On voulait simplement vous expliquer de manière ludique que Python évalue des expressions pour savoir si elles sont fausses ou vraies. Prenons un autre exemple plus parlant:

```
[code language= »python »]
nombre_1 = 5
nombre_2 = 8
result = (nombre_1 < nombre_2)
print(result)
[/code]
```

True

La variable **result** évalue si l'expression (**nombre\_1 < nombre\_2**) est vraie ou fausse. Il se trouve qu'elle est vraie puisque 5 est inférieur à 8. Donc le résultat est **True**.

## Le type bool ()

Les variables **faux** et **vrai** sont de types **bool ()** car elles stockent les valeurs **False** et **True**.

```
[code language= »python »]
faux = False
vrai = True
print (type(faux))
print(type(vrai))
[/code]
```

<class 'bool'>

<class 'bool'>

Si on déclare une variable de cette manière : **var = bool()**, alors **var** stockera la valeur **False**.

```
[code language= »python »]
var = bool()
print(var)
[/code]
```

False

# Les conditions sont sous l'emprise des booléens

Lorsque Python évalue une condition, soit le résultat est **False** (`== 0`), soit il est **True** (`== 1`). C'est comme cela qu'un ordinateur pense... uniquement avec des zéros et des uns!

```
[code language= »python »]
a = 5
if a < 9:
    print(a, "est inférieur à 9")
else:
    print(a, "est supérieur à 9")
[/code]
```

Résultat: 5 est inférieur à 9

Qu'est ce qui s'est passé dans la tête de l'ordinateur? Il a évalué la condition. Est-ce que la variable **a** est inférieure à 9? Oui, c'est vrai. Donc la condition vaut **1** car elle est **True**. Remplacez **5** par **12** par exemple et vous verrez que la condition vaudra **0**. Elle sera **False**. Par conséquent, c'est le deuxième message qui s'affichera :

Résultat: 12 est inférieur à 9

Pour vous en convaincre, on va de nouveau utiliser une expression conditionnelle avec la même variable:

```
[code language= »python »]
a = 5
result = (a < 9)
print(result)
[/code]
```

True

L'expression conditionnelle est vraie.

À l'instar d'autres langages de programmation tels que le « C » par exemple, Python considère que toute valeur autre que **0** est vraie (**True**). Seule la valeur **0** est fausse (**False**).

- Le nombre entier **0** est **False** tandis que les nombres entiers **6** et **-6** sont **True**.
- Une chaîne de caractères vide est **False**, une liste vide aussi.
- Une chaîne de caractères avec au moins un élément est **True**, une liste avec au moins un élément est **True** aussi.

```
[code language= »python »]
chain_car = ""
if chain_car:
    print("vrai")
else:
    print("faux")
```

```
chain_car2 = "Boole"
if chain_car2:
```

```
print ("vrai")
else:
print("faux")
[/code]
```

faux

vrai

```
[code language= »python »]
liste = []
if liste:
print("vrai")
else:
print("faux")
[/code]
```

```
liste2 = ["Boole", "George"]
if liste2:
print("vrai")
else:
print("faux")
[/code]
```

faux

vrai

Comme Python considère que toute valeur autre que **0** est **True**, on a même pas besoin d'écrire par exemple « **if a == 5:** »

```
[code language= »python »]
a = 5
if a == 5:
print("C'est vrai")
else:
print("C'est faux")
[/code]
```

On peut se contenter d'écrire « **if a:** »

```
[code language= »python »]
a = 5
if a:
print("C'est vrai")
else:
print("C'est faux")
[/code]
```

La fonction **print ()** affichera « **C'est vrai** » car elle a évalué que la condition était **True**.

# L'instruction else

Dans une condition, le code indenté qui suit l'instruction **else** s'exécute lorsque l'évaluation est **False** (ou les évaluations précédentes sont toutes **False**). L'instruction **else** exécute un code qui est **False**.

## L'instruction elif

Le code indenté qui suit l'instruction **elif** s'exécute lorsque les tests précédents **if** et **elif** sont déclarés **False**. Mais contrairement à **else**, l'instruction **elif** exécute un code qui est **True**.

```
[code language= »python »]
a = 5
if a == 6: # False
    print("C'est vrai")
elif a == 7: # False
    print("C'est vrai")
elif a == 5: # True
    print("C'est vrai")
else: # False
    print("C'est faux") [/code]
```

## L'instruction not

En Python, l'instruction **not** transforme **False(0)** en **True(1)** et **True(1)** en **False(0)**.

a	not a
False (0)	True (1)
True (1)	False (0)

```
[code language= »python »]
a = 5
a = not a
print(a)
[/code]
```

False

## L'instruction and

L'expression **a and b** est vraie si **a** et **b** sont vrais tous les deux. Elle est fausse si **a** ou **b** est faux ou si **a** et **b** sont faux tous les deux. Voici un tableau qui résume ce que je viens d'écrire:

<b>a</b>	<b>b</b>	<b>a and b</b>
False (0)	False (0)	False (0)
False (0)	True (1)	False (0)
True (1)	False (0)	False (0)
True (1)	True (1)	True (1)

```
[code language= »python »]
a, b = 5, 13
if a == 5 and b == 13:
    print("C'est vrai.")
else:
    print("C'est faux.")
[/code]
```

C'est vrai.

## L'instruction or

L'expression **a or b** est vraie si au moins l'une des deux variables est vraie. Elle est fausse si les deux variables sont fausses.

<b>a</b>	<b>b</b>	<b>a or b</b>
False (0)	False (0)	False (0)
False (0)	True (1)	True (1)
True (1)	False (0)	True (1)
True (1)	True (1)	True (1)

```
[code language= »python »]
a, b = 5, 13
if a == 5 or b == 18:
    print("C'est vrai.")
else:
    print("C'est faux.")
[/code]    C'est vrai.
```

## Les boucles while et for

On retrouve les booléens dans les boucles while et for qui exécutent un programme tant que la condition est vraie (**True**). À partir du moment où elle devient fausse (**False**), la boucle s'interrompt. Il est donc aisé (mais pas très malin!) de créer une boucle infinie si on fait en sorte que la condition soit vraie pour l'éternité (ou au moins pour les capacités de la mémoire RAM!).

```
[code language= »python »]  
a = 5  
while a :  
print("Au secours! Je viens de créer une boucle infinie.")  
[/code]
```

À partir de là, seul un **Ctrl + C** est à même d'interrompre l'irréversible processus.

```
Au secours! Je viens de créer une boucle infinie.  
Au secours! Je viens de créer une boucle infinie.  
Au secours! Je viens de créer une boucle infinie.
```

line1344, in write return self.shell. Write (s, self.tags)

KeyboardInterrupt

## Conclusion

Les booléens sont au cœur de la conscience-machine. Un ordinateur ne comprend que le langage binaire. C'est la raison pour laquelle il évalue la fausseté ou la véracité d'une expression uniquement en utilisant le 0 et le 1.

# LES FONCTIONS

Une fonction est une portion de code qui effectue une tâche ou un calcul relativement indépendant du reste du programme. Une fonction comporte des paramètres d'entrée sur lesquels elle travaille avant de retourner une valeur de sortie qui peut être stockée dans une variable. Si la fonction ne retourne rien (**None**), on parlera plutôt de procédure. Une fonction est un sous-programme réutilisable partout dans le code. Il est donc inutile de réécrire des instructions identiques. Il suffit de les inclure dans une fonction et d'appeler cette dernière à chaque fois que cela est nécessaire.

## Les fonctions prédéfinies

Nous avons déjà utilisé quelques fonctions sans le savoir :

- **print ()** : affiche n'importe quel nombre de valeurs passées en arguments.
- **input ()** : interrompt le programme jusqu'à ce que l'utilisateur entre une chaîne de caractères puis presse la touche « Entrée ». Alors, le programme redémarre et affiche la chaîne de caractères.
- **int ()** : transforme une chaîne de caractères en nombre entier (si celle-ci s'y prête) ou un nombre décimal en nombre entier (en ignorant ce qui est après la virgule).
- **float ()** : transforme une chaîne de caractères en nombre décimal (si celle-ci s'y prête) ou un nombre entier en nombre décimal (exemple : 9.0)
- **str ()** : transforme un nombre entier ou un nombre décimal en chaîne de caractères.

## Créer une fonction

Pour créer une fonction, on utilise le mot-clé **def**. Voici comment une fonction est construite :

```
def nom_de_la_fonction(paramètre_1, paramètre_2,...): # Ne pas oublier les deux petits points!  
    "Docstring" # La docstring est une documentation qui explique ce que la fonction réalise  
    Bloc_d_instruction indenté # Le bloc d'instructions est indenté.  
    return résultat # Valeur retournée
```

Pour appeler la fonction et stocker dans une variable la valeur qu'elle retourne, il suffit d'écrire :

```
variable = nom_de_la_fonction(paramètre_1, paramètre_2,...)
```

Prenons l'exemple d'une fonction qui divise par deux la valeur passée en argument :



```

1#!/usr/bin/python3
2# -*- coding: utf8 -*-
3
4#---Fonction-----
5
6def division_par_deux(x):
7    "Fonction qui divise par deux le nombre x passé en argument"
8    return x/2 # Valeur retournée
9
10#===Programme principal=====
11
12a = 8 # Variable à laquelle j'affecte le nombre entier 8.
13b = division_par_deux(a) # Appel de la fonction en lui passant la variable a comme argument
14print(b)
15
16# Je déclare la variable "b" qui va stocker le résultat retourné par la fonction division_par_deux
17# à laquelle j'ai passé la variable "a" en argument.
18# Cela équivaut à :
19# b = division_par_deux(8)
20# Donc la fonction divise 8 par 2 :
21#     return 8/2
22

```

La fonction doit toujours précéder l'appel de fonction.

## Passer plusieurs arguments à une fonction

Il est possible de passer plusieurs arguments à une fonction :

```

1#!/usr/bin/python3
2# -*- coding: utf8 -*-
3
4#---Fonction-----
5
6def division(x, y):
7    "Fonction qui divise par y le dividende x passé en argument"
8    return x/y # Valeur retournée
9
10#===Programme principal=====
11
12a = 15 # Variable à laquelle j'affecte le nombre entier 8.
13b = 3
14# Appel de la fonction en lui passant les variables "a" et "b" en arguments :
15resultat = division(a, b)
16print(resultat)
17

```

## Nombre d'arguments indéterminés

Il est possible de passer à une fonction un nombre infini d'arguments. Pour ce faire, il suffit de rajouter un astérisque devant le paramètre qui recevra les arguments. En fait, cela a pour effet de créer une liste :

```

1#!/usr/bin/python3
2# -*- coding: utf8 -*-
3
4#---Fonction-----
5
6def rajouter_deux(*args):
7    "Fonction qui rajoute 2 à chaque élément de la liste args"
8    for i in args : # Création d'une boucle "for" pour rajouter 2 à chaque élément de la liste.
9        print(i + 2) # La fonction print affiche chaque élément de la liste args
10
11# Pas de valeur retournée. La fonction rajouter_deux ne renvoie rien (None).
12# C'est la fonction print() qui s'en charge.
13
14#===Programme principal=====
15
16rajouter_deux(8, 5, 4, 89, -6, 9.8, -75.9)
17

```

On exécute ce bout de code dans un terminal IDLE-3 PYSCRIPTER .

## À quoi sert la docstring?

La docstring est une chaîne de caractères indentée juste après la déclaration d'une fonction. Elle fournit une documentation accessible grâce à la fonction `help()` :

```
>>> help(rajouter_deux)
Help on function rajouter_deux in module __main__:

rajouter_deux(*args)
    Fonction qui rajoute 2 à chaque élément de la liste args
```

## Paramètres par défaut

Une fonction peut avoir des paramètres avec des arguments par défaut. Ces paramètres ne doivent pas précéder les paramètres sans arguments par défaut, sous peine de lever une exception.

Voici deux exemples. Dans l'appel de fonction du premier exemple, on ne passe qu'un seul argument qui correspond au premier paramètre de la fonction. Le deuxième paramètre utilise l'argument par défaut (2).

```
1#!/usr/bin/python3
2# -*- coding: utf8 -*-
3
4#---Fonction-----
5
6def diviser(x, diviseur = 2): # diviseur est un paramètre avec un argument par défaut
7    "Fonction qui divise x par diviseur"
8    return x/diviseur
9
10===Programme principal=====
11
12a = 10
13b = diviser(a) # Je passe une seule valeur en argument
14print(b)
```

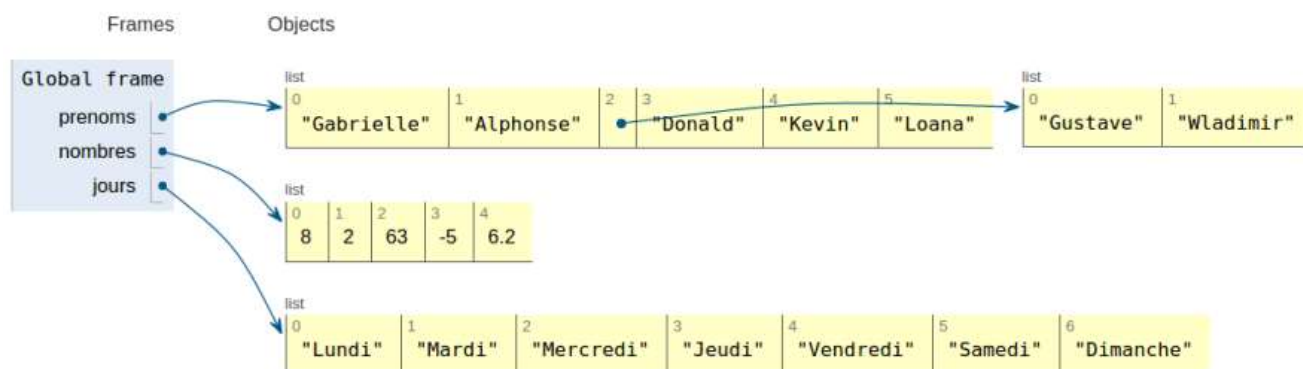
Dans ce deuxième exemple, on passe deux variables en arguments en modifiant l'argument du deuxième paramètre. On doit préciser quel paramètre est modifié (**diviseur = b**).

```
1#!/usr/bin/python3
2# -*- coding: utf8 -*-
3
4#---Fonction-----
5
6def diviser(x, diviseur = 2): # diviseur est un paramètre avec un argument par défaut
7    "Fonction qui divise x par diviseur"
8    return x/diviseur
9
10===Programme principal=====
11
12a = 15
13b = 4
14# Je passe deux variables en arguments en précisant quel est le paramètre par défaut
15# dont je modifie la valeur (diviseur = b)
16b = diviser(a, diviseur = b)
17print(b)
```

Si on crée une fonction en déclarant un paramètre par défaut avant les autres paramètres (exemple : `def diviser (diviseur = 2, x)`), Python retourne une exception et le programme ne s'exécute pas.

# LES LISTES

Une liste est un ensemble d'éléments séparés par des virgules et entourés de crochets. Ces éléments peuvent être de n'importe quel type: **str()**, **int()**, **float()**. Il existe même des listes de listes! Contrairement aux chaînes de caractères, il est possible de modifier une liste grâce au slicing et à différentes méthodes que je vais vous présenter dans ce chapitre.



## Déclarer une liste

Pour déclarer une liste vide, il existe deux syntaxes différentes:

```
[code language= »python »] liste_1 = []
liste_2 = list()
print(liste_1, liste_2)[/code]
```

Résultat : [] []

Bien sûr, vous pouvez déclarer une liste et lui affecter directement des valeurs :

```
[code language= »python »] exemple_liste = [8, 6.9, "Kevin"]
print(exemple_liste)[/code]
```

Résultat : [8, 6.9, 'Kevin']

La deuxième syntaxe, **list ()**, est utilisée pour convertir par exemple un tuple en liste. Nous verrons les tuples dans un prochain chapitre mais sachez déjà qu'un tuple est un ensemble d'éléments immuable séparés par des virgules et entourés de parenthèses. En fait, un tuple est une liste qu'on ne peut pas modifier. La seule solution est donc de transformer le tuple en liste.

```
[code language= »python »] tuple_prenoms = ("Gustave", "Solange", "Alphonse")
liste_prenoms = list(tuple_prenoms)
print(liste_prenoms)[/code]
```

Résultat: ['Gustave', 'Solange', 'Alphonse']

## La méthode `append()`

Grâce à la méthode **`append()`**, vous pouvez ajouter des éléments qui seront affectés à la fin de la liste.

```
[code language= »python »]
exemple_liste = [8, 6.9, "Kevin"]
exemple_liste.append("Loana")
print(exemple_liste)[/code]
```

Résultat : `[8, 6.9, 'Kevin', 'Loana']`

## La fonction intégrée `len()`

La fonction intégrée **`len()`** retourne un entier qui correspond au nombre d'éléments contenus dans la liste passée en argument.

```
[code language= »python »]
print(len(exemple_liste))
[/code]
```

Résultat : `4`

## Accéder à un élément d'une liste grâce à son indice

**`exemple_liste`** contient quatre éléments indiqués à partir de zéro, c'est-à-dire que l'indice du quatrième élément est **3**. Avec l'indice, il est donc possible d'accéder à un élément précis de la liste.

```
[code language= »python »]
# Je place l'indice de l'élément recherché entre crochets:
element = exemple_liste [2]
print(element)[/code]
```

Résultat : `'Kevin'`. Cela correspond au troisième élément de la liste c'est-à-dire celui qui porte l'indice 2.

## La méthode `insert()`

La méthode **`insert(indice, nouvel_element)`** permet d'insérer un nouvel élément dans une liste:

```
[code language= »python »]
exemple_liste.insert(1, "Gabrielle")
print(exemple_liste)[/code]
```

Résultat : `[8, 'Gabrielle', 6.9, 'Kevin', 'Loana']`

Cette méthode insère l'élément passé en argument à l'indice passé en argument. « Gabrielle » se retrouve donc à l'indice 1 et les valeurs qui suivent, changent d'indice (i devient i + 1). Ainsi, le nombre décimal 6.9 qui était à l'indice 1, se retrouve à l'indice 2.

## La méthode `extend()`

La méthode **`extend()`** permet de concaténer deux listes :

```
[code language= »python «]
jours = ["Lundi", "mardi", "mercredi"]
jours2 = ["Jeudi", "Vendredi"]
jours.extend(jours2)
print(jours)[/code]
```

Résultat : [« Lundi », « Mardi », « Mercredi », « Jeudi », « Vendredi »]

Attention! Il ne faut pas confondre `extend()` et `append()`!

`extend()` effectue une concaténation de listes tandis que `append()` rajoute l'élément passé en argument, à la fin de la liste sur laquelle la méthode est appliquée.

Dans l'exemple précédent, voici ce que j'aurais obtenu si j'avais utilisé la méthode **`append()`**:

```
[code language= »python «]
jours = ["Lundi", "mardi", "mercredi"]
jours2 = ["Jeudi", "Vendredi"]
jours.append(jours2)
print(jours)[/code]
```

Résultat : [« Lundi », « mardi », « mercredi », [« Jeudi », « Vendredi »]]

Je n'ai pas concaténé deux listes mais j'ai rajouté la liste **`jours2`** aux éléments de la liste **`jours`**. Cette dernière ne contient donc pas 5 éléments (comme dans le premier exemple) mais quatre éléments : Trois chaînes de caractères et une liste! Pour vous en convaincre, il vous suffit d'utiliser la fonction intégrée **`len()`**.

## Accéder à un élément d'une liste qui se trouve elle-même dans une autre liste

« **`jours`** » est donc une liste qui contient une autre liste plus petite. Nous avons vu qu'il était possible d'accéder à un élément en utilisant l'indice entouré de crochets.

```
[code language= »python «]
jours = ["Lundi", "mardi", "mercredi", ["Jeudi", "Vendredi"]]
# Je place l'indice de l'élément recherché entre crochets:
element = jours[2]
print(element)[/code]
```

Résultat : Mercredi

Mais comment faire pour accéder à « Vendredi » qui est un élément de la petite liste qui se trouve en dernière position de la liste principale? C'est très simple! Il suffit d'utiliser cette syntaxe :

```
[code language= »python »]
jours = ["Lundi", "mardi", "mercredi", ["Jeudi", "Vendredi"]]
# Je place les indices de l'élément recherché entre crochets:
element = jours[3][1]
print(element)/code]
```

Résultat: Vendredi

Cette valeur correspond donc à **jours [3][1]**.

- **3** est l'indice de la petite liste dans la liste principale.
- **1** est l'indice de « **Vendredi** » dans la petite liste.

## Utiliser le slicing pour modifier une liste

Il est tout à fait possible d'utiliser les crochets et les indices pour insérer ou supprimer des éléments dans une liste. Cette technique s'appelle le **slicing**. Cela se traduit par **découpe en tranches**. C'est clairement plus délicat à manipuler que les méthodes mais c'est très efficace et d'une grande souplesse.

Par exemple, au lieu d'appliquer la méthode **list.insert ()**, on peut utiliser la syntaxe **list [i:i]**.

```
[code language= »python »]
jours = ["Lundi", "Mardi", "Jeudi"]
# Je souhaite insérer "Mercredi"
jours[2:2] = ["Mercredi"]
print(jours)/code]
```

Résultat: ['Lundi', 'Mardi', 'Mercredi', 'Jeudi']

**Attention! L'élément à insérer doit forcément être une liste. Si vous n'insérez qu'un seul élément comme dans l'exemple ci-dessus, il faut le présenter entre crochets et par conséquent le convertir en une liste d'un seul élément.**

**Si vous oubliez les crochets, Python va insérer chaque caractère de la chaîne « **Mercredi** » en tant qu'élément :**

['Lundi', 'Mardi', 'M', 'e', 'r', 'c', 'r', 'e', 'd', 'i', 'Jeudi']

Pour faire un **print ()** d'une partie des éléments d'une liste, par exemple les éléments indicés 1 et 2 dans la liste **['Gabrielle', 'Alphonse', 'Kevin', 'Loana']**, le code sera celui-ci :

```
[code language= »python »]
prenoms = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana']
print(prenoms[1:3])
/code]
```

Résultat: ['Alphonse', 'Kevin']

Dans cette tranche, l'élément indicé 1 est pris en compte tout comme l'élément indicé 2 mais pas l'élément indicé 3! Voici un schéma qui permet de mieux comprendre comment fonctionne le slicing. La tranche **[1:3]** correspond à « **Alphonse** » et « **Kevin** ». La tranche **[3:4]** correspond au prénom « **Loana** ».



['Gabrielle', 'Alphonse', 'Kevin', 'Loana']

↑

↑

↑

↑

↑

0

1

2

3

4

Si j'insère « **Gustave** » dans la tranche [1:3], alors ce prénom remplace « **Alphonse** » et « **Kevin** ».

```
[code language= »python »]
prenoms = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana']
prenoms[1:3] = ["Gustave"]
print(prenoms)
[/code]
```

Résultat : ['Gabrielle', 'Gustave', 'Loana']

## Autres exemples de slicing

- `prenoms[:] = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana']` (liste entière)
- `prenoms[1:] = ['Alphonse', 'Kevin', 'Loana']`
- `prenoms[:-1] = ['Gabrielle', 'Alphonse', 'Kevin']`
- `prenoms[:-2] = ['Gabrielle', 'Alphonse']`
- `prenoms[-1] = prenoms[3] = 'Loana'`
- `prenoms[::2] = ['Gabrielle', 'Kevin']`
- `prenoms[::3] = ['Gabrielle', 'Loana']`
- `prenoms[::-1] = ['Loana', 'Kevin', 'Alphonse', 'Gabrielle']` (inverse les éléments)

Il n'est pas toujours évident de manipuler les indices négatifs. Pour vous faciliter la tâche, il faut savoir que `prenoms [-3]` correspond à `prenoms [len(prenoms) - 3]` c'est-à-dire à `prenoms[1]`.

`[-1]` est un indice négatif particulièrement intéressant car il permet d'accéder au dernier élément d'une liste sans connaître la longueur de celle-ci.

Il est possible de parcourir une liste en utilisant un pas qui ne prend qu'un élément sur deux `[::2]` ou qu'un élément sur trois `[::3]` par exemple. Un pas de `[::-1]` inverse les éléments d'une liste.

## Quelques méthodes associées aux objets de type list ()

Nous avons déjà vu `list.append ()`, `list.extend ()` et `list.insert ()`. Il existe également :

- `list.sort ()`

Effectue le tri d'une liste dans l'ordre numérique croissant (ou dans l'ordre alphabétique croissant).

```
[code language= »python »]
prenoms = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana']
nombres = [8, 52, 0, 3.2, -5.6, 7, -23]
prenoms.sort()
nombres.sort()
print(prenoms)
```



```
print(nombres)
[/code]
```

Résultat : ['Alphonse', 'Gabrielle', 'Kevin', 'Loana']  
[-23, -5.6, 0, 3.2, 7, 8, 52]

**list.sort ()** peut prendre l'argument **reverse = True**. Dans ce cas, le tri est décroissant :

```
[code language= »python »]
prenoms = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana']
nombres = [8, 52, 0, 3.2, -5.6, 7, -23]
prenoms.sort(reverse = True)
nombres.sort(reverse = True)
print(prenoms)
print(nombres)
[/code]
```

Résultat : ['Loana', 'Kevin', 'Gabrielle', 'Alphonse']  
[52, 8, 7, 3.2, 0, -5.6, -23]

- **list.index ()**

**list.index ()** retourne l'index de l'élément passé en argument. Si l'élément est absent de la liste, Python lève une exception. Si l'élément est présent plusieurs fois, la méthode retourne l'index de la première occurrence.

```
[code language= »python »]
prenoms = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana']
nombres = [8, 52, 0, 3.2, -5.6, 7, -23]
i = prenoms.index('Gabrielle')
i2 = nombres.index(53)
print(i)
print(i2)
[/code]
```

Résultat: 0

**ValueError: 53 is not in list**

Si une valeur est présente plusieurs fois, il est possible de passer en argument l'index à laquelle doit débiter la recherche sous la forme `list.index (valeur, i)`:

```
[code language= »python »]
prenoms = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana', 'Gabrielle']
nombres = [8, 52, 0, 3.2, -5.6, 7, -23]
i = prenoms.index('Gabrielle', 3)
print(i)
[/code]
```

Résultat: 4 (la première valeur 'Gabrielle' est ignorée)

- **list.reverse ()**

Cette méthode inverse les valeurs de la liste.

```
[code language= »python »]
prenoms = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana']
prenoms.reverse()
print(prenoms)
[/code]
```

Résultat: ['Loana', 'Kevin', 'Alphonse', 'Gabrielle']

- **list.pop ()**

**list.pop ()** est une drôle de méthode. Elle supprime et retourne par défaut la dernière valeur de la liste. Cette méthode peut prendre un indice en argument. Dans ce cas, c'est la valeur indiquée qui est supprimée et retournée.

```
[code language= »python »]
prenoms = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana']
popped = prenom.pop()
print(popped, prenom)
[/code]
```

Résultat: Loana ['Gabrielle', 'Alphonse', 'Kevin']

```
[code language= »python »]
prenoms = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana']
popped = prenom.pop(1)
print(popped, prenom)
[/code]
```

Résultat: Alphonse ['Gabrielle', 'Kevin', 'Loana']

- **list.remove ()**

Cette méthode supprime la première occurrence de la valeur passée en argument. Voici un exemple avec une liste contenant deux « **Kevin** ».

```
[code language= »python »]
prenoms = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana', 'Kevin']
prenoms.remove('Kevin')
print(prenoms)
[/code]
```

Résultat: ['Gabrielle', 'Alphonse', 'Loana', 'Kevin']

- **list.count ()**

Cette méthode compte le nombre de fois où une occurrence apparaît dans une liste et elle retourne la valeur correspondante.

```
[code language= »python »]
prenoms = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana', 'Kevin']
n = prenom.count('Kevin')
```

```
print(n)
[/code]
```

Résultat: 2

## Tester l'appartenance d'une valeur à une liste

L'instruction conditionnelle **if... in** autorise très facilement ce genre de test. Elle valide la présence ou l'absence de la valeur recherchée dans la liste.

```
[code language= « python »]
prenoms = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana']
if 'Kevin' in prenoms:
    print("Le test d'appartenance est positif.")
else :
    print ("Ce prénom est absent de la liste.")
[/code]
```

Résultat : Le test d'appartenance est positif.

```
[code language= « python »]
prenoms = ['Gabrielle', 'Alphonse', 'Kevin', 'Loana']
result = 'Baptiste' in prenoms # Retourne un booléen (True ou False)
print(result)
[/code]
```

Résultat : False

# LA COMPRÉHENSION DE LISTE

La compréhension de liste est une expression qui permet de construire une liste à partir de tout autre type itérable (liste, tuple, chaîne de caractères...). Le résultat obtenu est toujours une liste.

## Une compréhension de liste toute simple

Nous allons créer une liste de nombres entiers :

```
liste_initiale = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

À l'aide d'une boucle **for** et de la méthode **append ()**, nous allons ajouter à une nouvelle liste baptisée **liste\_2**, chaque élément de **liste\_initiale** multiplié par deux.

```
[code language= »python »]
liste_initiale = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
liste_2 = []
for n in liste_initiale:
    liste_2.append(n*2)
print(liste_2)
[/code]
```

Résultat : liste\_2 = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]

On a donc utilisé une boucle **for** pour écrire ce petit bout de code de cinq lignes. Mais on aurait tout aussi bien pu utiliser une compréhension de liste pour obtenir le même résultat. Voici la syntaxe de cette dernière :

```
[code language= »python »]
liste_initiale = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
liste_2 = [n*2 for n in liste_initiale]
print(liste_2)
[/code]
```

Résultat : liste\_2 = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]

Que remarque-t-on?

Le code est plus concis puisqu'il tient sur trois lignes au lieu de cinq. Vous allez me dire que la différence est minime et dans cet exemple tout simple, on doit reconnaître que vous avez raison! Mais attendez de voir la suite...

## Une compréhension de liste avec une condition

On souhaite créer une nouvelle liste de nombres pairs multipliés par 2 à partir de **liste\_initiale** qui contient, des nombres pairs et impairs. Il va donc falloir introduire une condition pour ignorer les nombres impairs. Comparons une nouvelle fois le code écrit avec une boucle **for** et celui écrit avec une compréhension de liste.

### Avec une boucle for

```
def main():  
    pass  
  
if __name__ == '__main__':  
    main()  
    liste_initiale = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]  
    liste_pair = []  
    for n in liste_initiale:  
        if n % 2 == 0:  
            liste_pair.append(n*2)  
    print(liste_pair)
```

Résultat : liste\_pair = [0, 4, 8, 12, 16, 20, 24, 28]

### Avec une compréhension de liste

On introduit la condition à la fin de la compréhension de liste. Au lieu d'avoir six lignes de code, on en obtient plus que trois.

```

• . def main():
• .     pass
• .
• . if __name__ == '__main__':
• .     main()
• .     liste_initiale = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
• .     liste_pair = [n*2 for n in liste_initiale if n % 2 == 0]
• 18 print(liste_pair)

```

Résultat : liste\_pair = [0, 4, 8, 12, 16, 20, 24, 28]

## Une compréhension de liste avec une expression conditionnelle

Encore plus fort, on souhaite créer une nouvelle liste avec les éléments de **liste\_initiale**. Les chiffres pairs seront multipliés par deux, les chiffres impairs par trois. Il va donc falloir introduire condition **if... else**.

### Avec une boucle for

```

• . def main():
• .     pass
• .
• . if __name__ == '__main__':
• .     main()
• .     liste_initiale = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
• .     nouvelle_liste = []
• .     for n in liste_initiale:
• 19         if n % 2 == 0:
• .             nouvelle_liste.append(n*2)
• .         else:
• 22             nouvelle_liste.append(n*3)
• .     print(nouvelle_liste)

```

Résultat : nouvelle\_liste = [0, 3, 4, 9, 8, 15, 12, 21, 16, 27, 20, 33, 24, 39, 28, 45]

### Avec une compréhension de liste

Ça se complique un peu car il faut utiliser une expression conditionnelle qui évalue si la condition est vraie (**True**) ou fausse (**False**). Mais le résultat est au rendez-vous puisqu'au lieu d'avoir huit lignes de code, je n'en ai que trois!

```

def main():
    pass

if __name__ == '__main__':
    main()
liste_initiale = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
nouvelle_liste = [n*2 if n % 2 == 0 else n*3 for n in liste_initiale]
print(nouvelle_liste)

```

Résultat : nouvelle\_liste = [0, 3, 4, 9, 8, 15, 12, 21, 16, 27, 20, 33, 24, 39, 28, 45]

## Une compréhension de liste avec une liste de listes

Il est possible de faire une compréhension de liste avec une liste de listes. Pour s'y retrouver dans l'imbrication des instructions, le plus simple est de comparer la syntaxe d'une boucle **for** avec celle d'une compréhension de liste.

### Avec une boucle for

```

def main():
    pass

if __name__ == '__main__':
    main()
liste_initiale = [[0, "a"], [2, "b"], [3, "c"]]
nouvelle_liste = []
for i in liste_initiale :
    for n in i :
        nouvelle_liste.append(n*2)
print(nouvelle_liste)

```

Résultat: nouvelle\_liste = [0, 'aa', 4, 'bb', 6, 'cc']

### Avec une compréhension de liste

```

def main():
    pass

if __name__ == '__main__':
    main()
liste_initiale = [[0, "a"], [2, "b"], [3, "d"]]
nouvelle_liste = [n*2 for i in liste_initiale for n in i]
print(nouvelle_liste)

```

Résultat: nouvelle\_liste = [0, 'aa', 4, 'bb', 6, 'cc']

### Avec une expression conditionnelle



Par exemple, on veut multiplier les nombres entiers par deux et les strings par trois.

```
[code language= »python »]
liste_initiale = [[0, 'a'], [2, 'b'], [3, 'c']]
nouvelle_liste = [n*2 if type(n) == int else n*3 for i in liste_initiale for n in i]
print(nouvelle_liste)
[/code]
```

[Résultat:](#) [0, 'aaa', 4, 'bbb', 6, 'ccc']

Dans ce dernier exemple, si on avait fait usage d'une boucle **for**, écriture de neuf lignes de code au lieu de trois!

```
[code language= »python »]
liste_initiale = [[0, 'a'], [2, 'b'], [3, 'c']]
nouvelle_liste = []
for i in liste_initiale:
    for n in i:
        if type(n) == int:
            nouvelle_liste.append(n*2)
        else:
            nouvelle_liste.append(n*3)
print(nouvelle_liste)
[/code]
```

## Les compréhensions de liste avec les autres types itérables

Comme vu en introduction, les compréhensions de liste fonctionnent avec les autres types itérable mais le résultat obtenu est toujours une liste. Par exemple, avec une chaîne de caractères :

```
[code language= »python »]
prenom = "Gustave"
liste_lettres = [lettre for lettre in prenom]
print(liste_lettres)
[/code]
```

[Résultat:](#) ['G', 'u', 's', 't', 'a', 'v', 'e']

## Conclusion

La compréhension de liste est une méthode puissante qui a remplacé les anciennes fonctions **map ()** et **filter ()** dont l'usage est aujourd'hui déconseillé. Bien utilisée, la compréhension de liste rend le code plus concis, plus élégant et plus facile à comprendre qu'avec les fonctions **map ()** et **filter**

() Il existe une page consacrée aux compréhensions de liste dans la documentation officielle du langage Python.

<https://www.python.org/dev/peps/pep-0202/>

Il existe également une compréhension de dictionnaire et d'ensemble que nous aborderons dans le chapitre consacré à ces deux structures de données.

## LES TUPLES

Un tuple est un ensemble d'éléments comparable aux listes mais qui, une fois déclaré, ne peut plus être modifié. Il s'agit donc d'une séquence immuable d'objets indicés qui peuvent être des nombres entiers ou décimaux, des chaînes de caractères, des listes, des tuples etc...

### Syntaxe et déclaration

D'un point de vue syntaxique, un tuple est un ensemble d'éléments séparés par des virgules. Cet ensemble d'éléments est entouré de parenthèses mais ce n'est pas une obligation. Cela permet toutefois d'améliorer la lisibilité du code.

Déclarons un tuple :

```
[code language= « python »]tuple_1 = (8, "Solange", 5.3 )[/code]
```

```
[Code language= « python » ]tuple_1 = 8, "Solange", 5.3[/code]
```

Cela revient au même sauf que c'est moins lisible.

```
[code language= « python » ] print ( type(tuple_1) )[/code]
```

<class 'tuple'>

Sans le savoir, nous avons déjà rencontré des tuples, notamment lorsque nous avons fait des affectations multiples :

```
[code language= « python » ] a, b = 5, 6[/code]
```

Cette instruction d'affectation correspond à :

```
[code language= « python » ] (a, b) = (5, 6) [/code]
```

Si nous souhaitons déclarer un tuple ne contenant qu'une seule valeur, il faut absolument que cette dernière soit suivie d'une virgule. En outre, il est vivement conseillé de l'inclure entre parenthèses.

```
[code language= « python » ] c = (9,)
```

```
Print (type(c)) [/code]
```

<class 'tuple'>

Si nous oublions la virgule (et malgré la présence de parenthèses), le type de la variable ne sera pas un tuple :

```
[code language= « python » ] c = (9)
```

```
Print (type(c) ) [/code]
```

<Class 'int'>

## Utilité d'un tuple par rapport à une liste

Comme nous l'avons vu plus haut, il permet de faire des affectations multiples.

```
[code language= « python » ] a, b = 5, 6[/code]
```

Le tuple permet également de renvoyer plusieurs valeurs lors d'un appel de fonction.

```
[code language= « python » ]
def information_personne (prenom, nom, age) :
    "Traitement des informations transmises"
    prenom = prenom.capitalize() # Première lettre en majuscule.
    nom = nom.upper () # remplace toutes les minuscules en majuscules
```

```
age = int (age)
return prenom, nom, age # Tuple sans parenthèses
```

```
p = input ("Entrez votre prénom: ")
n = input("Entrez votre nom: ")
a = input("Entrez votre âge: ")
```

```
prenom, nom, age = information_personne (p, n, a) # Affectation multiple
print ("Prénom:", prenom + '\n' + "Nom:", nom + '\n' + "Âge:", age)[/code]
```

```
Entrez          votre          prénom:          alphonsine
Entrez          votre          nom:              Chafouin
Entrez votre âge: 92
```

Résultat:

```
Prénom:          Alphonsine
Nom:             CHAFOUIN
Âge: 92
```

Un tuple est « protégé en écriture ».

Vous allez me dire que dans le code ci-dessus, nous pouvons obtenir le même résultat en renvoyant une liste (**return [prenom, nom, age]**). Mais l'avantage d'un tuple, c'est qu'il s'agit d'une séquence non modifiable (immuable) donc **protégée en écriture**. Nous sommes sûrs que les données transmises par la fonction **input ()** ne seront pas modifiées en cours d'exécution par un autre programme. Dans le code ci-dessus, je n'ai aucun moyen de modifier les informations transmises.

Par contre, dans le code ci-dessous où je renvoie une liste, je peux modifier les données transmises et rajeunir Mamie Alphonsine !

```
[code language= « python »]
def information_personne(prenom, nom, age) :
    "Traitement des informations transmises"
    prenom = prenom.capitalize() # Première lettre en majuscule.
    nom = nom.upper () # remplace toutes les minuscules en majuscules
    age = int (age)
    return [prenom, nom, age] # liste
```

```
p = input ("Entrez votre prénom: ")
n = input ("Entrez votre nom: ")
a = input ("Entrez votre âge: ")
```

```
identification = information_personne(p, n, a) #Liste
identification[2] = '45' # Je modifie l'âge
print("Prénom:", identification[0] + '\n' + "Nom:", identification[1] + '\n' + "Âge:",
identification[2])[/code]
```

```
Entrez          votre          prénom:          alphonsine
Entrez          votre          nom:              Chafouin
Entrez votre âge: 92
```

Prénom:  
Nom:  
Âge: 45

Alphonsine  
CHAFOUIN

Un tuple est moins gourmand en ressources système qu'une liste.

Il a besoin de moins de mémoire et il s'exécute plus rapidement.

## Opérations sur les tuples

Il n'existe pas de méthodes associées aux tuples

Les tuples sont des séquences non modifiables donc il n'est pas possible d'utiliser les méthodes **remove ()** ou **append ()** par exemple.

Il n'est pas possible également d'utiliser l'opérateur **[]** pour insérer ou remplacer un élément :

```
[code language= « python »]  
tuple_1 = (5, 2, 25, 56)  
tuple_1[2] = 23  
print(tuple_1)[/code]
```

**TypeError: 'tuple' object does not support item assignment.**

L'instruction **in**

Mais il est possible d'utiliser l'instruction **in** pour faire un test d'appartenance

```
[code language= « python »]  
tuple_1 = (5, 2, 25, 56)  
print(2 in tuple_1)[/code]
```

Résultat: True

## La fonction intégrée **len ()**

Il est également possible d'utiliser la fonction intégrée **len ()** pour déterminer la longueur d'un tuple.

```
[code language= « python »]  
tuple_1 = (5, 2, 25, 56)  
print(len(tuple_1))[/code]
```

Résultat : 4

## La boucle **for** et la compréhension de liste

Tout comme les chaînes de caractères ou les listes, les tuples peuvent être parcourus par une boucle **for** ou une **compréhension de liste**. Dans ce dernier cas, le résultat obtenu est toujours une liste.

## Concaténation et multiplication

Nous pouvons créer un nouveau tuple par concaténation ou par multiplication

```
[code language= « python »]
tuple_1 = (5, 2, 25, 56)
tuple_2 = ("Jacky", "Hervé")
tuple_3 = tuple_1 + tuple_2
print(tuple_3)[/code]
```

Résultat: (5, 2, 25, 56, « Jacky », « Hervé »)

```
[code language= « python »]
tuple_prenoms = ("Jacky", "Hervé")
tuple_prenoms = tuple_prenoms * 3
print(tuple_prenoms)[/code]
```

Résultat: ('Jacky', 'Hervé', 'Jacky', 'Hervé', 'Jacky', 'Hervé')

Dans ce dernier exemple, on nous a dit qu'un tuple n'était pas modifiable et on vient juste de modifier **tuple\_prenoms** ! »

**Faux !** On n'a appliqué aucune méthode de modification sur **tuple\_prenoms**. On a seulement écrasé la valeur contenue dans la variable qui porte le nom de **tuple\_prenoms** en lui affectant une nouvelle valeur.

## Conclusion

Les tuples sont des séquences non modifiables qui s'avèrent bien utiles pour protéger en écriture des données transmises. En outre, ils sont moins gourmands en ressource système et sont traités plus rapidement.

Puisqu'il est immuable, il est possible d'utiliser un tuple comme clé de dictionnaire mais à la seule condition que le tuple ne contienne que des éléments non modifiables !

# LES DICTIONNAIRES

Nous habitons en Allemagne et l'idiome guttural de ce pays n'est pas notre langue maternelle. Par conséquent, lorsqu'un mot nous manque, On consulte un dictionnaire. On se rend directement à l'emplacement précis du mot français, par exemple **Ambulance**, et ce mot français est comme une clé qui nous donne accès à la valeur correspondante dans la langue de [Rammstein](#) :

Ambulance : **Krankenwagen**

Figurez-vous qu'il existe exactement le même procédé en Python et c'est bien plus pratique qu'une liste. Les dictionnaires sont la solution idéale pour réaliser un test d'appartenance et en extraire une valeur.

## Limitation des listes

Le problème des listes, c'est qu'il s'agit d'une séquence d'objets indicés, c'est-à-dire que pour avoir accès à un élément précis, il faut connaître son indice :

```
[code language= « python »]
mots_fr = ["Ambulance", "Hôpital", "Infirmière"]
mots_allemands = ["Krankenwagen", "Krankenhaus", "Krankenschwester"]
[/code]
```

Par exemple, on sait que l'indice d'**Infirmière** est le nombre entier 2. On consulte la liste allemande. Que trouve-t-on à l'indice 2? **Krankenschwester**.

Pas très pratique tout ça! En plus, est-ce que l'on est sûr que les traductions sont rangées dans le bon ordre. Est-ce que le mot allemand qui est à l'indice 1 (**Krankenhaus**) correspond bien au mot français au même indice (**Hôpital**)? Si ça se trouve, c'est dans le désordre!

Pour éviter ce genre de désagréments, on pourrait alors imaginer une liste de listes contenant des paires mot\_allemand – mot\_français:

```
[code language= « python »] dictionnaire = [{"Ambulance", "Krankenwagen"}, {"Hôpital",
"Krankenhaus"}, {"Infirmière", "Krankenschwester"}], [(...), (...)] [/code]
```

**Et pour trouver la traduction, on pourrait utiliser ce code:**

```
[code language= »python «]
dictionnaire = [{"Ambulance", "Krankenwagen"}, {"Hôpital", "Krankenhaus"}, {"Infirmière",
"Krankenschwester"}]
for i, element in enumerate(dictionnaire):
if "Hôpital" in dictionnaire[i]:
print(dictionnaire[i][1]) [/code]
```

Résultat: « Krankenhaus »

« dictionnaire français-allemand ».



Attention on rappelle que les listes sont des séquences. Cela signifie que pour faire un test d'appartenance, elles sont parcourues de l'indice **0** jusqu'à l'indice de l'élément recherché. En clair, si l'on cherche la traduction de **zygomatique** qui se trouve à la fin du dictionnaire, il va falloir que l'on tourne toutes les pages une par une et lire tous les mots un par un! Tu sais quoi? On va se donner rendez-vous en 2025. Tu m'apporteras ta traduction. Allez vas-y, tourne les pages! **Le dictionnaire est une liste !!!**

## Définition et déclaration d'un dictionnaire

### Définition

Un dictionnaire est un ensemble dont les éléments sont des paires **clé-valeur**. Au niveau syntaxique, un dictionnaire est contenu entre deux accolades. Les éléments sont séparés par des virgules tandis que les paires clé-valeur sont séparées par deux points. Voici ce que ça donne:

**nom\_du\_dictionnaire = {clé : valeur, clé : valeur, clé : valeur, (...) : (...)}**

Un dictionnaire est modifiable et les valeurs peuvent être n'importe quel objet (immuable ou modifiable, il n'y a aucune restriction) mais les clés doivent absolument être des objets immuables (string, nombre entier, nombre décimal, tuple). Si la clé est un tuple, celui-ci ne doit contenir que des éléments immuables. Par exemple, ce genre de clé **([0, 1], [2, 3])** qui est un tuple contenant des listes modifiables va lever une exception (erreur):

**TypeError: unhashable type: 'list'**

### Déclarer un dictionnaire

- Pour déclarer un dictionnaire, on utilise des accolades :

```
[code language= « python »]
dictionnaire = {} # dictionnaire vide
[/code]
```

- On vient de créer un dictionnaire vide mais je peux très bien créer un dictionnaire et y placer des éléments dès sa déclaration :

```
[code language= « python »]
capitales = {"France": "Paris", "Allemagne" : "Berlin"}
[/code]
```

Dans cet exemple, « **France** » est une clé tandis que « **Paris** » est une valeur.

- Il existe une autre méthode pour créer un dictionnaire. Elle consiste à créer une liste de tuples que l'on transforme en dictionnaire grâce au constructeur **dict ()**.

```
[code language= « python »]
liste_capitales = [("France", "Paris"), ("Allemagne", "Berlin")]
capitales = dict(liste_capitales)
print(capitales)
[/code]
```

**Résultat : {'Allemagne': 'Berlin', 'France': 'Paris'}**

- Il n'est pas possible de dupliquer une clé. Si on le fait, ce sera la dernière valeur entrée pour cette clé qui sera prise en compte.

```
[code language= « python »]
capitales = {"France": "Paris", "Allemagne": "Berlin", "France": "Marseille"}
print(capitales)
[/code]
```

Résultat: {'Allemagne': 'Berlin', 'France': 'Marseille'}

## Ajouter un élément (paire clé-valeur)

Si je veux rajouter un élément, c'est très simple, je crée une nouvelle paire clé-valeur. La clé est contenue entre des crochets et la valeur se trouve à la droite du signe d'affectation. La syntaxe est donc la suivante :

**nom\_du\_dico [clé] = valeur**

```
[code language= « python »]
capitales['Islande'] = "Reykjavik"
print(capitales)
[/code]
```

Résultat : {'Allemagne': 'Berlin', 'Islande': 'Reykjavik', 'France': 'Paris'}

Diantre! Les éléments ne sont plus dans le même ordre? Comment se fait-ce?

Ça n'a aucune espèce d'importance. L'ordre est aléatoire car un dictionnaire n'est pas une séquence, c'est une implémentation de [tables de hachage](#). Pour retrouver une valeur, nous avons seulement besoin de connaître sa clé.

## Accéder à une valeur

Voici comment on accède à une valeur:

```
[code language= « python »]
capitales = {'Allemagne': 'Berlin', 'Islande': 'Reykjavik', 'France': 'Paris'}
result = capitales["Allemagne"] #Je stocke le résultat dans une variable.
print(result)
[/code]
```

Résultat : « Berlin »

C'est simple et instantané! Le programme n'a pas besoin de parcourir le dictionnaire du début à la fin. Grâce à la clé (« **Allemagne** »), Python est en mesure de se rendre directement à la « page » souhaitée pour trouver la valeur et renvoyer cette dernière. Que le dictionnaire contienne dix éléments ou cinq cent millions, la vitesse d'exécution sera de toute façon identique!

En utilisant une clé qui n'est pas dans le dictionnaire, Python lève une exception (erreur):

```
[code language= « python »]
result = capitales["Syldavie"]
print(result)
[/code] KeyError: 'Syldavie'
```

## La méthode `get ()` comme test d'appartenance

Pour contourner cette exception et éviter d'avoir un message d'erreur qui stoppe le programme, il suffit d'utiliser la méthode `get ()` en lui passant deux arguments : la clé et le résultat renvoyé au cas où la clé serait absente du dictionnaire. Reprenons le code précédent :

```
[code language= « python »]
result = capitales.get("Syldavie", "Non répertorié.")
print(result)
[/code]
```

Résultat : Non répertorié.

## Tester l'appartenance avec l'instruction `in`

Il est possible de tester l'appartenance d'une clé à un dictionnaire grâce à la puissante instruction `in`.

```
[code language= « python »]
capitales = {"France": "Paris", "Allemagne": "Berlin", "Islande": "Reykjavik"}
if "Islande" in capitales:
    print(capitales["Islande"])
[/code]
```

Résultat : Reykjavik

L'instruction `in` ne teste que l'appartenance des clés et non pas l'appartenance des valeurs.

```
[code language= « python »]
capitales = {"France": "Paris", "Allemagne": "Berlin", "Islande": "Reykjavik"}
if "Reykjavik" in capitales:
    print("Test d'appartenance réussi")
else:
    print("Cette clé est absente du dictionnaire")
[/code]
```

Résultat: Cette clé est absente du dictionnaire

## Mettre à jour une valeur

Si on souhaite mettre à jour une valeur, c'est fort simple. Voici comment on procède:

```
[code language= « python »]
capitales = {"France": "Paris", "Allemagne": "Berlin", "Islande": "Reykjavik"}
capitales["France"] = "Marseille"
print(capitales)
[/code]
```

Résultat: {'France': 'Marseille', 'Allemagne': 'Berlin', 'Islande': 'Reykjavik'}

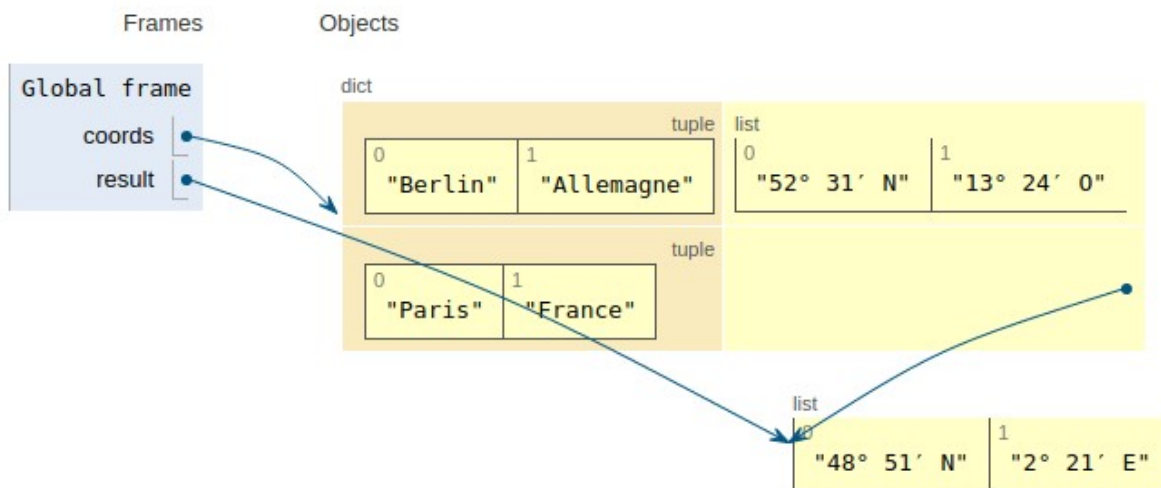
## Utiliser un tuple comme clé

C'est tout à fait possible, à la condition que le tuple ne contienne que des objets immuables. Quant à la valeur, elle peut être constituée par n'importe quel objet donc une liste ne pose aucun problème.

Dès lors, on peut très bien imaginer un dictionnaire avec en guise de clés, des tuples contenant les noms de capitales et les noms de pays et en guise de valeurs, des listes contenant la latitude et la longitude.

```
[code language= « python »]
coords = {("Paris","France":["48° 51' N", "2° 21' E"], ("Berlin","Allemagne"): ["52° 31' N", "13° 24' O"]}
result = coords [("Paris", "France")]
print(result)
[/code]
```

Résultat : ['48° 51' N', '2° 21' E']



Visualisation du code avec [Pythontutor](#)

## Supprimer un élément grâce à l'instruction del

Nous avons vu comment rajouter un élément dans un dictionnaire. C'est d'une simplicité enfantine. Pas besoin de faire appel à la méthode **append ()**.

Pour supprimer un élément, on utilise l'instruction **del**. En fait, on supprime la clé et par conséquent, la valeur qui lui est associée.

```
[code language= « python »]
capitales = {"France": "Paris", "Allemagne": "Berlin", "Islande": "Reykjavik"}
del capitales["Allemagne"]
print(capitales)
[/code]
```

Résultat : {'France': 'Paris', 'Islande': 'Reykjavik'}

## Supprimer tout le dictionnaire grâce à l'instruction **del**

Notez bien que l'instruction **del** permet également de supprimer tout le dictionnaire. Il suffit pour cela, de ne pas lui passer de clé entre crochets.

```
[code language= « python »]
capitales = {"France": "Paris", "Allemagne" : "Berlin", "Islande": "Reykjavik"}
del capitales
print(capitales)
[/code]
```

**NameError: name 'capitales' is not defined**

Ce message d'erreur nous confirme que le dictionnaire n'existe plus.

## Utiliser la fonction intégrée **len ()**

On peut également utiliser la fonction intégrée **len ()** pour connaître le nombre de paires clé-valeur contenues dans un dictionnaire.

```
[code language= « python »]
capitales = {"France": "Paris", "Allemagne" : "Berlin", "Islande": "Reykjavik"}
print(len(capitales))
[/code]
```

Résultat: 3

## Aperçu de quelques méthodes associées aux dictionnaires

- **dict.clear ()** supprime tous les éléments d'un dictionnaire. En clair, il le vide.

```
[code language= « python »]
capitales = {"France": "Paris", "Allemagne" : "Berlin", "Islande": "Reykjavik"}
capitales.clear() # Vide le dictionnaire
print(capitales)
[/code]
```

Résultat: {}

- **dict.items ()** retourne toutes les paires clé-valeur sous la forme d'une liste de tuples

```
[code language= « python »]
capitales = {"France": "Paris", "Allemagne" : "Berlin", "Islande": "Reykjavik"}
result = capitales.items()
print(result)
[/code]
```

Résultat: dict\_items([('Islande', 'Reykjavik'), ('France', 'Paris'), ('Allemagne', 'Berlin')])

- **dict.keys ()** retourne toutes les clés du dictionnaire sous la forme d'une liste

```
[code language= « python »]
capitales = {"France": "Paris", "Allemagne" : "Berlin", "Islande": "Reykjavik"}
result = capitales.keys()
print(result)
[/code]
```

Résultat: dict\_keys(['Islande', 'Allemagne', 'France'])

- **dict.values ()** retourne toutes les valeurs du dictionnaire sous la forme d'une liste

```
[code language= « python »]  
capitales = {"France": "Paris", "Allemagne" : "Berlin", "Islande": "Reykjavik"}  
result = capitales.values()  
print(result)
```

[/code]                      Résultat: dict\_values(['Berlin', 'Reykjavik', 'Paris'])

## Conclusion

Un dictionnaire est un ensemble non ordonné de paires **clé-valeur**. Chaque clé donne accès à la valeur qui lui est associée. Plus puissant qu'une liste, Un dictionnaire est idéal pour réaliser un test d'appartenance. En effet, comme il s'agit d'une implémentation de table de hachage et non pas d'une séquence, la vitesse d'exécution du test d'appartenance n'est pas liée au nombre d'éléments que le dictionnaire contient.

Exercices et corrigés [http://www.grappa.univ-lille3.fr/~coulom/Python/listes\\_prog.html](http://www.grappa.univ-lille3.fr/~coulom/Python/listes_prog.html)

# IMPORTER UN MODULE

Un module est un fichier Python qui contient un programme que l'on peut exécuter de manière indépendante ou bien qui regroupe des attributs que l'on peut charger dans un programme principal grâce à l'instruction **import**.

Pour faciliter leur maintenance, la plupart des programmes Python sont constitués d'un code principal sur lequel vient se greffer différents modules importés. Il s'agit soit de modules qui appartiennent à la [bibliothèque standard de Python](#) (par exemple le [module math](#)) soit de modules écrits par le programmeur lui-même. La [bibliothèque standard de Python](#) contient des dizaines de modules.

Amusons-nous à importer des attributs (c'est-à-dire des fonctions ou des variables) du [module math](#). Ce dernier contient par exemple la variable **pi** qui correspond à la valeur approchée [3.141592653589793](#).

Source et licence : <https://commons.wikimedia.org/wiki/File:Elster2.jpg> – Auteur : Stauss

Si vous voulez calculer le périmètre d'un cercle, vous en aurez besoin. Pour accéder à cette variable, il existe deux méthodes :

## 1<sup>ère</sup> méthode d'importation :

### Instruction **from math import pi**

```
[code language= »python «]  
#!/usr/bin/python3  
# -*- coding: utf8 -*-
```

```
from math import pi #Importation de la constante pi du module math
```

```
rayon = 3  
perimetre = 2*pi*3  
print (perimetre)  
[/code]
```

L'appel de la fonction **print ()** retourne le résultat [18.84955592153876](#)

Prendre soin d'importer la constante **pi**, Python m'aurait retourné cette exception :

**NameError: name 'pi' is not defined**

Comme je l'ai déjà précisé, il y a différentes manières d'importer un module et/ou les variables qu'il contient. Dans le programme qui calcule le périmètre, nous n'avons pas importé le module mais seulement la constante **pi** qui devient alors une variable de mon programme principal.

```
[code language= « python »] from math import pi[/code]
```

En écrivant `print (math)`, On obtient une erreur :

**NameError: name 'math' is not defined**

Tandis que l'on écrit `print (pi)`, on obtient ceci :

3.141592653589793

C'est donc bien la preuve que dans le programme principal, uniquement la variable **pi**.

Il est possible d'importer tous les attributs du module **math** en utilisant l'astérisque :

```
[code language= »python »] from math import *[/code]
```

## 2ème méthode d'importation : l'instruction **import math**

```
[code language= »python »]
```

```
#!/usr/bin/python3
```

```
# -*- coding: utf8 -*-
```

```
import math #importation du module math
```

```
rayon = 3
```

```
perimetre = 2*math.pi*3
```

```
print (perimetre)
```

```
[/code]
```

Dans cet exemple, importation du module **math** qui devient donc une nouvelle variable de notre programme principal mais la constante **pi** est accessible uniquement en utilisant cette syntaxe : **math.pi**

En langage Python où tout est objet, le point est un signe d'appartenance. **pi** est un attribut de l'objet **math**.

En écrivant `print (math)`, résultat :

<module 'math' (built-in)>

En écrivant `print (pi)`, Python retourne une exception :

**NameError: name 'pi' is not defined**

Alors qu'avec `print (math.pi)`, la variable **pi** qui est dans le module **math** et on bien le résultat escompté :

3.141592653589793

Le module **math** offre les fonctions et les constantes mathématiques usuelles, résumées dans ce tableau:



Python	mathématiques
<code>math.factorial(n)</code>	$n!$
<code>math.fabs(x)</code>	$ x $
<code>math.exp(x)</code>	$e^x$
<code>math.log(x)</code>	$\ln(x)$
<code>math.log10(x)</code>	$\log_{10}(x)$
<code>math.pow(x, y)</code>	$x^y$
<code>math.sqrt(x)</code>	$\sqrt{x}$
<code>math.sin(x)</code>	$\sin(x)$
<code>math.cos(x)</code>	$\cos(x)$
<code>math.tan(x)</code>	$\tan(x)$
<code>math.asin(x)</code>	$\arcsin(x)$
<code>math.acos(x)</code>	$\arccos(x)$
<code>math.atan(x)</code>	$\arctan(x)$
<code>math.pi</code>	$\pi$
<code>math.e</code>	$e$

## La fonction pow (power en anglais)

Prenons un autre exemple, en important la fonction **pow** du module **math**. Cette dernière permet d'élever un nombre  $x$  à la puissance  $y$ :

```
[code language= « python »]
#!/usr/bin/python3
# -*- coding: utf8 -*-
```

```
import math #importation du module math
nombre = 3
resultat = math.pow (nombre, 4)
print(resultat)
[/code]
```

Dans cet exemple, on élève le nombre trois à la puissance 4. Pour ce faire, passer deux paramètres à la fonction **pow** :

- la variable **nombre** (3)
- la puissance (4)

Le résultat obtenu est **81**.

## La fonction sqrt ()

**Sqrt ()** permet de calculer la racine carré du nombre passé en argument :

```
[code language= »python «]
import math #importation du module math
nombre = 36
resultat = math.sqrt(36)
print(resultat)
[/code]      Le résultat obtenu est 6.
```

## Utiliser `dir()` pour afficher tous les attributs d'un module

Pour afficher tous les attributs du module **math**, il suffit d'utiliser la fonction **dir** (bien sûr après avoir chargé le module **math**!).

```
[code language= »python »]
import math
print(dir(math))
[/code]
```

Résultat :

```
['doc', 'loader', 'name', 'package', 'spec', 'acos', 'acosh',
 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fm',
 'od', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'is',
 'inf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'loglp', 'log2', 'modf', 'nan',
 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>>
```

## Le module **webbrowser** (navigateur)

La bibliothèque standard Python contient des dizaines de modules. Outre le module **math**, on trouve également le module **webbrowser**. Ce dernier contient la fonction **open(url, new = 0, autoraise = true)**.

```
[code language= python »]
#!/usr/bin/python3
# -*- coding: utf8 -*-

import webbrowser
webbrowser.open("https://fr.wikipedia.org")
[/code]
```

Ce code provoque l'ouverture de la page wikipédia dans le navigateur défini par défaut. Pour ce faire, il faut passer l'url de la page en argument à la fonction **open()** sous la forme d'une chaîne de caractères.



# Importer un programme en tant que module dans une autre application

Un module peut aussi être un programme que l'on peut lancer en tant qu'application principale ou bien que l'on peut charger dans un autre programme plus important. Voici un exemple très concret :

Dans un programme **de choix xxx**, Il y a plusieurs onglets qui ouvrent notamment un agenda, un calendrier et un logiciel de post-it. Ces trois applications que l'on peut lancer de façon tout à fait indépendante ou que l'on peut inclure dans un autre programme. C'est le cas ici : importées dans l'application **de choix xxx** (ligne 7, 8 et 9).

```
[code language= « python »]
#!/usr/bin/python3
# -*- coding: utf8 -*-
from tkinter import* # Importation du module tkinter
from module_globs import Globales # Importation du module de redimensionnement
import os # Fonctions basiques du système d'exploitation
from PIL import Image, ImageTk # Module PIL (traitement des images importées)
from calendrier_perpetuel import Calendar
from agenda_perpetuel import Agenda_perpetuel
from module_notes import Notes
[/code]*
```

## L'instruction `if __name__ == « __main__ »:`

Pour déterminer si l'application « post-it » doit être lancée en tant que programme principal ou bien importée en tant que module dans un autre programme, il faut utiliser l'instruction:

```
if __name__ == « __main__ »:
```

Cette instruction se trouve à la fin du code du logiciel de post-it :

```
[code language= »python »]

# PROGRAMME PRINCIPAL:

if __name__ == "__main__":

    note = Notes () # Création de l'objet note par instanciation de la classe Notes.
    note.notes () # Appel de la méthode notes.

[/code]
```

## Conclusion

L'importation de modules est une opération très courante en Python. Elle permet de faire appel aux fonctions présentes dans la bibliothèque standard du langage qu'une communauté de développeurs ne cesse d'enrichir. Elle permet également d'importer des applications en tant que modules dans d'autres programmes.

# LES CLASSES

Une classe est un type permettant de regrouper dans la même structure : les informations (champs, propriétés, attributs) relatives à une entité ; les procédures et fonctions permettant de les manipuler (méthodes). Champs et méthodes constituent les membres de la classe.

## Remarques :

- La classe est un type structuré qui va plus loin que l'enregistrement (ce dernier n'intègre que les champs).
- Les champs d'une classe peuvent être de type quelconque. Ils peuvent faire référence à d'instances d'autres classes.

## Termes techniques :

- « Classe » est la structure.

Une classe est un ensemble incluant des variables ou *attributs* et des fonctions ou *méthodes*. Les attributs sont des variables accessibles depuis toute méthode de la classe où elles sont définies. En *python*, les classes sont des types modifiables.

- « Objet » est une instance de la classe (variable obtenue après instanciation).
- « Instanciation » correspond à la création d'un objet

Une instance d'une classe **C** désigne une variable de type **C**. Le terme instance ne s'applique qu'aux variables dont le type est une classe.

- « Méthode »

Les méthodes sont des fonctions qui sont associées de manière explicite à une classe. Elles ont comme particularité un accès privilégié aux données de la classe elle-même.

- L'objet est une référence (traité par le ramasse-miettes, destruction explicite inutile).

Ce n'est pas obligatoire, mais on a toujours intérêt à définir les classes dans des modules. On peut avoir plusieurs classes dans un module :

## Exemple : ModuleXXXXX.py

Python intègre des particularités –pour ne pas dire bizarreries –que l’on ne retrouve pas dans les langages objets populaires (C++, Java, C#). Ex. création à la volée d’une propriété sur un objet (instance de classe), possibilité de l’utiliser dans les méthodes (alors qu’elle n’apparaît nulle part dans la définition de la classe).

### Syntaxe S1 : Déclaration d’une classe

```
class nom_classe :  
    # corps de la classe  
    # ...
```

### Syntaxe S2 : Instanciation d’une classe

```
cl = nom_classe() « création de l’objet »
```

Il est tout de même possible de définir une instance de la classe `classe_vide` simplement par l’instruction suivante :

```
<<<
```

```
class classe_vide :  
    pass
```

```
cl = classe_vide()
```

```
>>>
```

Dans l’exemple précédent, la variable `cl` n’est pas de type `exemple_classe` mais de type `instance` comme le montre la ligne suivante :

```
<<<
```

```
class classe_vide :  
    pass
```

```
cl = classe_vide()
```

```
print (type(cl))    # affiche <type 'instance'>
```

```
<class 'pyquickhelper.sphinxext.sphinx_runpython_extension.run_python_script_14004852  
0084432.<locals>.classe_vide'>
```

Pour savoir si une variable est une instance d'une classe donnée, il faut utiliser la fonction `isinstance` :

<<<

```
class classe_vide:
    pass

cl = classe_vide()
print (type(cl))           # affiche <type 'instance'>
print (instance(cl, classe_vide)) # affiche True
```

>>>

```
<class 'pyquickhelper.sphinxext.sphinx_runpython_extension.run_python_script_14004852
1189560.<locals>.classe_vide'>

True
```

## Méthodes

Ces données ou *attributs* sont définis plus loin. Les méthodes sont en fait des fonctions pour lesquelles la liste des paramètres contient obligatoirement un paramètre explicite qui est l'instance de la classe à laquelle cette méthode est associée. Ce paramètre est le moyen d'accéder aux données de la classe.

### Syntaxe S3 : Déclaration d'une méthode

```
class nom_classe :
    def nom_methode (self, param_1, ..., param_n):
        # corps de la méthode...
```

A part le premier paramètre qui doit de préférence s'appeler `self`, la syntaxe de définition d'une méthode ressemble en tout point à celle d'une fonction. Le corps de la méthode est indenté par rapport à la déclaration de la méthode, elle-même indentée par rapport à la déclaration de la classe. Comme une fonction, une méthode suppose que les arguments qu'elle reçoit existe, y compris `self`. On écrit la méthode en supposant qu'un objet existe qu'on nomme `self`. L'appel à cette méthode obéit à la syntaxe qui suit :

### Syntaxe S4 : Appel d'une méthode

```
cl = nom_classe() # variable de type nom_classe
t = cl.nom_methode (valeur_1, ..., valeur_n)
```

L'appel d'une méthode nécessite tout d'abord la création d'une variable. Une fois cette variable créée, il suffit d'ajouter le symbole « `.` » pour exécuter la méthode. Le paramètre `self` est ici implicitement remplacé par `cl` lors de l'appel.

L'exemple suivant simule le tirage de nombres aléatoires à partir d'une suite définie par récurrence où et sont des entiers très grands. Cette suite n'est pas aléatoire mais son

comportement imite celui d'une suite aléatoire. Le terme `rand` est dans cet exemple contenu dans la variable globale `rand`.

```
rand = 42
```

```
class exemple_classe:
```

```
    def methode1(self, n):
```

```
        """simule la génération d'un nombre aléatoire
        compris entre 0 et n-1 inclus"""
```

```
        global rand
```

```
        rand = 397204094 * rand % 2147483647
```

```
        return int(rand % n)
```

```
nb = exemple_classe()
```

```
l1 = [nb.methode1(100) for i in range(0, 10)]
```

```
print(l1) # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

```
nb2 = exemple_classe()
```

```
l2 = [nb2.methode1(100) for i in range(0, 10)]
```

```
print(l2) # affiche [46, 42, 89, 66, 48, 12, 61, 84, 71, 41]
```

```
>>>
```

```
[19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

```
[46, 42, 89, 66, 48, 12, 61, 84, 71, 41]
```

Deux instances `nb` et `nb2` de la classe `exemple_classe` sont créées, chacune d'elles est utilisée pour générer aléatoirement dix nombres entiers compris entre 0 et 99 inclus. Les deux listes sont différentes puisque l'instance `nb2` utilise la variable globale `rand` précédemment modifiée par l'appel `nb.methode1(100)`.

Les méthodes sont des fonctions insérées à l'intérieur d'une classe. La syntaxe de la déclaration d'une méthode est identique à celle d'une fonction en tenant compte du premier paramètre qui doit impérativement être `self`. Les paramètres par défaut, l'ordre des paramètres, les nombres variables de paramètres présentés au paragraphe [Fonctions](#) sont des extensions tout autant applicables aux méthodes qu'aux fonctions.

# Attributs

## Définition D4 : attribut

Les attributs sont des variables qui sont associées de manière explicite à une classe. Les attributs de la classe se comportent comme des variables globales pour toutes les méthodes de cette classe.

Une classe permet en quelque sorte de regrouper ensemble des informations liées. Elles n'ont de sens qu'ensemble et les méthodes manipulent ces données liées. C'est le cas pour un segment qui est toujours défini par ces deux extrémités qui ne vont pas l'une sans l'autre.

## Syntaxe S5 : Déclaration d'un attribut

```
class nom_classe :  
    def nom_methode (self, param_1, ..., param_n) :  
        self.nom_attribut = valeur
```

Le paramètre **self** n'est pas un mot-clé même si le premier paramètre est le plus souvent appelé **self**. Il désigne l'instance de la classe sur laquelle va s'appliquer la méthode. La déclaration d'une méthode inclut toujours un paramètre **self** de sorte que **self.nom\_attribut** désigne un attribut de la classe. **nom\_attribut** seul désignerait une variable locale sans aucun rapport avec un attribut portant le même nom. Les attributs peuvent être déclarés à l'intérieur de n'importe quelle méthode, voire à l'extérieur de la classe elle-même.

L'endroit où est déclaré un attribut a peu d'importance pourvu qu'il le soit avant sa première utilisation. Dans l'exemple qui suit, la méthode **methode1** utilise l'attribut **rnd** sans qu'il ait été créé.

<<<

```
class exemple_classe:  
    def methode1(self, n):  
        """simule la génération d'un nombre aléatoire  
        compris entre 0 et n-1 inclus"""  
        self.rnd = 397204094 * self.rnd % 2147483647  
        return int (self.rnd % n)
```

```
nb = exemple_classe ()  
li = [nb.methode1 (100) for i in range (0, 10)]  
print(li)
```

[runpythonerror]

Traceback (most recent call last):



```
exec (obj, globs, loc)
File "", line 13, in <module>
File "", line 11, in run_python_script_140048520101104
File "", line 11, in <listcomp>
File "", line 7, in methode1
AttributeError: 'exemple_classe' object has no attribute 'rnd'
```

Cet exemple déclenche donc une erreur (ou exception) signifiant que l'attribut `rnd` n'a pas été créé.

Pour remédier à ce problème, il existe plusieurs endroits où il est possible de créer l'attribut `rnd`. Il est possible de créer l'attribut à l'intérieur de la méthode `methode1`. Mais le programme n'a plus le même sens puisqu'à chaque appel de la méthode `methode1`, l'attribut `rnd` reçoit la valeur 42. La liste de nombres aléatoires contient dix fois la même valeur.

<<<

```
class exemple_classe:
    def methode1(self, n):
        """simule la génération d'un nombre aléatoire
        compris entre 0 et n-1 inclus"""
        self.rnd = 42 # déclaration à l'intérieur de la méthode,
        # doit être précédé du mot-clé self
        self.rnd = 397204094 * self.rnd % 2147483647
        return int(self.rnd % n)

nb = exemple_classe ()
li = [nb.methode1 (100) for i in range(0, 10)]
print(li) # affiche [19, 19, 19, 19, 19, 19, 19, 19, 19, 19]
```

>>>

```
[19, 19, 19, 19, 19, 19, 19, 19, 19, 19]
```

Il est possible de créer l'attribut `rnd` à l'extérieur de la classe. Cette écriture devrait toutefois être évitée puisque la méthode `methode1` ne peut pas être appelée sans que l'attribut `rnd` ait été ajouté.

<<<

```
class exemple_classe:
    def methode1(self, n):
        """simule la génération d'un nombre aléatoire
```

```

        compris entre 0 et n-1 inclus""")
self.rnd = 397204094 * self.rnd % 2147483647
return int (self.rnd % n)

nb = exemple_classe()
nb.rnd = 42          # déclaration à l'extérieur de la classe,
# indispensable pour utiliser la méthode methode1
li = [nb.methode1 (100) for i in range(0, 10)]
print(li) # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]

>>>

```

```
[19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

Ceux qui découvrent la programmation se posent toujours la question de l'utilité de ce nouveau concept qui ne permet pas de faire des choses différentes, tout au plus de les faire mieux. La finalité des classes apparaît avec le concept d'[Héritage](#). L'article illustre une façon de passer progressivement des fonctions aux classes de fonctions.

## Constructeur

L'endroit le plus approprié pour déclarer un attribut est à l'intérieur d'une méthode appelée le *constructeur*. S'il est défini, il est implicitement exécuté lors de la création de chaque instance. Le constructeur d'une classe se présente comme une méthode et suit la même syntaxe à ceci près que son nom est imposé : `__init__`. Hormis le premier paramètre, invariablement `self`, il n'existe pas de contrainte concernant la liste des paramètres excepté que le constructeur ne doit pas retourner de résultat.

### Syntaxe S6 : Déclaration d'un constructeur

```

class nom_classe :
    def __init__(self, param_1, ..., param_n):
        # code du constructeur

```

`nom_classe` est une classe, `__init__` est son constructeur, sa syntaxe est la même que celle d'une méthode sauf que le constructeur ne peut employer l'instruction `return`. La modification des paramètres du constructeur implique également la modification de la syntaxe de création d'une instance de cette classe.

### Syntaxe S7 : Appel d'un constructeur

```
x = nom_classe (valeur_1,...,valeur_n)
```

`nom_classe` est une classe, `valeur_1` à `valeur_n` sont les valeurs associées aux paramètres `param_1` à `param_n` du constructeur.

L'exemple suivant montre deux classes pour lesquelles un constructeur a été défini. La première n'ajoute aucun paramètre, la création d'une instance ne nécessite pas de paramètre

supplémentaire. La seconde classe ajoute deux paramètres **a** et **b**. Lors de la création d'une instance de la classe **classe2**, il faut ajouter deux valeurs.

<<<

```
class classe1:
    def __init__(self):
        # pas de paramètre supplémentaire
        print ("constructeur de la classe classe1")
        self.n = 1 # ajout de l'attribut n

x = classe1() # affiche constructeur de la classe classe1
print (x.n)   # affiche 1

class classe2:
    def __init__(self, a, b):
        # deux paramètres supplémentaires
        print ("constructeur de la classe classe2")
        self.n = (a + b) / 2 # ajout de l'attribut n

x = classe2(5, 9) # affiche constructeur de la classe classe2
print (x.n)      # affiche 7
```

>>>

```
constructeur de la classe classe1
1
constructeur de la classe classe2
7.0
```

Le constructeur autorise autant de paramètres qu'on souhaite lors de la création d'une instance et celle-ci suit la même syntaxe qu'une fonction. La création d'une instance pourrait être considérée comme l'appel à une fonction à ceci près que le type du résultat est une instance de classe.

En utilisant un constructeur, l'exemple du paragraphe précédent simulant une suite de variable aléatoire permet d'obtenir une classe autonome qui ne fait pas appel à une variable globale ni à une déclaration d'attribut extérieur à la classe.

```
class exemple_classe:
```

```

def __init__(self): # constructeur
    self.rnd = 42    # on crée l'attribut rnd, identique pour chaque instance
    # --> les suites générées auront toutes le même début

def methode1(self, n):
    self.rnd = 397204094 * self.rnd % 2147483647
    return int(self.rnd % n)

```

```

nb = exemple_classe()
l1 = [nb.methode1(100) for i in range(0, 10)]
print(l1) # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]

```

```

nb2 = exemple_classe()
l2 = [nb2.methode1(100) for i in range(0, 10)]
print(l2) # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]

```

```
>>>
```

```

[19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
[19, 46, 26, 88, 44, 56, 56, 26, 0, 8]

```

De la même manière qu'il existe un constructeur exécuté à chaque création d'instance, il existe un destructeur exécuté à chaque destruction d'instance. Il suffit pour cela de redéfinir la méthode `__del__`. A l'inverse d'autres langages comme le C++, cet opérateur est peu utilisé car le *python* nettoie automatiquement les objets qui ne sont plus utilisés ou plus référencés par une variable.