Programmer en python

1. Fonctions de base et variables

print

Le premier programme à connaître dans n'importe quel langage de programmation consiste à afficher un simple message sur le terminal (l'interface où le résultat du programme est affiché).
En python, on peut le faire grâce à la commande *print* suivie du message encadré de parenthèse et de guillemets comme le montre le code suivant :

```
1. print("hello!")
```

```
> hello!
```

Ici le programme affiche en sortie (ici dénoté par le second bloc et suivi du signe '>') la chaîne de caractères "hello!" (tout ce qui est affiché entre les guillemets).

Les variables

La chaîne de caractères entre guillemets représente une valeur qui est stockée quelque part dans la mémoire de l'ordinateur. Dans le programme précédent, je n'ai aucun moyen d'accéder à l'emplacement dans la mémoire de cette valeur "hello!". Les **variables** permettent de donner un **nom** à cette valeur pour pouvoir ensuite la modifier, la réutiliser et la combiner avec d'autres valeurs.

Voici le même programme, cette fois en stockant d'abord la valeur "hello!" dans une variable que j'appelle *mon_texte* puis en l'utilisant dans print à la place de la valeur :

```
1. mon_texte = "hello!"
2. print(mon_texte)
```

```
> hello!
```

Ici le signe '=' ne représente pas une égalité mais une **déclaration** ou **affectation**. On place ce qui est à **droite** du '=' dans la variable dont le nom est écrit à **gauche**. Contrairement aux chaînes de caractères entre guillemets qui représentent des valeurs à être affichées et pouvant contenir tous les caractères standards (minuscules, majuscules, chiffres, espaces, ponctuation, retours à la ligne, etc), les noms de variables ne peuvent pas contenir d'espaces, ni de caractères spéciaux à part le _. Un nom de variable ne peut pas commencer par un chiffre. Notez que le nom du variable est entièrement interne au programme. Choisir un nom différent, tant qu'il est valide, ne change absolument pas ce qui est affiché par le programme.

Voici quelques exemples corrects de noms de variables :

```
1. Montexte = "hello!"
2. montexte1 = "hello!"
3. mon_Texte = "hello!"
```

Et voici quelques exemples incorrects de noms de variables qui ne sont pas acceptés par Python et affichent une erreur :

```
1. mon texte = "hello!"
2. 123montexte = "hello!"
3. mon.texte = "hello!"
```

Grâce aux variables, je peux réutiliser la même valeur plusieurs fois :

```
1. mon_texte="hello"
2. print(mon_texte)
3. print(mon_texte)
4. print(mon_texte)
```

```
> hello
  hello
  hello
```

Je peux également combiner cette valeur plusieurs fois avec l'opérateur '+' :

```
1. mon_texte="hello"
2. print(mon_texte + mon_texte + mon_texte)
```

```
> hellohellohello
```

Notez qu'une chaîne de caractères compte également les espaces. Dans l'exemple du dessus, les 3 "hello" sont collés, mais je peux combiner ma variable avec d'autres chaînes de caractères qui eux sont définis juste entre guillemets, sans variable :

```
1. mon_texte="hello"
2. print(mon_texte + " " + mon_texte + " " + mon_texte)
```

```
> hello hello hello
```

Je pourrais également mettre plein d'espaces plutôt qu'un seul entre les guillemets pour espacer encore plus mes trois "hello", ou je peux définir une deuxième variable par exemple :

```
1. mon_texte="hello"
2. plusieurs_espaces="        "
3. print(mon_texte + plusieurs_espaces + mon_texte +
   plusieurs_espaces + mon_texte)
```

```
> hello        hello        hello
```

Nous reviendrons sur les variables et d'autres opérations, mais d'abord passons à une autre commande.

input

La commande *input* est un peu "l'inverse" de *print*. Elle permet de prendre ce que l'utilisateur tape sur le clavier en entrée et de le stocker dans une variable. De plus, elle permet en option d'afficher un texte qui donne un indice à l'utilisateur de ce qu'il doit taper.
Le programme suivant affiche le nom de l'utilisateur. A la ligne 1, on déclare une nouvelle variable appelée *nom* et on lui donne comme valeur le résultat de la fonction input, c'est-à-dire ce que l'utilisateur tape sur le clavier.
Désormais notre variable *nom* contient une valeur qu'on ne connaît pas à l'avance car elle dépend de ce que l'utilisateur a tapé. On sait simplement qu'il s'agit d'une chaîne de caractères.
A la ligne 2, on utilise la fonction print pour afficher la chaîne de caractères "Bonjour " (remarquez bien l'espace après "Bonjour") suivie de la chaîne de caractères contenue dans la variable *nom*.

```
1. nom = input("Votre nom?")
2. print("Bonjour " + nom)
```

Le programme va d'abord bloquer son exécution après avoir affiché "Votre nom?" et attendre que l'utilisateur tape quelque chose. Ici admettons qu'il tape "John". C'est donc cette valeur qui sera stockée dans notre variable *nom* et on affichera "Bonjour " + "John" ce qui donne "Bonjour John".

```
> Votre nom? ______
> Votre nom? John
> Bonjour John
```

Exercice 1.1

Que fait le programme suivant ? Quelle est sa différence avec le programme précédent?

```
1. nom = input("")
2. print("Bonjour " + nom)
```

Solution 1.1

La chaîne de caractères entre guillemets que l'on écrit entre parenthèses à côté de input représente une sorte d'indice qui sera affiché pour indiquer à l'utilisateur ce qu'il doit taper. Mais on peut écrire n'importe quoi, même une chaîne de caractères vide comme c'est le cas ici. Le reste du programme est identique, on affiche donc d'abord **rien** (une chaîne de caractères vide) et on attend que l'utilisateur tape quelque chose :

```
> _____
> 123
> Bonjour 123
```

L'utilisateur ne voit donc rien qui lui indique ce qu'il doit taper. Peut-être qu'il tape son nom comme avant ("John"), ou peut-être, comme dans cet exemple, il tape "123". Pour le reste du programme, les deux sont des chaînes de caractères valides. La chaîne de caractères "123" est traitée exactement de la même manière même si ce n'est pas un prénom : stockée dans la variable *nom* à la ligne 1 et utilisée à la ligne 2 pour afficher "Bonjour " + "123".

Exercice 1.2

Écrivez un programme qui demande **successivement** le prénom, puis le nom de famille de l'utilisateur. Il devra ensuite afficher "Bonjour" suivi du prénom, puis du nom de famille, puis d'un "!". Il devra y avoir des espaces entre chaque mot.

Solution 1.2

Commençons par le prénom. Il nous faut stocker ce que l'utilisateur tape comme prénom dans une variable grâce à la fonction input :

```
1. prenom = input("Quel est votre prénom ?")
```

Ce programme n'affiche rien pour l'instant car on ne fait rien avec cette variable *prenom*. Essayons de l'afficher avec la fonction *print* :

```
1. prenom = input("Quel est votre prénom ?")
2. print(prenom)
```

```
> Quel est votre prénom ? _____
> Quel est votre prénom ? Fanny
> Fanny
```

Voilà nous avons validé que notre variable prenom contient effectivement le prénom tapé par l'utilisateur car à la ligne 3 nous l'affichons. Maintenant faisons à la suite la même chose pour le nom de famille. Nous aurons donc besoin d'une 2e variable :

```
1. prenom = input("Quel est votre prénom ?")
2. print(prenom)
3. nom_famille = input("Quel est votre nom de famille ?")
4. print(nom_famille)
```

```
> Quel est votre prénom ? _____
> Quel est votre prénom ? Harry
> Harry
> Quel est votre nom de famille ? ______
> Quel est votre nom de famille ? Potter
> Potter
```

Les lignes 3 et 4 sont quasiment les mêmes que les lignes 1 et 2. Seuls les indices donnés à l'utilisateur et les noms de variables changent. Le programme bloque donc une première fois en attendant que l'utilisateur tape son prénom, l'affiche, puis bloque à nouveau en attendant que l'utilisateur tape son nom de famille, puis l'affiche également.

Maintenant que nous avons nos deux variables *prenom* et *nom_famille* qui contiennent ce que l'utilisateur a tapé, nous pouvons enlever les affichages *print* aux lignes 2 et 4 et rajouter un seul *print* qui combine "Bonjour" et ces deux variables :

```
1. prenom = input("Quel est votre prénom ?")
2. nom_famille = input("Quel est votre nom de famille ?")
3. print("Bonjour" + prenom + nom_famille + "!")
```

```
> Quel est votre prénom ? Albert
> Quel est votre nom de famille ? Einstein
> BonjourAlbertEinstein!
```

Les 3 mots s'affichent mais collés.. Rappelez vous que l'opérateur "+" met ensemble **exactement** ce qui est contenu dans les chaînes de caractères. On doit donc rajouter des espaces à la ligne 3, sans que l'utilisateur entre manuellement des espaces là où il doit juste donner son prénom et son nom. Voilà notre solution finale :

```
1. prenom = input("Quel est votre prénom ?")
2. nom_famille = input("Quel est votre nom de famille ?")
3. print("Bonjour " + prenom + " " + nom_famille + "!")
```

```
> Quel est votre prénom ? Albert
> Quel est votre nom de famille ? Einstein
> Bonjour Albert Einstein!
```

Types de variables

Pour l'instant nous avons vu qu'une variable contient une valeur et cette valeur jusqu'à présent est toujours une chaîne de caractères notée entre guillemets:

```
1. ma_variable = "bla bla bla"
```

Mais en réalité, une variable peut contenir d'autres choses comme des nombres, entiers ou à virgule, positifs ou négatifs, ou encore d'autres choses qui ne sont ni des chaînes de caractères, ni des nombres. On introduit donc la notion de **type de variable**. Une variable stocke un certain de type de valeurs et la plupart du temps ne peut être combinée qu'avec d'autres variables du même type.
Par exemple, les variables aux lignes 1 et 2 contiennent des chaînes de caractères. Ce type est appelé *string* et abrégé **str** en Python. Aux lignes 3 et 4, les variables contiennent des nombres entiers. Ce type est appelé *integer* et abrégé **int** en Python.

```
1. ma_chaine1 = "salut..."
2. ma_chaine2 = "45"
3. mon_nombre_entier1 = 3
4. mon_nombre_entier2 = 45
```

Remarquez que les deux variables *ma_chaine2* et *mon_nombre_entier2* ne contiennent **PAS** la même valeur. La première est une variable de type **str** qui contient la chaîne de caractères "45" notée entre guillemets, alors que la deuxième est une variable de type **int** qui contient le nombre 45.

La différence ne se remarquera pas en affichant ces variables avec print :

```
1. une_chaine = "356"
2. un_nombre_entier = 356
3. print(une_chaine)
4. print(un_nombre_entier)
```

```
> 356
> 356
```

Par contre, l'opérateur "+" que nous avons vu auparavant fonctionne différemment entre variables de type **str** ou **int**. Dans le cas **str**, il rajoute la 2e chaîne de caractères à la première, alors que dans le cas de **int**, il fait une addition arithmétique simple :

```
1. une_chaine = "356"
2. un_nombre_entier = 356
3. print(une_chaine + une_chaine)
4. print(un_nombre_entier + un_nombre_entier)
```

```
> 356356
> 712
```

Ici le programme a d'abord ajouté la chaîne "356" à la chaîne "356" pour constituer une nouvelle chaîne de 6 caractères : "356356". Puis, le programme a additionné les nombres 356 + 356 = 712.

Par contre, si on essaie d'utiliser l'opérateur "+" entre une **str** et **int** :

```
1. une_chaine = "356"
2. un_nombre_entier = 356
3. print(une_chaine + un_nombre_entier)
```

```
> erreur à la ligne 3!
```

Le programme donne une erreur car cela n'a pas de sens de combiner ou additionner des valeurs qui n'ont pas du tout le même type.

Opérations sur des variables int

Pour ce qui est des variables str, nous avons un opérateur “+” qui permet de combiner plusieurs. Cet opérateur est également défini pour les int afin d’effectuer des additions. Mais pour les int, il existe bien d’autres opérateurs qui effectuent différents calculs. Voici la liste de ceux à connaître pour l’instant :

- + : addition
- - : soustraction
- * : multiplication
- // : division entière (arrondie à l’entier le plus bas)
- % : modulo. Il s’agit du reste de la division entière.
- ** : puissance

Les **int** peuvent également être affichés avec la fonction *print*. Le programme suivant utilise tous les opérateurs ci-dessus pour afficher certains résultats à titre d’exemples :

```
1.  x = 3
2.  y = 5
3.  print(x + y)
4.  print(x + 2)
5.  print(y - 10)
6.  print(y - x - x)
7.  print(x * y)
8.  print(x ** y)
9.  print(x // y)
10. print(30 // 12)
11. print(30 % 12)
12. print(x % y)
```

```
> 8
> 5
> -5
> -1
> 15
> 243
> 0
> 2
> 6
> 3
```

La ligne x // y calcule la division entière de 3 par 5. Comme ce résultat vaut 0.6, arrondi en dessous, la division entière est 0. Une autre manière de voir ce résultat est qu'on peut mettre 0 fois 5 dans 3.
Par contre 30 // 12 vaut 2, car on peut mettre 2 fois 12 dans 30. 30 % 12 calcule le reste de cette division entière : 30 = 2*12 + 6. Le reste de la division est 6 ici.
De même, x %y calcule le reste de la division entière de 3 par 5. Comme on met 0 fois 5 dans 3, il reste toujours 3.

Vous remarquerez qu'il n'y pas de division à virgule. Nous les verrons plus tard car pour l'instant nous travaillons avec le type de variable **int** qui représente des nombres entiers uniquement.

Conversions de types

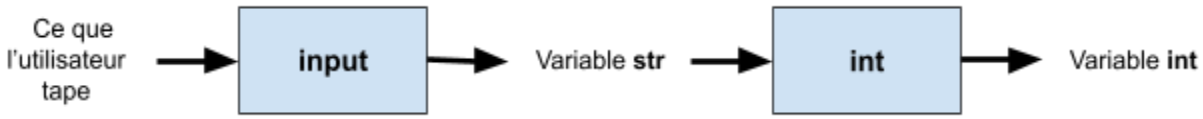
Nous aimerions maintenant utiliser nos nouveaux opérateurs arithmétiques sur des int pour créer des programmes interactifs qui utilisent ce que tape l'utilisateur grâce à la fonction input. Par exemple un programme qui prend 2 nombres en entrées et affiche leur somme :

```
1. n1 = input("Donnez un premier nombre: ")
2. n2 = input("Donnez un deuxième nombre: ")
3. somme = n1 + n2
4. print(somme)
```

```
> Donnez un premier nombre: 12
> Donnez un deuxième nombre: 90
> 1290
```

Malheureusement le résultat est faux. Pourquoi ? Comme nous l'avons vu plus haut, la fonction input retourne la **chaîne de caractères** que tape l'utilisateur et la stocke dans une variable qui a donc le type **str**. Dans cet exemple, *n1* et *n2* sont deux variables de type **str** qui contiennent les valeurs "12" et "90" entre guillemets et non pas les nombres 12 et 90. Donc l'opérateur "+" à la ligne 3 est l'opérateur entre **str** qui combine des chaînes de caractères, pas l'opérateur entre **int** qui effectue une addition. Il va donc ajouter la chaîne "90" après la chaîne "12" pour obtenir une nouvelle chaîne de 4 caractères : "1290" et non pas un nombre entier **int**.

Il nous faut donc **convertir** le résultat donné par *input* sous la forme d'une variable **str** pour le transformer en une variable **int**. Pour cela nous allons utiliser une nouvelle fonction qui s'appelle aussi *int*. Schématiquement cela peut se représenter ainsi :



Dans le programme suivant, la variable n1 a le type **str** car elle prend la valeur donnée par la fonction *input*. La variable n2 a le type **int** car elle prend la valeur donnée par la fonction *int* :

```
1. n1 = input("Donnez un nombre: ")
2. n2 = int(n1)
```

On n'est pas obligés de stocker le résultat de la fonction *input* dans une variable intermédiaire (ici *n1*) avant de passer cette variable à la fonction *int*. On peut directement mettre cette valeur dans la fonction *int*. Le programme suivant est entièrement équivalent au programme ci-dessus :

```
1. n2 = int(input("Donnez un nombre: "))
```

Si cela vous paraît complexe c'est normal. A ce stade vous n'avez pas besoin de tout comprendre entièrement ce qui se passe, cela viendra au fur et à mesure, surtout lorsque nous verrons plus en détails les fonctions. Pour l'instant, vous pouvez suivre cette règle simple :

- Si ce que l'utilisateur doit taper est une chaîne de caractères, j'utilise la fonction *input* comme vue initialement.
- Si ce que l'utilisateur doit taper est un nombre, "j'entoure" la fonction *input* de parenthèse et j'écris *int* juste devant. (Comme dans l'exemple ci-dessus)

En suivant cette règle, nous pouvons reprendre le programme incorrect que nous avions plus haut et le modifier ainsi :

```
1. n1 = int(input("Donnez un premier nombre: "))
2. n2 = int(input("Donnez un deuxième nombre: "))
3. somme = n1 + n2
4. print(somme)
```

```
> Donnez un premier nombre: 12
> Donnez un deuxième nombre: 90
> 102
```

Cette fois le résultat est correct ! Cette fois les deux variables *n1* et *n2* ont le type **int**. Elles ont le type **int** car cette fois, en suivant la règle simple énoncée plus haut, nous avons placé la fonction *int* avec des parenthèses autour de la fonction *input* aux lignes 1 et 2.
Cela a pour conséquence qu'à la ligne 3, l'opérateur "+" s'effectue sur deux variables **int** et calcule donc bien l'addition de deux nombres entiers, ce qui est affiché grâce à la ligne 4.

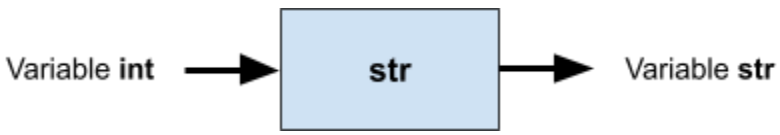
Essayons maintenant de ne pas juste afficher le résultat, mais de mettre une phrase pour décrire ce résultat, dans la fonction *print* :

```
1. n1 = int(input("Donnez un premier nombre: "))
2. n2 = int(input("Donnez un deuxième nombre: "))
3. somme = n1 + n2
4. print("La somme des deux nombres est : " + somme)
```

```
> erreur à la ligne 4!
```

Le programme ne fonctionne plus.. A la ligne 4, dans la fonction print, on utilise l'opérateur '+' entre la chaîne de caractères "La somme des deux nombres est : " (donc de type **str**) et la variable *somme* qui est de type **int**. Comme on l'a vu plus haut, on ne peut pas utiliser l'opérateur '+' entre des variables de types différents **str** et **int** car son comportement n'est pas défini. Ici on veut afficher un message, donc une **str**. Cela veut dire que le problème est la variable *somme* qui devrait être un **str** pas un **int**. On va donc devoir de nouveau **convertir** le type d'une variable mais cette fois dans l'autre sens, de **int** vers **str**.

Pour ce faire, il existe une autre fonction de conversion appelée *str*. Schématiquement on peut la représenter ainsi :



Le programme suivant convertit une variable de type **int** (*n1*) en une variable de type **str** (*n2*) :

```
1. n1 = 45
2. n2 = str(n1)
3. print(n1 + n1)
4. print(n2 + n2)
```

```
> 90
> 4545
```

Comme auparavant, lorsque à la ligne 3 on utilise l'opérateur '+' sur la variable *n1* qui est de type **int**, on calcule une addition entre 45 et 45 ce qui donne 90. A la ligne 2, on utilise la fonction *str* pour convertir le nombre 45 en la chaîne de caractères "45" qui est stockée dans *n2*. C'est pourquoi la ligne 4 effectue la combinaison de deux chaînes "45" et "45" ce qui donne la chaîne "4545".

On peut donc corriger notre programme précédent qui donnait une erreur, en utilisant la fonction str pour convertir la variable somme avant de l'utiliser dans le print :

```
1. n1 = int(input("Donnez un premier nombre: "))
2. n2 = int(input("Donnez un deuxième nombre: "))
3. somme = n1 + n2
4. print("La somme des deux nombres est : " + str(somme))
```

```
> Donnez un premier nombre: 20
> Donnez un deuxième nombre: 45
> La somme des deux nombres est : 65
```

Le programme fonctionne ! De nouveau, cela peut paraître complexe et deviendra plus clair avec la pratique. On peut de nouveau utiliser une règle simple pour le moment sans devoir tout comprendre :

- Si j'utilise des '+' dans mon *print* avec des variables **str**, je les laisse tel quel.
- Si j'utilise des '+' dans mon *print* avec des variables **int**, je les entoure de parenthèses et écrit *str* devant.

Exercice 1.3

Écrivez un programme qui demande à l'utilisateur d'entrer un nombre entier, calcule le triple de ce nombre et l'affiche sous la forme "Le triple est : " suivi du nombre. Par exemple, si l'utilisateur entre le nombre 12, le programme devra afficher "Le triple est : 36"

Solution 1.3

On commence par utiliser la fonction input pour stocker le nombre que l'utilisateur tape dans une variable :

```
1. n = input("Entrez un nombre : ")
```

Sauf qu'il s'agit d'un nombre et la fonction *input* retourne une valeur de type **str**. Donc notre variable *n* est de type **str** et contient une chaîne de caractères par exemple "23". Donc comme on l'a vu, on "entoure" le *input* de parenthèses et on rajoute *int* devant pour convertir de **str** en **int** afin que notre variable contienne plutôt le nombre 23 par exemple :

```
1. n = int(input("Entrez un nombre : "))
```

Maintenant que n est une variable de type int qui contient le nombre tapé par l'utilisateur, on peut l'utiliser pour calculer le triple :

```
1. n = int(input("Entrez un nombre : "))
2. triple = n * 3
```

Notre nouvelle variable triple contient le résultat, on peut maintenant l'afficher avec un

message à l'aide de la fonction print :

```
n = int(input("Entrez un nombre : "))
triple = n * 3
print("Le triple est : " + triple)
```

Sauf que triple est une variable de type int. Donc comme vu plus haut, dans la fonction print, on doit entourer la variable triple avec des parenthèses et écrire str devant :

```
1. n = int(input("Entrez un nombre : "))
2. triple = n * 3
3. print("Le triple est : " + str(triple))
```

Voilà notre solution finale et un exemple de l'exécution de ce programme :

```
> Entrez un nombre : 15
> Le triple est : 45
```

Les variables float

Maintenant que nous avons vu deux types de variables, **str** et **int**, et comment convertir de l'un vers l'autre, nous introduisons un 3e type : **float**. Les variables de type **float** représentent des nombres à virgule. Leur différence de comportement avec les **int** est assez subtile.

Voici un simple programme d'addition de nombres à virgules :

```
1. x1 = 5.432
2. x2 = 7.1
3. print(x1 + x2)
```

```
> 12.532
```

Les opérations arithmétiques valides entre variables de type **float** sont les suivantes:

- + pour l'addition
- - pour la soustraction
- * pour la multiplication
- / pour la division (à virgule)
- ** pour les puissances

Les opérations // (division entière) et % (modulo) pour les **int** que nous avons vu plus haut sont en théorie applicables sur des variables de type **float**, mais cela se fait peu car les questions de nombres entiers et de reste entier ont peu de sens dans le contexte de nombres à virgule.

A noter que la division à virgule avec le symbole / est utilisable avec des **int**, simplement que le résultat sera une variable de type **float**.

Pour ce qui est des conversions de type, on utilise toujours la même fonction *str*() pour convertir d'une variable **float** en variable **str**. Par contre, pour convertir de **str** en **float**, on utilise la fonction *float*() :

```
1. p1 = float(input("Prix d'un repas ?"))
2. p2 = float(input("Prix d'une boisson ?"))
3. somme = p1 + p2
4. print("Prix total : " + str(somme))
```

```
> Prix d'un repas ? 11.50
> Prix d'une boisson ? 3.80
> Prix total : 15.30
```

2. Les conditions

Les variables booléennes

Nous introduisons maintenant un 4e type de variable en plus des variables **str**, **int** et **float** : les variables booléennes. Ces variables peuvent prendre uniquement 2 valeurs : vrai ou faux. Ces variables sont dites de type **booléen**. En Python on le note **bool**. Les deux seules valeurs acceptées pour des variables Python de type **bool** sont **True** et **False** (sans guillemets et avec une majuscule):

```
1. a = True
2. b = False
3. print(a)
4. print(b)
```

```
> True
> False
```

Grâce au symbole *"<"*, nous pouvons comparer deux variables de type **int** ou **float**. L'expression *3 < 5* est une valeur **bool** qui peut être utilisée tel quel ou stockée dans une variable de type bool. Si le premier nombre est plus petit que le deuxième, la valeur de l'expression est **True**, sinon elle est **False**, comme le montre le programme suivant:

```
1. ma_variable_bool = 3 < 5
2. print(ma_variable_bool)
3. print(10 < 7)
```

```
> True
> False
```

Le programme suivant demande son âge à l'utilisateur, le compare et stocke le résultat dans une variable booléenne pour ensuite l'afficher :

```
1. a = int(input("âge ? "))
2. b = a < 18
3. print(b)
```

```
> âge ? 16
> True
```

```
> âge ? 21
> False
```

Pour pouvoir afficher ces variables **bool** en les combinant avec des chaînes de caractères, tout comme pour les variables de type **int** ou **float**, il faudra d'abord les convertir en **str** utilisant la fonction str():

```
1. a = int(input("âge ? "))
2. b = a < 18
3. print("Le résultat est : " + str(b))
```

Les comparaisons de valeurs

Dans les exemples précédents, nous avons utilisé la comparaison " strictement plus petit que" notée "<". De manière générale, l'expression a < b vérifie si a est strictement plus petit que b. Les valeurs a et b peuvent être des nombres ou des variables contenant des nombres. Il existe d'autres comparateurs de nombres:

- 5 < a : 5 strictement plus petit que la valeur de la variable a
- 3.23 <= b : 3.23 plus petit ou égal à la valeur de la variable b
- c > 10 : c strictement plus grand que 10
- d >= e : d plus grand ou égal à e
- f == 90 : f égal à 90. Remarquez le double '='.
- g != 100 : g n'est pas égal à 100

Il est également possible de comparer des chaînes de caractères et des variables **str**. Dans ce cas là, seuls les comparateurs “==” et “!=” ont du sens :

```
1. texte = input("Qui a inventé le langage Python ? ")
2. print(texte == "Guido van Rossum")
```

```
1. texte = input("Qui n’a PAS inventé le langage Python ? ")
2. print(texte != "Guido van Rossum")
```

if/else

Maintenant que nous avons vu les variables booléennes, nous pouvons les utiliser pour exécuter un certain code seulement si une condition est remplie ou un autre code si la condition n’est pas remplie grâce aux mots clés **if** et **else**:

```
1. a = int(input("Votre âge ? "))
2. if (a < 18):
3.   print("Vous êtes mineur.")
4. else:
5.   print("Vous êtes majeur.")
```

```
> Votre âge ? 16
> Vous êtes mineur.
```

```
> Votre âge ? 19
> Vous êtes majeur.
```

L’exécution du programme est différente selon si l’utilisateur entre un nombre plus petit que 18 ou pas. Après le **if**, on trouve toujours entre parenthèses une expression booléenne appelée **condition** qui est donc toujours soit **True** soit **False**. Remarquez les “:” obligatoires après le **if** et **else**, ainsi que les espaces/tabs aux lignes suivantes. On

appelle cela l'indentation. Python l'utilise pour savoir quel code sera exécuté seulement si une condition est remplie. Le **else** qui veut dire "sinon" est optionnel, je peux avoir le code suivant :

```
1. a = int(input("Votre âge ? "))
2. if (a < 18):
3.   print("Vous êtes mineur.")
4.   print("Vous serez majeur dans " + str(18-a) + " ans.")
5. print("Programme terminé.")
```

```
> Votre âge ? 16
> Vous êtes mineur.
> Vous serez majeur dans 2 ans.
> Programme terminé
```

```
> Votre âge ? 90
> Programme terminé
```

Remarquez la différence entre les lignes 3,4 et la ligne 5. Les lignes 3 et 4 s'exécutent seulement si la condition à la ligne 2 est remplie. La ligne 5 ne fait pas partie du if car il n'y a pas d'espaces ni de tabs devant. Elle s'exécute donc de toute façon.

Les if/else peuvent également être imbriqués les uns dans les autres:

```
1. a = int(input("Votre âge ? "))
2. if (a < 0):
3.   print("âge invalide!")
4. else:
5.   if (a < 18):
6.     print("Vous êtes mineur.")
7.     print("Vous serez majeur dans " + str(18-a) + " ans.")
8.   else:
9.     print("Vous êtes majeur.")
```

Ici on vérifie si a < 0, si oui on exécute la ligne 3 et rien d'autre. Sinon, on exécute d'abord la ligne 5. Comme c'est aussi un if, on regarde la condition et on exécute les lignes 6 et 7 si elle est remplie. Sinon on exécute la ligne 9.

Reprenons notre exemple de comparaison avec des variables de type str mais cette fois utilisons la construction if/else pour afficher d'autres messages:

```
3. texte = input("Qui a inventé le langage Python ? ")
4. if (texte == "Guido van Rossum"):
5.   print("juste")
6. else:
7.   print("faux")
```

```
3. texte = input("Qui a inventé le langage Python ? ")
4. if (texte != "Guido van Rossum"):
5.   print("faux")
6. else:
7.   print("juste")
```

Les deux programmes ci-dessus sont équivalents. Dans le premier on teste si la variable *texte* est égale à une valeur. Si oui on affiche "juste" sinon on affiche "faux".
Dans le deuxième programme, on teste si la variable *texte* **n'est pas égale** à cette valeur. Et on a inversé les *print* donc le comportement du programme reste le même.

Les opérateurs logiques

Il est également possible de modifier et de combiner les conditions pour créer des conditions plus complexes. Dans l'exemple plus haut où on demande l'âge de l'utilisateur, on aimerait afficher "invalide" si l'utilisateur entre un nombre plus petit que 0 ou un nombre plus grand qu'un âge maximum pour un humain, disons 150 pour être large. On pourrait le faire ainsi avec deux ifs :

```
1. a = int(input("Votre âge ? "))
2. if (a < 0):
3.   print("invalide")
4. if (a > 150):
5.   print("invalide")
```

Ce programme fonctionne mais on a écrit deux fois la même chose aux lignes 3 et 5. On aimerait une seule condition qui nous dit si l'âge est plus petit que 0 OU s' il est plus grand que 150. On peut le faire en utilisant l'opérateur OU qui en Python se note "or" :

```
1. a = int(input("Votre âge ? "))
2. if (a < 0 or a > 150):
3.   print("invalide")
```

Il existe également l'opérateur ET qui permet de combiner deux conditions pour créer une condition qui est remplie seulement si les deux premières sont toutes deux remplies. En Python, on le note "and":

```
1. username = input("Nom d'utilisateur: ")
2. password = input("Mot de passe: ")
3. if (username == "admin" and password == "sdghILG374_j3rv")
4.   print("accès autorisé")
5. else:
6.   print("accès refusé")
```

Dans l'exemple ci-dessus, on utilise l'opérateur "==" pour vérifier que la variable username contient la valeur "admin" ET on vérifie que la variable password contient la valeur "sdghILG374_j3rv". Si l'une des deux conditions n'est pas remplie, on exécute la partie else à la ligne 6. Seulement si les deux variables sont égales aux valeurs précisées on exécutera la ligne 4.

Il est également possible d'inverser une condition avec l'opérateur NON qui en Python s'écrit avec un "not". On peut grouper les conditions logiques avec des parenthèses. Le programme suivant se comporte exactement comme le précédent mais on a inversé la condition avec un "not". Cette fois on teste si ce n'est PAS le cas que username et password sont égaux à deux valeurs :

```
1. username = input("Nom d'utilisateur: ")
2. password = input("Mot de passe: ")
3. if (not (username == "admin" and password == "sdghILG374_j3rv")):
4.   print("accès refusé")
5. else:
6.   print("accès autorisé")
```

On peut combiner autant d'opérateurs et de parenthèses que l'on veut, tant que cela forme une condition qui peut être vérifiée. Ce programme affiche "condition remplie" si n1 est plus petit que 10 et que soit n2 est égal à 100 ou n3 n'est pas égal à n1 mais en plus il faut toujours que ce ne soit pas le cas que n4 soit plus grand que n3 et que n3 soit plus grand que n2 :

```
n1 = int(input("Nombre 1: "))
n2 = int(input("Nombre 2: "))
n3 = int(input("Nombre 3: "))
n4 = int(input("Nombre 4: "))
if (n1 < 10 and (n2 == 100 or n3 != n1) and not (n4 > n3 and n3 >
n2)):
  print("condition remplie")
```

En pratique vous aurez rarement besoin d'écrire des conditions aussi complexes mais il est important de comprendre que l'on peut les combiner à l'infini.

3. Les fonctions - définitions et appels

Nous introduisons maintenant de manière plus formelle un élément que nous avons rencontré depuis le début sans l'expliciter : les fonctions.
Dans ce chapitre nous allons introduire les bases des fonctions et nous y reviendrons plus en détails dans le chapitre 6.
Les fonctions nous permettent de réutiliser du code à plusieurs endroits sans devoir le réécrire. Cela nous permet également "d'oublier" comment nous avons codé une certaine fonctionnalité et de nous concentrer sur son utilisation dans le reste du code. On appelle cela l'encapsulation.

Considérons l'exemple de code suivant :

```
1.  choix = input("que voulez-vous faire ?")
2.  if (choix == "voir mes notes"):
3.    username = input("Nom d'utilisateur: ")
4.    password = input("Mot de passe: ")
5.    if (username == "admin" and password == "1234"):
6.      print("accès autorisé")
7.    else:
8.      print("accès refusé")
9.  if (choix == "voir mes absences"):
10.   username = input("Nom d'utilisateur: ")
11.   password = input("Mot de passe: ")
12.   if (username == "admin" and password == "1234"):
13.     print("accès refusé")
14.   else:
15.     print("accès autorisé")
16.  if (choix == "voir l'adresse email d'un.e enseignant.e"):
17.   print("accès autorisé")
```

Un élève interagit avec un service en ligne du gymnase. Il peut soit voir ses notes, voir ses absences ou consulter une adresse email. Admettons que les deux premiers choix demandent d'être connecté, alors que le dernier choix est public et ne demande aucun mot de passe.
Dans les deux premiers ifs aux lignes 2 et 9, le code est en réalité identique. On peut donc le séparer dans une fonction avec un nom approprié. Grâce au mot-clé **def**, nous pouvons mettre notre code répétitif dans une fonction, lui donner un nom et ***appeler cette fonction par son nom suivi de parenthèses*** à chaque fois que nous voulons

exécuter le même code. Voici le même programme simplifié grâce à une fonction :

```
1.  def login():
2.    username = input("Nom d'utilisateur: ")
3.    password = input("Mot de passe: ")
4.    if (username == "admin" and password == "1234"):
5.      print("accès autorisé")
6.    else:
7.      print("accès refusé")
8.
9.  choix = input("que voulez-vous faire ?")
10.  if (choix == "voir mes notes"):
11.    login()
12.  if (choix == "voir mes absences"):
13.    login()
14.  if (choix == "voir l'adresse email d'un.e enseignant.e"):
15.    print("accès autorisé")
```

On commence par **définir** notre fonction avant de pouvoir l'utiliser, de la même manière que l'on doit créer une variable avant de pouvoir l'utiliser. Remarquez la syntaxe de cette définition à l'aide du mot-clé *def*, suivi du nom de la fonction (ici **login**), de parenthèses et d'un ":" qui implique que tout ce qui est inclus dans la définition de la fonction sera décalé d'un tab vers la droite.
En l'occurrence, cette fonction utilise une boucle *while* pour afficher le contenu du tableau *tab*.
On peut ensuite **appeler** cette fonction 2 fois avec le nom de la fonction suivi de parenthèses. Chacun de ces 2 appels exécutera le même code présent dans la définition de la fonction.
En résumé, une fonction est présente de deux manières distinctes dans un code : d'abord sa **définition** (1 seule fois), puis son **appellation** (autant de fois que désiré).

Remarquez maintenant que *print(), input(), int(), str()* et *float()* que nous avons vues précédemment sont toutes des fonctions exactement au même titre que login(). La différence est que nous avons seulement **appelé** les fonctions *print(), input(), int(), str()* et *float()*. Nous n'avons pas vu leur **définition** qui est cachée par python pour ces fonctions de base.

4. Les boucles

Si je veux répéter une instruction, par exemple afficher plusieurs fois la même chose sans copier-coller, j'ai besoin d'une nouvelle construction qu'on appelle une **boucle**. Il y a 2 types de boucles : **while** et **for**. Dans ce chapitre, on se concentre sur les boucles **while**.

Voilà un exemple de boucle **while**:

```
1. compteur = 1
2. while (compteur <= 10):
3.   print("Salut !")
4.   compteur = compteur + 1
5. print("Fini")
```

```
> Salut !
> Salut !
> Salut !
> Salut !
> Salut !
> Salut !
> Salut !
> Salut !
> Salut !
> Salut !
> Fini
```

Analysons ce qui se passe. Le mot-clé "while", tout comme le mot-clé "if" vu au chapitre précédent, prend une condition en argument entre les parenthèses, ici "compteur <= 10". Les instructions qui suivent et qui sont décalées vers la droite (lignes 3 et 4) font partie de la structure while et sont exécutées ***tant que la condition est vraie***. Donc dans le cas présent, tant que compteur <= 10.
Initialement le compteur vaut 1, donc à la ligne 2 la condition 1<=10 est vraie. Donc la ligne 3 s'exécute et affiche "Salut !". Ensuite la ligne 4 s'exécute et le compteur vaut 2. Maintenant la suite de l'exécution se complique : on revient à la ligne 2 et on re-vérifie la condition compteur <=10. La condition 2 <=10 est vraie donc on ré-exécute les lignes 3 et 4. On revient à la ligne 2 encore une fois et cette fois le compteur vaut 3 et ainsi de suite…
On affiche donc 10 fois "Salut !", puis la dernière fois à la ligne 4, compteur vaut 10 + 1

= 11. On revient à la ligne 2 et cette fois la condition compteur <= 10 est fausse puisque 11 est plus grand que 10. ***On sort donc de la boucle while*** et on continue l'exécution à la ligne 5 qui affiche "Fini".

Ce premier exemple montre comment exécuter un certain nombre de fois la même chose, mais on peut surtout utiliser la valeur du compteur comme dans cet exemple qui affiche un décompte :

```
1. decompte = int(input("Donnez un compte de départ: "))
2. while (decompte > 0):
3.   print("Décollage dans " + str(decompte) + "s...")
4.   decompte = decompte - 1
5. print("Décollage!")
```

```
> Donnez un compte de départ: 3
> Décollage dans 3s...
> Décollage dans 2s...
> Décollage dans 1s...
> Décollage!
```

Cette fois on exécute des instructions tant que la variable "decompte" est plus grande que 0. Si l'utilisateur lui donne 3 comme valeur, on va d'abord entrer dans la boucle puisque 3 > 0, afficher le message en utilisant la valeur actuelle de la variable "decompte" puis ré-assigner la variable "decompte" avec 3 - 1 = 2 comme valeur.

On re-vérifie que cette fois 2 > 0, on entre de nouveau dans la boucle, puis même chose avec 1 > 0 et finalement la condition 0 > 0 est fausse. A ce moment-là, on sort de la boucle et on continue l'exécution à la ligne 6 qui affiche le dernier message.

La condition n'a pas besoin d'être un compteur, on peut exécuter par exemple un programme tant que l'utilisateur n'a pas donné un certain input:

```
1. answer = ""
2. while (answer != "Guido Van Rossum"):
3.   answer = input("Qui a inventé le langage Python ?")
4.   if (answer == "Guido Van Rossum"):
5.     print("Juste!")
6.   else:
7.     print("Faux!")
```

```
> Qui a inventé le langage Python ? Je sais pas
> Faux!
> Qui a inventé le langage Python ? moi
> Faux!
> Qui a inventé le langage Python ? guido van rossum
> Faux!
> Qui a inventé le langage Python ? Guido Van Rossum
> Juste!
```

Dans le programme ci-dessus, la boucle est exécutée tant que l'utilisateur n'a pas tapé exactement la chaîne de caractères "Guido Van Rossum". Initialement on compare la chaîne de caractères vide "" avec la chaîne de caractères "Guido Van Rossum", comme elles ne sont pas égales (l'opérateur !=), la condition est vérifiée et on entre dans la boucle. L'utilisateur donne une nouvelle valeur à la variable "answer" et on recommence, jusqu'à ce que l'utilisateur donne la valeur attendue pour que la condition à la ligne 2 devienne fausse. Le programme peut donc potentiellement tourner à l'infini.

La condition d'une boucle while peut être complexe avec des opérateurs logiques and, or ou not comme vus au chapitre précédent. Ou alors extrêmement simple en utilisant juste la valeur True comme le montre ce programme qui ne termine jamais :

```
1. while(True):
2.   print("Salut à l'infini!")
```

```
> Salut à l'infini!
> Salut à l'infini!
> Salut à l'infini!
> Salut à l'infini!
> Salut à l'infini!
> Salut à l'infini!
> ...
```

Pour forcer un programme qui tourne à l'infini à s'arrêter, appuyer les touches *Ctrl* et *C* en même temps.

5. Les tableaux

Jusqu'à présent, nous avons stocké chaque information nécessaire à nos programmes dans une variable de type **str**, **int** ou **float**. Si nous avons une grande quantité d'informations similaires à traiter, le code peut vite devenir long et répétitif.
Prenons cet exemple d'un programme qui demanderait 5 notes à l'utilisateur pour calculer la moyenne puis les afficher :

```
note1 = float(input("Donnez une note : "))
note2 = float(input("Donnez une note : "))
note3 = float(input("Donnez une note : "))
note4 = float(input("Donnez une note : "))
note5 = float(input("Donnez une note : "))
somme = note1 + note2 + note3 + note4 + note5
moyenne = somme/5.0
print("Votre moyenne : " + str(moyenne))
print("Votre note 1 : " + str(note1))
print("Votre note 2 : " + str(note1))
print("Votre note 3 : " + str(note1))
print("Votre note 4 : " + str(note1))
print("Votre note 5 : " + str(note1))
```

Imaginez si ce programme devait traiter non pas 5 notes mais 100 notes..

Définition de tableau/array

Pour résoudre ce problème, nous introduisons la notion de **tableau** ou **array** en anglais. L'idée est de regrouper des variables d'un même type dans une seule grosse variable appelée tableau.
Par exemple on peut avoir un tableau de 5 **int**, un tableau de 200 **str** ou un tableau de 1000 **float**. Chaque tableau est également une variable. La syntaxe pour déclarer un tableau utilise les symboles []. L'exemple suivant déclare un tableau de **int** et un tableau de **str**.

```
1. mon_tableau = [2, 4, 5, 10, -3, 0]
2. mon_autre_tableau = ["salut", "hello", "blablabla"]
```

La variable *mon_tableau* est dite de type **int[]** alors que la variable *mon_autre_tableau*

est dite de type **str[]**. Notez bien les crochets après **int** pour dénoter qu'il ne s'agit pas d'un nombre entier, mais d'un tableau de nombres entiers. En l'occurrence, ce tableau contient 6 nombres entiers. Similairement, on note **str[]** pour dénoter qu'il ne s'agit pas d'une chaîne de caractères mais d'un tableau de chaînes de caractères (en l'occurrence 3 chaînes de caractères).

Accès aux éléments du tableau

Une fois que nous avons pu définir nos variables de type tableau, nous pouvons accéder à chaque élément du tableau individuellement en utilisant un **index**. Un index est un nombre entier, donc une variable de type **int**, qui désigne une position dans le tableau en partant de la gauche vers la droite et en commençant par 0.
Dans l'exemple précédent, la variable *mon_autre_tableau* contient 3 chaînes de caractères. La chaîne "salut" est en première position depuis la gauche, on lui définit donc l'index 0. La chaîne "hello" en deuxième position obtient l'index suivant : 1. Et finalement la chaîne "blablabla" obtient l'index 2.
On peut récupérer la valeur à un index i donné avec la syntaxe ***mon_autre_tableau[i]***. Une fois la valeur récupérée, on peut la traiter comme n'importe quelle variable. Ici chacune des 3 valeurs est une variable de type **str**. Exemples :

```
1. mon_autre_tableau = ["salut", "hello", "blablabla"]
2. print(mon_autre_tableau[0])
3. ma_variable = mon_autre_tableau[1]
4. print(ma_variable)
5. mon_index = 2
6. print(mon_autre_tableau[mon_index])
```

```
> salut
> hello
> blablabla
```

A la ligne 2, on accède au premier élément du tableau à l'index 0 et on l'affiche. A la ligne 3, on accède au deuxième élément du tableau (index 1) et on le stocke dans une variable qui est donc de type **str**. On affiche cette variable en ligne 4. A la ligne 5, on définit un index qui vaut 2 et on l'utilise en ligne 6 pour afficher le 3e élément du tableau.
Cela vous montre qu'on peut traiter chaque élément comme n'importe quelle variable et que même l'index peut être une variable que vous avez définie au préalable.

Attention : le tableau a une taille bien définie. Dans notre exemple, il contient 3 éléments. Donc le dernier index vaut 2. Si vous essayez d'accéder à un index qui n'existe pas, par exemple, *mon_autre_tableau[8]*, python vous retournera une erreur.

Taille/Longueur d'un tableau

La taille ou longueur d'un tableau est définie par le nombre d'éléments que le tableau contient. On peut y accéder grâce à la fonction *len()* :

```
1. tab = [5, 4, -10, 0]
2. print(len(tab))
```

```
> 4
```

Modifier et ajouter des éléments au tableau

Avec la même syntaxe [], il est possible de modifier des éléments du tableau en spécifiant à quel index on souhaiterait modifier le contenu du tableau. On utilise l'assignation avec = de la même manière qu'on définit une variable pour une variable de type **int**, **str** ou **float**.

```
1. un_tableau = [0.5, 1.23, 12.004, 4.5, 6.1]
2. print(un_tableau[3])
3. un_tableau[3] = 28.0
4. print(un_tableau[3])
```

```
> 4.5
> 28.0
```

A la ligne 2, on affiche le contenu initial du 4e élément du tableau (index 3). A la ligne 3, on modifie le contenu toujours à l'index 3 avec une nouvelle valeur. A la ligne 4, on affiche de nouveau le contenu du tableau à l'index 3 et on obtient une valeur différente, la nouvelle valeur écrite à la ligne précédente.

Il est également possible d'ajouter des nouveaux éléments au tableau tout à droite sans modifier ceux déjà présents. Pour cela, on utilise la fonction *append()* de la manière suivante :

```
1. tab = ["jean", "marc", "charlotte", "lise"]
2. print(tab)
3. print(len(tab))
4. tab.append("vincent")
5. print(tab)
6. print(len(tab))
```

```
> ["jean", "marc", "charlotte", "lise"]
> 4
> ["jean", "marc", "charlotte", "lise", "vincent"]
> 5
```

Dans cet exemple, le tableau est passé de 4 à 5 éléments.

Supprimer des éléments d'un tableau

On peut également supprimer un élément d'un tableau à un index donné en utilisant le mot-clé "del" :

```
1. tab = ["jean", "marc", "charlotte", "lise"]
2. print(tab)
3. print(len(tab))
4. del tab[1]
5. print(tab)
6. print(len(tab))
```

```
> ["jean", "marc", "charlotte", "lise"]
> 4
> ["jean", "charlotte", "lise", "vincent"]
> 3
```

Parcourir tous les éléments du tableau

On peut maintenant combiner la boucle **while** avec les nouvelles notions d'index pour parcourir un tableau de n'importe quelle taille et afficher tous ses éléments.

```
1. index = 0
2. tab = ["jean", "marc", "charlotte", "lise", "vincent"]
3. while (index < 5):
4.   print("Prénom en index " + str(index) + " : " + tab[index])
5.   index = index + 1
```

```
> Prénom en index 0 : jean
> Prénom en index 1 : marc
> Prénom en index 2 : charlotte
> Prénom en index 3 : lise
> Prénom en index 4 : vincent
```

On peut maintenant reprendre notre exemple initial et cette fois écrire facilement un programme qui affichera 100 notes et leur moyenne :

```
1. mes_notes = []
2. compteur  = 1
3. while (compteur <= 100):
4.   nouvelle_note = float(input("Donnez une " + str(compteur) + "e note : "))
5.   compteur = compteur + 1
6. mes_notes.append(nouvelle_note)
7. index = 0
8. somme = 0
9. while (index < 100):
10.      print("Votre " + str(index + 1) + "e note est : " + mes_notes[index])
11.   somme = somme + mes_notes[index]
12.   index = index + 1
   print("Votre moyenne est : " + str(somme/100.0))
```

Remarquez qu'à la ligne 1, on déclare un tableau vide. Il ne contient aucun élément, puis dans la première boucle, on lui ajoute 100 éléments donnés par l'utilisateur avec la fonction *append()*. Dans la deuxième boucle, on peut accéder aux 100 éléments du tableau pour les afficher et calculer leur moyenne.

Vérifier si un tableau contient une valeur

Grâce au mot-clé “*in*” il est possible d’obtenir une valeur booléenne *True/False* pour savoir si une valeur fait partie des éléments contenus dans un tableau :

```
1. tab1 = ["jean", "marc", "charlotte", "lise"]
2. tab2 = [45, -9, 18.4, 29, 103]
3. print("jean" in tab1)
4. print("vincent" in tab1)
5. print(4 in tab2)
6. if (103 in tab2):
7.   print("Oui")
8. else:
9.   print("Non")
```

```
> True
> False
> False
> Oui
```

6. Les fonctions - arguments et return

Grâce aux listes/tableaux, on peut écrire des programmes courts même avec de grandes quantités de données. Cependant, imaginez qu'on veuille écrire un programme qui effectue des opérations diverses sur un tableau, puis affiche tous les éléments du tableau ligne par ligne entre chaque opération:

```
tab = [6, 32, 0, -1]
index = 0
while (index  < 4):
  print("Contenu du tableau à l'index " + str(index) + " : " + str(tab[index]))
  index = index + 1
tab[0] = 4
index = 0
while (index  < 4):
  print("Contenu du tableau à l'index " + str(index) + " : " + str(tab[index]))
  index = index + 1
tab[3] = 10
index = 0
while (index  < 4):
  print("Contenu du tableau à l'index " + str(index) + " : " + str(tab[index]))
  index = index + 1
```

```
> Contenu du tableau à l'index 0 : 6
> Contenu du tableau à l'index 1 : 32
> Contenu du tableau à l'index 2 : 0
> Contenu du tableau à l'index 3 : -1
> Contenu du tableau à l'index 0 : 4
> Contenu du tableau à l'index 1 : 32
> Contenu du tableau à l'index 2 : 0
> Contenu du tableau à l'index 3 : -1
> Contenu du tableau à l'index 0 : 4
> Contenu du tableau à l'index 1 : 32
> Contenu du tableau à l'index 2 : 0
> Contenu du tableau à l'index 3 : 10
```

Nous avons 3 boucles while identiques pour afficher le contenu du tableau ligne par ligne avec une phrase. Notre programme peut vite devenir très long si les répétitions sont fréquentes, le rendant difficile à lire. De plus, si vous faites une erreur dans cette partie du code, vous devrez la réparer à chaque endroit où vous l'avez utilisé.

Définition et appels de fonctions (rappel)

Nous avons déjà vu au chapitre 3 comment réutiliser du code grâce aux fonctions :

```
def afficher_tableau():
  index = 0
  while (index < 4):
    print("Contenu du tableau à l'index " + str(index) + " : " + str(tab[index]))
    index = index + 1


tab = [6, 32, 0, -1]
afficher_tableau()
tab[0] = 4
afficher_tableau()
tab[3] = 10
afficher_tableau()
```

Arguments de fonctions

Regardons ce qui se passe si nous avons 2 tableaux distincts et que vous voulons afficher leurs contenus sans modifier notre fonction :

```
def afficher_tableau():
  index = 0
  while (index < 4):
    print("Contenu du tableau à l'index " + str(index) + " : " + str(tab[index]))
    index = index + 1


tab = [6, 32, 0, -1]
afficher_tableau()
tab2 = [100, 43, 3, 974]
afficher_tableau()
```

```
> Contenu du tableau à l'index 0 : 6
> Contenu du tableau à l'index 1 : 32
> Contenu du tableau à l'index 2 : 0
> Contenu du tableau à l'index 3 : -1
> Contenu du tableau à l'index 0 : 6
> Contenu du tableau à l'index 1 : 32
> Contenu du tableau à l'index 2 : 0
> Contenu du tableau à l'index 3 : -1
```

On remarque que le contenu de tab est affiché deux fois et que le contenu de tab2 n'est jamais affiché. En effet, la fonction afficher_tableau() est appelée deux fois mais si on regarde sa définition, elle utilise le tableau tab, donc forcément chaque appel de cette fonction utilise toujours le tableau tab.
Il nous faut un moyen de spécifier une information supplémentaire lors de chaque appel à la fonction pour lui dire quel tableau elle devrait utiliser dans son code.

Jusqu'à présent les parenthèses étaient vides, leur but est en réalité de contenir les **arguments** ou **paramètres** de la fonction.

```
def afficher_tableau(un_tableau):
  index = 0
  while (index < 4):
    print("Contenu du tableau à l'index " + str(index) + " : " + str(un_tableau[index]))
    index = index + 1


tab = [6, 32, 0, -1]
afficher_tableau(tab)
tab2 = [100, 43, 3, 974]
afficher_tableau(tab2)
```

```
> Contenu du tableau à l'index 0 : 6
> Contenu du tableau à l'index 1 : 32
> Contenu du tableau à l'index 2 : 0
> Contenu du tableau à l'index 3 : -1
> Contenu du tableau à l'index 0 : 100
> Contenu du tableau à l'index 1 : 43
> Contenu du tableau à l'index 2 : 3
> Contenu du tableau à l'index 3 : 974
```

Remarquez que maintenant notre fonction *afficher_tableau()* contient dans ses parenthèses une variable *un_tableau*. Il s'agit d'un **paramètre** ou **argument** de la fonction. C'est une variable qui n'est définie qu'à l'intérieur de la fonction. Elle prendra une valeur différente à chaque appel de la fonction. Remarquez que dans la fonction on utilise maintenant *un_tableau[index]* pour obtenir la valeur dans le tableau passé en argument.
Dans les deux appels de la fonction, on passe maintenant *tab* à la fonction avec *afficher_tableau(tab)* puis *tab2* avec *afficher_tableau(tab2)*. Cela veut dire que dans le premier appel de la fonction, le paramètre *un_tableau* de la fonction sera en réalité le tableau *tab*. Dans le deuxième appel de la fonction, le paramètre *un_tableau* sera en réalité *tab2*.
De cette manière, on peut dire à la fonction quel tableau elle doit afficher et on peut la réutiliser.

On peut passer autant d'arguments que l'on veut à la fonction en les séparant par des virgules. Par exemple, on pourrait passer une **str** pour donner le nom du tableau à afficher :

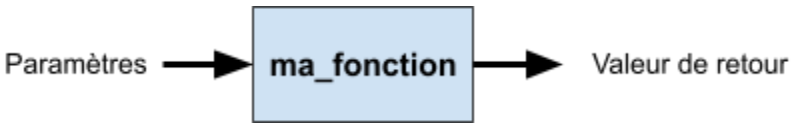
```
def afficher_tableau(nom_du_tableau, un_tableau):
  index = 0
  while (index < 4):
    print("Contenu du tableau appelé " + nom_du_tableau + " à l'index " +
str(index) + " : " + str(un_tableau[index]))
    index = index + 1


tab = [6, 32, 0, -1]
afficher_tableau("mon tableau", tab)
tab2 = [100, 43, 3, 974]
afficher_tableau("mon autre tableau", tab2)
```

```
> Contenu du tableau appelé mon tableau à l'index 0 : 6
> Contenu du tableau appelé mon tableau à l'index 1 : 32
> Contenu du tableau appelé mon tableau à l'index 2 : 0
> Contenu du tableau appelé mon tableau à l'index 3 : -1
> Contenu du tableau appelé mon autre tableau à l'index 0 : 100
> Contenu du tableau appelé mon autre tableau à l'index 1 : 43
> Contenu du tableau appelé mon autre tableau à l'index 2 : 3
> Contenu du tableau appelé mon autre tableau à l'index 3 : 974
```

Valeur de retour d'une fonction

Une fonction peut être vue comme une boîte qui prend des données en entrée, les transforme ou effectue une action, puis retourne d'autres données en sortie.



On a vu comment donner certaines valeurs d'entrée en paramètres. Pour l'instant, notre fonction ne retourne rien en sortie. Elle affiche un tableau, puis termine son exécution.
Il est possible, grâce au mot-clé return, de renvoyer une valeur de sortie depuis une fonction. Cette valeur pourra être utilisée depuis l'endroit dans le code où la fonction a été appelée. Le code suivant montre un exemple avec une fonction qui calcule la moyenne d'un tableau :

```
def moyenne(un_tableau):
  index = 0
  somme = 0
  while (index < len(un_tableau)):
    somme = somme + un_tableau[index]
    index = index + 1

  moyenne = somme/len(un_tableau)
  return moyenne


tab = [6, 32, 0, -1]
moyenne1 = moyenne(tab)
print(moyenne1)
tab2 = [100, 43, 3, 974]
print("deuxième moyenne: " + str(moyenne(tab2)))
```

```
> 9.25
> deuxième moyenne: 280.0
```

La fonction *moyenne()* prend un tableau appelé *un_tableau* en argument, calcule une valeur stockée dans une variable *moyenne*, puis retourne cette variable comme résultat grâce à la dernière ligne de la fonction : ***return moyenne***.
Dans le premier appel de la fonction, on passe le tableau *tab* en argument et on récupère la moyenne de ce tableau dans la variable *moyenne1* qu'on affiche à la ligne suivante.
Dans le deuxième appel de la fonction, on passe le tableau *tab2* en argument et on utilise directement cette valeur dans un *print()* pour l'afficher. Remarquez donc qu'on n'est pas obligés de stocker temporairement cette valeur de retour dans une variable.

7. Les tables de hachage / dictionnaires

Jusqu'à présent, pour stocker de plus grandes quantités de données que les variables simples ne permettent, nous avons toujours utilisé les tableaux. Les tableaux sont en réalité une manière d'organiser des données et font partie d'un ensemble de constructions appelées ***data structures (structures de données)***.

Il existe beaucoup de structures de données qui dépassent le cadre de ce cours. Ici, nous allons parler d'une structure de données : les tables de hachage. Dans le contexte du langage de programmation Python, elles sont souvent appelées "dictionnaires" et nous utiliserons ces deux termes comme synonymes.

Les tableaux permettent de stocker des valeurs côte à côte de manière contiguë dans la mémoire. Cela est pratique pour parcourir tous les éléments du premier au dernier rapidement. Mais que se passe-t-il lorsque nous voulons chercher si le tableau contient une valeur spécifique ? Il nous faut parcourir tout le tableau pour rechercher cette valeur. Supprimer un élément du tableau est également coûteux car tous les éléments qui suivent l'index supprimé doivent être décalés.

Les tables de hachage fournissent une structure de données qui permet de trouver/supprimer un élément particulier beaucoup plus rapidement. Au lieu d'avoir un index pour chaque valeur, on aura une clé. Cette clé est une variable de n'importe quel type. A chaque clé correspond une valeur.

Le schéma suivant montre un exemple d'une table de hachage à 3 entrées, donc 3 clés et 3 valeurs, qui donne l'âge d'une personne en fonction de son prénom. Ici les clés sont des variables str qui représentent un prénom et les valeurs sont des variables int qui représentent un âge.

"Paul"	→	16
"Maria"	→	42
"Clara"	→	28

Cette table peut être déclarée en Python ainsi :

```
1. ages = {"Paul": 16, "Maria": 42, "Clara": 28}
```

Accès et modification

Chaque valeur peut être accédée ou modifiée en donnant la clé entre [ ], similairement à ce qu'on a vu pour les éléments d'un tableau qui sont accédés par leur index. La clé dans un dictionnaire est l'équivalent d'un index dans un tableau, sauf que la clé peut être une variable de n'importe quel type ou contenu alors que l'index est toujours un *int* dans un ordre croissant.

```
1. ages = {"Paul": 16, "Maria": 42, "Clara": 28}
2. print(ages["Paul"])
3. x = ages["Maria"]
4. n = "Clara"
5. print(x)
6. print(ages[n])
7. ages["Paul"] = 22
8. print(ages["Paul"])
9. print(n + " a " + str(ages[n]) + " ans.")
```

```
> 16
> 42
> 28
> 22
> Clara a 28 ans.
```

Appartenance à une table de hachage

On peut vérifier de manière rapide si un élément est présent dans une table de hachage de manière identique au tableau en utilisant le mot-clé “*in*”. Sauf que dans le cas du tableau, en arrière-plan Python doit parcourir tous les éléments pour savoir si le tableau contient cet élément. Dans le cas de la table de hachage, Python peut beaucoup plus rapidement avoir immédiatement accès à cette information en vérifiant s’il existe une valeur qui correspond à la clé recherchée.

```
1. ages = {"Paul": 16, "Maria": 42, "Clara": 28}
2. print("Maria" in ages)
3. print("Jean" in ages)
```

```
> True
> False
```

Ce résultat booléen (*True* ou *False*) peut être utilisé n’importe où dans un *if/else* ou un *while*.

Parcours des clés/valeurs d'une table de hachage

Il existe plusieurs manières de parcourir tous les éléments (chaque élément étant une clé + une valeur) d'une table de hachage. La plus courante et la plus simple utilise de nouveau le mot-clé "*in*" mais combiné avec une boucle **for**. Attention à ne pas confondre sa signification ! Dans le cas précédent, il fallait le comprendre comme "est-ce que cette clé est *dans* (*in*) la table de hachage ?", alors que dans ce cas il faut le comprendre comme "pour chaque clé *dans* (*in*) la table de hachage".

Les boucles for sont une abstraction pour écrire certaines boucles while de manière plus simple. On peut s'en servir pour parcourir les éléments d'une table de hachage / dictionnaire ou d'un tableau de manière similaire.

```
ages = {"Paul": 16, "Maria": 42, "Clara": 28}
for prenom in ages:
  print(prenom + " a  " + ages[prenom] + " ans.")
```

```
> Paul a 16 ans.
> Maria a 42 ans.
> Clara a 28 ans.
```

Ici la variable "prenom" représente une clé dans la table de hachage "ages". On entre dans la boucle **for** une fois par clé différente trouvée dans la table. La variable "prenom" prendra la valeur de chacune de ces clés et on pourra accéder à chaque valeur correspondante avec "ages[prenom]". Ainsi, on peut parcourir toutes les clés et valeurs et s'en servir pour n'importe quelle utilisation, ici simplement les afficher dans une phrase.