



جامعة عبد المالك السعدي
Université Abdelmalek Essadi



Programmation Python



2023-2024

Prof : Anouar RAGRAGUI

Plan du cours

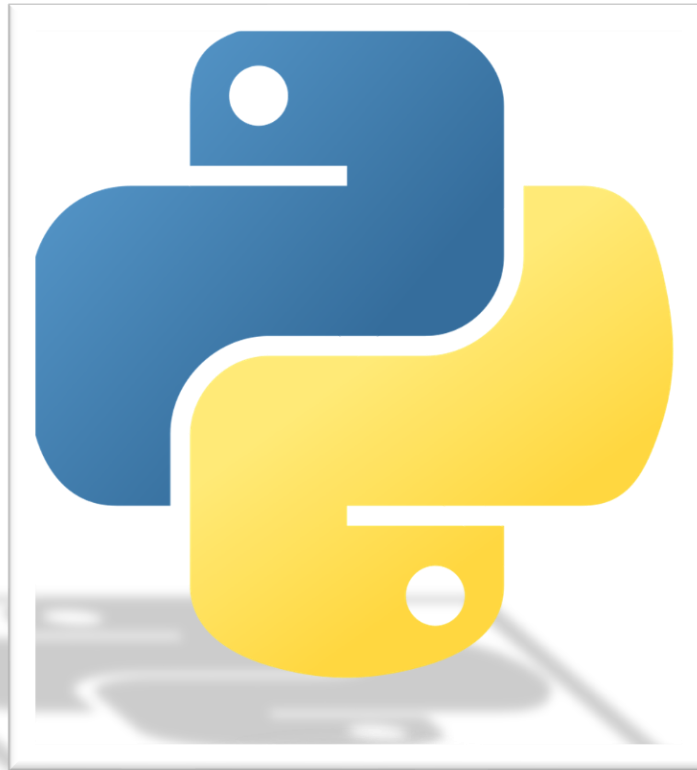
- Chapitre 1: Introduction
- **Chapitre 2: Les bases du langage Python**
- Chapitre 3: Notions avancées du langage Python



جامعة عبد المالك السعدي
Université Abdelmalek Essadi



Chapitre 2 : Les bases du langage Python



- ❑ Notion de bloc d'instructions et d'indentation
- ❑ Variables
- ❑ Structure de contrôle
- ❑ Les containers
- ❑ Les fonctions
- ❑ Les modules et les packages

Variables

Structure de contrôle

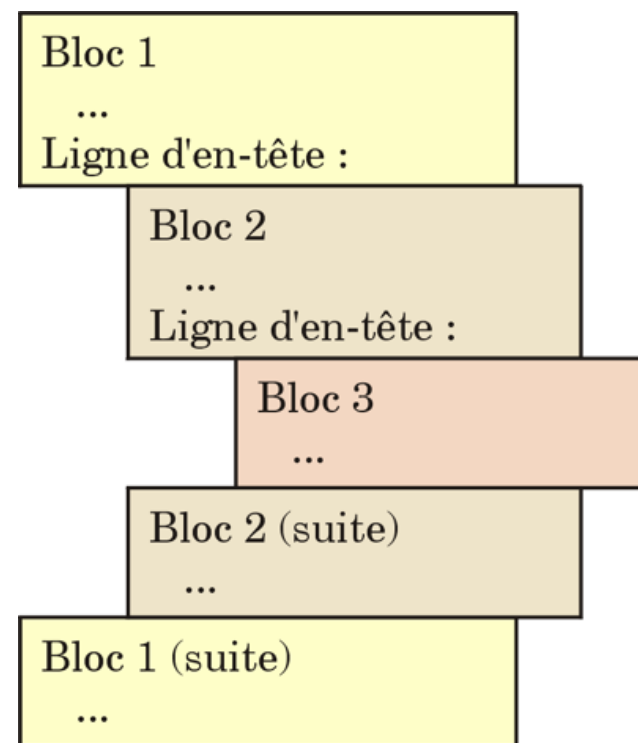
Les containers

Les fonctions

Les modules et les packages

5

- En **programmation**, il est courant de répéter un certain nombre de choses ou d'exécuter **plusieurs instructions (Bloc d'instructions)** si une condition est **vraie**.
- En **Python**, les **blocs d'instructions** ne sont pas délimités par des mots (endIf, enfFor) ni par des symboles, mais par **des lignes indentées** (décalées) d'un nombre fixe de caractères (**4 espaces** ou une **tabulation en général**)
- Pratiquement, **l'indentation** en **Python** doit être **homogène** (soit des espaces, soit des tabulations, mais pas un mélange des deux). Une **indentation** avec **4 espaces** est le style d'indentation recommandé
- **Remarque:** Une mauvaise indentation va provoquer des erreurs.



- Une **variable** est une **zone de la mémoire** de l'ordinateur dans laquelle une valeur est stockée.
- Cette variable est définie par un **nom**, alors que pour l'ordinateur il s'agit en fait d'une **adresse**, c'est-à-dire d'une zone particulière de la mémoire.
- En **Python**, la déclaration d'une **variable** et son **initialisation** (c'est-à-dire la première valeur que l'on va stocker dedans) se font en même temps.
- **Exemple:**

```
>>> x = 2
>>> x
2
```
- Dans l'exemple ci-dessus **Python** a :
 - ▣ Deviné que la **variable** était un **entier**. On dit que **Python** est un langage au **typage dynamique**.
 - ▣ **Alloué** (réservé) l'**espace en mémoire** pour y accueillir un **entier**.
 - ▣ **Assigné** la valeur 2 à la variable x

□ Le **type d'une variable** correspond à la **nature** de celle-ci. Les **trois principaux** types sont:

- ▣ **Entiers (**int**)** : Les **entiers** représentent des **nombre entiers**, **positifs** ou **négatifs**, **sans partie décimale**. Par **exemple** :

`x = 5`

`y = -10`

- ▣ **Flottants (**float**)** : Les **flottants** représentent des nombres à **virgule flottante**, c'est-à-dire des nombres **avec une partie décimale**. Par **exemple** :

`pi = 3.14`

`temperature = 98.6`

- ▣ **Chaînes de caractères (**str**)** : Les **chaînes de caractères** représentent des **séquences de caractères**. Elles peuvent contenir des **lettres**, des **chiffres**, des **espaces** et d'autres **caractères spéciaux**. Les chaînes de caractères sont déclarées en utilisant des **guillemets simples** (`'`) ou **doubles** (`"`). Par **exemple** :

`nom = "ENSAH"`

`message = 'Bienvenue!'`

- **Python** prend en charge de nombreux autres **types de données**. Voici quelques-uns d'entre eux :

- **Booléens** (**bool**) : Les booléens représentent les valeurs de vérité, **True** ou **False**. Ils sont souvent utilisés dans des **expressions conditionnelles**. Par **exemple** :

```
est_vrai = True
```

```
est_faux = False
```

- **Nombres complexes** (**complex**) : Les **nombres complexes** sont utilisés pour représenter des **quantités** avec une **partie réelle** et une **partie imaginaire**. Ils sont déclarés en utilisant la lettre **j** pour représenter la **partie imaginaire**. Voici un **exemple** :

```
nombre_complexe = 3 + 2j
```

- **Listes** (**list**), **Tuples** (**tuple**), **Dictionnaires** (**dict**), **Ensembles** (**set**)...

- **Remarque:**

- Il faut **entourer** une chaîne de caractères de **guillemets** (**doubles**, **simples**, voire **trois guillemets successifs** doubles ou simples) afin d'indiquer à **Python** le **début** et la **fin** de la chaîne de caractères
- En **Python**, comme dans la plupart des **langages de programmation**, c'est le **point** qui est utilisé comme **séparateur décimal**.

□ Exemple:

```
>>> y = 3.14
>>> y
3.14
>>> a = " bonjour "
>>> a
'bonjour '
>>> b = 'salut '
>>> b
'salut '
>>> c = """ girafe """
>>> c
'girafe '
>>> d = '''lion '''
>>> d
'lion '
```

- ❑ Le **nom des variables** en **Python** peut être constitué de **lettres minuscules** (a à z), de **lettres majuscules** (A à Z), de **nombres** (0 à 9) ou du **caractère souligné** (_).
- ❑ **Remarque:**
 - ❑ Vous ne pouvez pas utiliser **d'espace** dans un nom de variable.
 - ❑ Par ailleurs, un nom de variable ne doit pas **débuter par un chiffre** et il n'est pas recommandé de le faire débuter par le caractère _
 - ❑ De plus, il faut absolument éviter d'utiliser **un mot réservé** par **Python** comme nom de variable (par exemple : print, range, for, from, etc.).
- ❑ **Python** est **sensible à la casse**, ce qui signifie que les variables Test, test et TEST sont différentes.

- Le **typage** fait référence au **catégorisation** des données dans un langage de programmation en fonction de leur **type**.
- En effet, il concerne la manière dont les **variables** et les **valeurs** sont **traitées** en termes **de type de données**, c'est-à-dire le genre d'informations qu'elles représentent :
 - ▣ **Typage statique** : Le **type** d'une **variable** est déterminé **au moment de la compilation**, et il ne peut **pas être modifié** pendant **l'exécution** du programme. Les langages comme C, C++, et Java sont des exemples de langages à **typage statique**.
 - ▣ **Typage dynamique** : Dans un système de typage dynamique, le **type** d'une **variable** est déterminé au **moment de l'exécution** du programme. Les **variables** peuvent **changer de type** pendant **l'exécution**. **Python** est un exemple de langage à **typage dynamique**.
- En **Python**, il n'est pas nécessaire d'écrire des lignes de programme spécifiques pour **définir le type des variables** avant de pouvoir les utiliser.
- Il suffit **d'assigner** une **valeur** à un **nom** de **variable** pour que celle-ci soit **automatiquement** créée avec le **type** qui **correspond** au mieux à la **valeur fournie**.

□ Exemple:

```
>>> n = 7 # nombre entier
```

```
>>> msg = "Bonjour" # chaîne de caractères
```

```
>>> pi = 3.14159 # nombre à virgule flottante
```

□ Le **typage** des **variables** sous **Python** est un **typage dynamique**.

□ Le **typage statique** est préférable dans le cas des **langages compilés**, parce qu'il permet **d'optimiser l'opération de compilation** (dont le résultat est un code binaire)

- L'affectation est l'instruction qui permet d'attribuer à une **variable**, une **valeur**, le **contenu d'une autre variable** ou une **expression**, en utilisant l'opérateur d'affectation **=**
- Une expression est composée **d'opérandes**, **d'opérateurs** et de **parenthèses**, et **équivalente à une seule valeur**.
- **Un opérande** est une quantité sur laquelle une opération est exécutée. Il peut être une valeur, une constante, une fonction, etc.
- **Un opérateur** est un signe qui relie deux opérandes, pour produire un résultat.
- Les types d'opérateurs et des opérandes dépendent de la nature de l'expression entrant en jeux.
- En Programmation, on utilise trois types d'expressions :
 - Les expressions **arithmétiques**.
 - Les expressions **logiques**.
 - Les expressions **alphanumériques**.

- **Remarque :** Sous Python, on peut assigner une valeur à plusieurs variables simultanément.

- **Exemple:**

```
>>> x = y = 7
```

```
>>> x
```

```
7
```

```
>>> y
```

```
7
```

- **Remarque :** On peut aussi effectuer des affectations parallèles à l'aide d'un seul opérateur:

- **Exemple:** Dans cet exemple, les variables a et b prennent simultanément les nouvelles valeurs 4 et 8,33.

```
>>> a, b = 4, 8.33
```

```
>>> a
```

```
4
```

```
>>> b
```

```
8.33
```

- **Exemple:** Écrivez les instructions permettant d'échanger les valeurs des variables A=6 et B=7 (la permutation).

- **Solution 1:**

```
a = 4
```

```
b = 6
```

```
c = a
```

```
a = b
```

```
b = c
```

- **Solution 2:**

```
a = 4
```

```
b = 6
```

```
a, b = b, a
```

- Une **expression arithmétique** est équivalente à une **valeur de type numérique**, et on a :

- ▣ Les **opérandes** sont des valeurs **numériques** ou des **variables** de type numérique.
- ▣ Et les **opérateurs** sont les opérateurs arithmétiques et unaires :

+ : addition

- : soustraction

***** : multiplication

/ : division réelle

****** : puissance

% : modulo

// : division entière

Unaire : + et -

- Les **quatre opérations arithmétiques** de base se font de manière **simple** sur les types numériques (int et floats)
- **Remarque:**
 - ▣ Si vous mélangez les types **entiers** et **floats**, le résultat est renvoyé comme un **float** (car ce type est plus général).
 - ▣ L'utilisation de **parenthèses** permet de gérer les **priorité**

□ Exemple:

```
>>> x = 45
```

```
>>> x + 2
```

```
47
```

```
>>> x - 2
```

```
43
```

```
>>> x * 3
```

```
135
```

```
>>> y = 2.5
```

```
>>> x - y
```

```
42.5
```

```
>>> (x * 10) + y
```

```
452.5
```

- Les opérateurs d'affectation élargi : Il existe des opérateurs combinés qui effectue une opération et une affectation en une seule étape

- Ces opérateurs remplace l'instruction

`Opérande1 = Opérande1 Opérateur Opérande2 ;`

Par :

`Opérande1 Opérateur= Opérande2 ;`

- Ainsi on pourra remplacer l'instruction $n = n + k$ par $n += k$.
- Cette possibilité concerne tous les opérateurs binaires arithmétiques et de manipulation de bits.
- Voici la liste complète :

`+=, -=, *=, /=, %=, |=, &=, ^=, <<= et >>=.`

□ Exemple

```
>>> i = 0
>>> i = i + 1
>>> i
1
>>> i += 1
>>> i
2
>>> i += 2
>>> i
4
```

□ Remarque:

- ▣ L'opérateur += effectue une addition puis affecte le résultat à la même variable. Cette opération s'appelle une incrémentation.
- ▣ Les opérateurs -=, *= et /= se comportent de manière similaire pour la soustraction, la multiplication et la division

- Pour **les chaînes de caractères**, deux opérations sont possibles, l'addition et la multiplication :
 - ▣ **L'opérateur d'addition +** concatène (assemble) deux chaînes de caractères.
 - ▣ **L'opérateur de multiplication *** entre un nombre entier et une chaîne de caractères duplique (répète) plusieurs fois une chaîne de caractères
- **Exemple**

```
>>> chaine = "Salut"
>> chaine
'Salut '
>>> chaine + " Python "
'Salut Python '
>>> chaine * 3
'SalutSalutSalut'
```
- **Remarque :** les opérateurs **+** et ***** se comportent différemment s'il s'agit d'entiers ou de chaînes de caractères : $2 + 2$ est une addition alors que $"2" + "2"$ est une concaténation. On appelle ce comportement redéfinition des opérateurs.

□ Les expressions logiques simples

- **Python** est capable d'effectuer toute une série de **comparaisons** entre le contenu de deux **variables**, telles que :

Syntaxe Python	Signification
==	égal à
!=	différent de
>	strictement supérieur à
>=	supérieur ou égal à
<	strictement inférieur à
<=	inférieur ou égal à

□ Exemple:

```
>>> x = 5
>>> x == 5
True
>>> x > 10
False
>>> x < 10
True
```

Remarques:

- Les objets d'une comparaison doivent être du même type.
- La comparaison entre les objets de type alphanumérique s'effectue en fonction de l'ordre alphabétique des caractères, cet ordre est établi selon le code ASCII.

□ Remarque :

- On peut également effectuer des comparaisons sur des chaînes de caractères.
- On peut aussi utiliser les opérateurs `<`, `>`, `<=` et `>=`. Dans ce cas, l'ordre alphabétique est pris en compte
- La comparaison entre les objets de type alphanumérique s'effectue en fonction de l'ordre alphabétique des caractères, cet ordre est établi selon le code ASCII.

□ Exemple:

```
>>> animal = "tigre"
```

```
>>> animal == "tig"
```

```
False
```

```
>>> animal != "tig"
```

```
True
```

```
>>> animal == "tigre"
```

```
True
```

□ Exemple:

```
>>> "a" < "b"
```

```
#"a" est inférieur à "b" car le caractère a est situé avant le caractère b dans  
#l'ordre alphabétique.
```

```
True
```

```
>>> "ali" < "alo"
```

```
True
```

```
>>> "abb" < "ada"
```

```
True
```

□ Les expressions logiques composées

- En **Python**, on utilise le mot réservé **and** pour l'opérateur **ET** et le mot réservé **or** pour l'opérateur **OU**.

- **Remarque:** En Python, les opérateurs **and** et **or** s'écrivent en **minuscule**:

□ Exemple :

```
>>> x = 2
>>> y = 2
>>> x == 2 and y == 2:
True
```

- On peut utiliser l'opérateur logique de négation **not** qui inverse le résultat d'une condition :

□ Exemple:

```
>>> not True
False
```


- En **Python**, la fonction **type()** est utilisée pour obtenir le type d'un objet. Elle prend **un seul argument** et renvoie **le type de cet argument** :

```
>>> x = 2
>>> type (x)
<class 'int '>
>>> y = 2.0
>>> type (y)
<class 'float '>
>>> z = '2'
>>> type (z)
<class 'str '>
```

- **Remarque:** Pour Python, la valeur 2 (nombre entier) est différente de 2.0 (float) et est aussi différente de '2' (chaîne de caractères).

- En **programmation**, on est souvent amené à **convertir** les types, c'est-à-dire passer d'un type numérique à une chaîne de caractères ou vice-versa.
- En **Python**, on peut utiliser les fonctions **int()**, **float()** et **str()** pour convertir le type d'un objet passer en paramètre

□ **Exemple :**

```
>>> i = 3
>>> str (i)
'3'
>>> i = '456 '
>>> int (i)
456
>>> float (i)
456.0
>>> i = '3.1416 '
>>> float (i)
3.1416
```

□ **Remarque:**

- Lorsqu'on lit ou écrit des nombres dans un fichier, ils sont considérés comme du texte, donc des chaînes de caractères.
- Toute conversion d'une variable d'un type en un autre est appelé casting en anglais

- Pour afficher **leur valeur à l'écran**, il existe deux possibilités:
 - ▣ La première consiste à entrer au clavier le nom de la variable, puis Enter. Python répond en affichant la valeur correspondante :
 - ▣ La fonction **print()** est utilisée pour afficher des valeurs à l'écran. Vous pouvez lui passer une ou plusieurs valeurs séparées par des virgules, et elles seront affichées à la suite.
- **Remarque :** À l'intérieur d'un programme, vous utiliserez toujours la fonction **print()**
- **Exemple:**

```
>>> n = 7
>>> msg = "Bonjour"
>>> pi = 3.14159
>>> n
7
>>> msg
'Bonjour'
>>> pi
3.14159
```
- **Exemple:**

```
>>> print(msg)
Bonjour
>>> print(n)
7
>>> print(n,msg,pi)
7 Bonjour 3.14159
```

❑ Prise en main des f-strings:

- ❑ L'écriture formatée est un mécanisme permettant d'afficher des variables avec un certain format, par exemple justifiées à gauche ou à droite, ou encore avec un certain nombre de décimales pour les floats. L'écriture formatée est incontournable lorsqu'on veut créer des fichiers organisés
- ❑ Depuis la version 3.6, Python a introduit les f-strings pour mettre en place l'écriture formatée
- ❑ Les f-strings permettent une meilleure organisation de l'affichage des variables.

❑ Exemple:

```
x = 32
nom = "Ahmed"
print(f"{nom} a {x} ans")
```

```
C:\Users\DELL\Desktop\A
Ahmed a 32 ans
```

```
prop_GC = (4500 + 2575) / 14800
print("La proportion de GC est", prop_GC)
print(f"La proportion de GC est {prop_GC:.2f}")
print(f"La proportion de GC est {prop_GC:.3f}")
```

```
La proportion de GC est 0.4780405405405405
La proportion de GC est 0.48
La proportion de GC est 0.478
```

- L'instruction **input()** permet de **lire** des données saisies au clavier et les enregistrer dans des variables appropriées.

□ Exemple:

```
>>> nom_utilisateur = input("Entrez votre nom : ")
>>> print("Bonjour,", nom_utilisateur)
```

□ Exemple

```
>>> nom_utilisateur = input("Entrez votre nom : ")
>>> age_utilisateur = input("Entrez votre âge : ")
>>> print("Nom:", nom_utilisateur, "Âge:", age_utilisateur)
```

- Les **tests** sont un élément essentiel à tout langage informatique si on veut lui donner un peu de complexité car ils permettent à l'ordinateur de prendre des décisions. Pour cela, **Python** utilise **l'instruction if** :

- **Exemple :**

```
>>> x = 2
>>> if x == 2:
...     print (" Le test est vrai !")
...
Le test est vrai !
```

- **Remarques:**

- ▣ Les **blocs d'instructions** dans les tests doivent forcément être **indentés**.
- ▣ **L'indentation** indique la **portée** des instructions à exécuter si le test est vrai.
- ▣ la ligne qui contient l'instruction **if** se termine par le caractère deux-points « **:** ».

- **Exemple :**

```
>>> x = " souris "
>>> if x == " tigre ":
...     print (" Le test est vrai !")
...
```

- Parfois, il est pratique de tester si la **condition** est **vraie** ou si elle est **fausse** dans une même **instruction if**.
- Plutôt que d'utiliser deux **instruction if**, on peut se servir des instructions **if** et **else**
- **Exemple:**

```
>>> x = 2
>>> if x == 2:
... print (" Le test est vrai !")
... else :
... print (" Le test est faux !")
...
Le test est vrai !
```
- **Exemple:**

```
>>> x = 3
>>> if x == 2:
... print (" Le test est vrai !")
... else :
... print (" Le test est faux !")
...
Le test est faux !
```
- **Remarque :** On peut utiliser une série de tests dans la même instruction **if**, notamment pour tester plusieurs valeurs d'une même variable.

□ **Exemple** : Écrivons les instructions qui permettent de demander une valeur à l'utilisateur, puis de déterminer si cette valeur est paire ou impaire

□ **Soution:**

```
a = input("donner une valeur : ");  
    if (float(a) % 2 == 0):  
        print("a est pair")  
        print("parce que le reste de sa division par 2 est nul")  
else:  
    print("a est impair")
```


- Il est possible **d'imbriquer** les structures alternatives les unes dans les autres, de manière à réaliser des structures de décision complexes.

- **Exemple :**

```
a = 5
b = 10
if a > 0:
    print("a est positif.")
    if b > 0:
        print("b est aussi positif.")
    else:
        print("b est négatif.")
elif a == 0:
    print("a est nul.")
else:
    print("a est négatif.")
    if b > 0:
        print("b est positif.")
    elif b == 0:
        print("b est nul.")
    else:
        print("b est négatif.")
```

- En algorithmique une structure itérative est tout simplement **une boucle**, c'est-à-dire une **répétition d'instructions**.
- On l'utilise souvent quand on doit **exercer plusieurs fois** le **même traitement** sur un même objet.
- Mais son réel intérêt réside dans le fait que l'on peut modifier, à chaque répétition, les objets sur lesquels s'exerce l'action répétée.
- Alors dans une boucle on a :
 - ▣ Une instruction ou un ensemble d'instructions à répéter.
 - ▣ Cette répétition doit avoir un arrêt.
 - ▣ L'arrêt de cette répétition dépend d'une condition.
 - ▣ Cette condition est appelé **condition d'arrêt**.
- Python propose deux instructions particulières pour construire des boucles :
 - ▣ l'instruction **for ... in ...**
 - ▣ l'instruction **while**

□ Syntaxe:

while **condition**:

Instructions à exécuter tant que la condition est vraie

Ces instructions seront répétées tant que la condition est vraie

- Cette **instruction** indique à **Python** qu'il lui faut répéter continuellement le bloc d'instructions qui suit, tant que la **condition** est vrai.
- L'instruction **while** amorce une **instruction composée**.
- Le double point à la fin de la ligne introduit le bloc d'instructions à répéter, lequel doit obligatoirement se trouver en retrait.
- **Principe** :
 - ▣ Avec l'instruction while, Python commence par évaluer la validité de la **condition** fournie entre parenthèses
 - ▣ Si la **condition** se révèle fausse, alors tout le bloc qui suit est ignoré et l'exécution du programme se termine.
 - ▣ Si la **condition** est vraie, alors Python exécute tout le bloc d'instructions constituant le corps de la boucle
- **Remarque** : lorsque ces instructions ont été exécutées, nous avons à une première itération

36

❑ **Exemple:** Ecrivons un programme python qui *demande à l'utilisateur de saisir des valeurs entières jusqu'à ce que la somme dépasse 100, puis affiche la somme totale.*

❑ **Solution:**

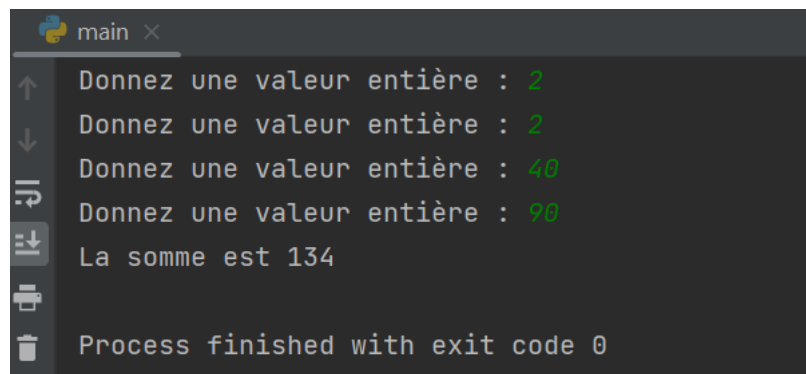
```
somme = 0
```

```
while somme <= 100:
```

```
    valeur = int(input("Donnez une valeur entière : "))
```

```
    somme += valeur
```

```
print("La somme est", somme)
```



```
main x
↑
Donnez une valeur entière : 2
↓
Donnez une valeur entière : 2
↻
Donnez une valeur entière : 40
↻
Donnez une valeur entière : 90
↻
La somme est 134
Process finished with exit code 0
```

□ Syntaxe:

for **element** **in** **sequence**:

Instructions à exécuter pour chaque élément de la séquence

- Cette boucle est utilisée pour itérer sur une séquence (comme une liste, un tuple, un dictionnaire, etc.) ou sur un objet itérable (comme un objet générateur).

- **Exemple** : écrire un programme qui permet de calculer le nombre d'espace dans la phrase suivante "Bonjour mes élèves ingénieurs"

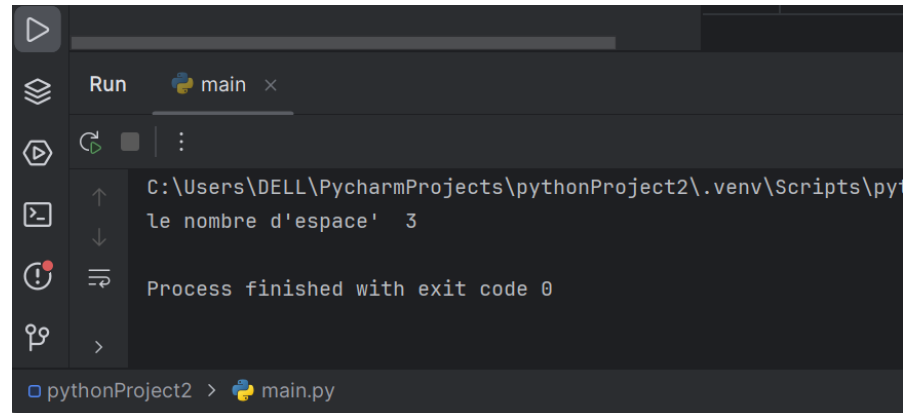
□ Solution

```
ch = "Bonjour mes élèves ingénieurs"
```

```
n=0
```

```
for c in ch:  
    if c==' '  
        n+=1
```

```
print( "le nombre d'espace ", n)
```



The screenshot shows a Python IDE interface. At the top, there's a 'Run' button and a tab labeled 'main'. Below that, the command prompt shows the execution of a Python script: 'C:\Users\DELL\PycharmProjects\pythonProject2\.venv\Scripts\python.exe main.py'. The output of the script is 'le nombre d'espace' 3'. At the bottom, a status bar indicates 'Process finished with exit code 0'.

- **Les chaînes de caractères:** Sous Python, une donnée de type **string** est une suite quelconque de caractères délimitée soit par des **apostrophes** (guillemets simples), soit par **des guillemets** (guillemets doubles), soit **par des guillemets triples** (" ou """). Les chaînes de caractères peuvent être considérées comme des listes (de caractères) un peu particulières :

- **Exemple :**

```
ch="Bonjour tout le monde"
print(ch)
print(ch[0])
print("la taille de cette chaine est",
len(ch))
print(ch[8:])
```

```
Bonjour tout le monde
B
la taille de cette chaine est 21
tout le monde
```

- **len(ch)** permet de déterminer la taille d'une chaîne.
- Pour générer des **sous-chaînes**, on utilise l'opérateur **d'indexation** suivant : **[i:j]**.
- Pour **concaténer** deux chaînes de caractères on utilise l'opérateur **+**.

- **Exemple :**

```
ch1="Bonjour "
ch2="tout le monde"
print(ch1+ch2)
```

- Sous **Python**, les **chaînes de caractères** sont des **objets**. On peut donc effectuer de nombreux traitements dessus en utilisant des méthodes appropriées.
- En voici quelques-unes, choisies parmi les plus utiles. Mais vous pouvez obtenir la liste complète de toutes les méthodes associées à un objet à l'aide de la fonction intégrée **dir()** ou **help(str)** :
 - ▣ **index(c)** : retrouve l'index de la première occurrence du caractère "c" dans la chaîne.
 - ▣ **find(s_ch)** : cherche la position d'une sous-chaîne dans la chaîne.
 - ▣ **count(s_ch)** : compte le nombre de sous-chaînes dans la chaîne.
 - ▣ Pour rechercher/remplacer, nous avons à notre disposition les méthodes **count**, **find** et **replace**, à savoir « **compter** », « **rechercher** » et « **remplacer** ».

- **Exemple :**

```
ch="Bonjour tout le monde, nous sommes le 27/02/2024"
print(ch.index('j'))
print(ch.find("monde"))
print(ch.count('le'))
```

```
3
16
2
```

- Une **liste** est une structure de données qui contient **une série de valeurs**.
- Une **liste** est déclarée par une série de valeurs séparées par des **virgules**, et le tout encadré par des crochets `[]`.
- **Exemple:**

```
animaux = ["girafe", "tigre", "singe", "souris"]
tailles = [5, 2.5, 1.75, 0.15]
```
- **Python** autorise la construction de liste contenant **des valeurs de types différents** (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité.
- **Exemple:**

```
jour = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 476, 3.14]
```


- Les éléments qui constituent une **liste** peuvent être de types variés. L'accès par indice `maListe[index]`

□ **Exemple:**

```
jour = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 476, 3.14]  
print(jour[0])  
print(jour[5])
```

```
lundi  
476
```

- Les **tranches**: **découpage** (*slicing*) permet de dégager une sous-liste en précisant deux index correspondant aux bornes de la plage.

□ **Exemple:**

```
jour = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 476, 3.14]  
print(jour[2:6])
```

```
['mercredi', 'jeudi', 'vendredi', 476]
```

- Tout comme les chaînes de caractères, les listes supportent l'opérateur **+** de concaténation, ainsi que l'opérateur ***** pour la duplication

- **Exemple:**

```
jour1 = ['lundi', 'mardi', 'mercredi']
jour2 = ['jeudi', 'vendredi']
jour3 = ["samedi", "dimanche"]
jours = jour1 + jour2 + jour3
print(jours)
```

```
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
```

- **Remarque :** On peut utiliser la méthode **.append()** lorsque on souhaite ajouter un seul élément à la fin d'une liste.

```
jours = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', "samedi"]
jours.append("dimanche")
print(jours)
```

```
jours = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', "samedi"]
jours = jours + ["dimanche"]
print(jours)
```

- La **liste** peut également être indexée avec **des nombres négatifs** selon le modèle suivant :

liste : jours = ['lundi', 'mardi', 'mercredi', 'jeudi' , 'vendredi', 'samedi', 'dimanche']

indice positif : 0 1 2 3 4 5 6

indice négatif : -7 -6 -5 -4 -3 -2 -1

- **Exemple:**

```
jours = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']  
print(jours[1], " = ", jours[-6])
```

```
mardi = mardi
```

- **Remarque:**

- ▣ Les **indices négatifs** reviennent à compter à partir de la fin.
- ▣ Leur principal avantage est qu'on peut accéder au dernier élément d'une liste à l'aide de l'indice **-1** sans pour autant connaître la longueur de la liste.

- ❑ **La tranche de la liste** : Un autre avantage des listes est la possibilité de sélectionner une partie d'une liste en utilisant un indigage construit sur le modèle

[**m**:**n+1**]

- ❑ Pour récupérer tous les éléments, du l'élément **m** inclus à l'élément **n+1** exclu.
- ❑ On dit alors qu'on récupère une tranche de la liste.

- ❑ **Exemple :**

```
jours = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
print(jours[0:2])
print(jours[0:3])
print(jours[0:])
print(jours[:])
print(jours[1:])
print(jours[1:-1])
```

```
['lundi', 'mardi']
```

```
['lundi', 'mardi', 'mercredi']
```

```
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
```

```
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
```

```
['mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
```

```
['mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi']
```

❑ Remarque :

- ❑ Lorsqu'aucun **indice** n'est indiqué à gauche ou à droite du symbole deux-points, Python prend par défaut tous les éléments depuis **le début** ou tous les éléments jusqu'à **la fin** respectivement.
- ❑ On peut aussi préciser le **pas** en ajoutant un symbole **deux-points supplémentaire** et en indiquant le **pas** par un entier.

❑ Exemple:

```
jours = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
print(jours)
print(jours[0:3:2])
print(jours[:1])
print(jours[:2])
print(jours[:3])
print(jours[1:6:3])
```

```
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
['lundi', 'mercredi']
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
['lundi', 'mercredi', 'vendredi', 'dimanche']
['lundi', 'jeudi', 'dimanche']
['mardi', 'vendredi']
```

- **L'instruction `len()`** vous permet de connaître **la longueur d'une liste**, c'est-à-dire le nombre d'éléments que contient la liste.

```
len( list1)
```

- **Exemple:**

```
jours = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']  
print("la longueur d'une liste est ", len(jours))
```

```
la longueur d'une liste est 7
```

- **L'instruction `range()`** fonctionne sur le modèle

```
range([début,] fin[, pas])
```

- **L'instruction `range()`** est une fonction spéciale en Python qui génère des nombres entiers compris dans un intervalle.
- Lorsqu'elle est utilisée en **combinaison** avec la fonction `list()`, on obtient une **liste d'entiers**.

□ Exemple:

```
print(list(range(10)))  
print(list(range(0, 5)))  
print(list(range(15, 20)))  
print(list(range(0, 1000, 200)))  
print(list(range(2, -2, -1)))  
print(list(range(10, 0)))  
print(list(range(10, 0, -1)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
[0, 1, 2, 3, 4]  
[15, 16, 17, 18, 19]  
[0, 200, 400, 600, 800]  
[2, 1, 0, -1]  
[]  
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

- Les fonctions **min()**, **max()** et **sum()** renvoient respectivement le **minimum**, le **maximum** et la **somme** d'une **liste passée en argument**.

- **Exemple:**

```
liste = list(range(10))  
print(liste)  
print( "la somme est",sum(liste))  
print("le min est ", min(liste))  
print("le max est ",max(liste))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
la somme est 45  
le min est 0  
le max est 9
```


- ❑ **Listes de listes:** il est possible de construire des listes de listes.
- ❑ Cette fonctionnalité peut parfois être très pratique.
- ❑ Pour accéder à un élément de la liste, on utilise **l'indigage habituel**
- ❑ Pour accéder à un élément de la sous-liste, on utilise **un double indigage**
- ❑ **Exemple :**

```
jour1=["lundi", 1]
jour2=["mardi", 2]
jour3=["mercredi", 3]
jours=[jour1, jour2, jour3]
print(jours)
print(jours[1])
print(jours[1][1])
```

```
[['lundi', 1], ['mardi', 2], ['mercredi', 3]]
['mardi', 2]
2
```

❑ Méthodes associées aux listes

- ❑ les listes possèdent de nombreuses **méthodes** qui leur sont propres et qui peuvent se révéler très pratiques :

- ❑ **.insert()**: La méthode .insert() **insère** un objet dans une liste à un **indice déterminé**.

```
a = [1, 2, 3]
print(a)
a.insert(2, -15)
print(a)
```

```
[1, 2, 3]
[1, 2, -15, 3]
```

- ❑ **del**: L'instruction del **supprime** un élément d'une liste à **un indice déterminé** :

```
a = [1, 2, 3]
print(a)
del a[2]
print(a)
```

```
[1, 2, 3]
[1, 2]
```

- ❑ **.remove()**: La méthode .remove() **supprime** un élément d'une liste à partir de sa **valeur** :

```
a = [1, 2, 4, 6, 3]
print(a)
a.remove(3)
print(a)
```

```
[1, 2, 4, 6, 3]
[1, 2, 4, 6]
```

- **.sort()** : La méthode `.sort()` **trie les éléments** d'une liste **du plus petit au plus grand**. L'argument **`reverse=True`** spécifie le tri inverse, c'est-à-dire du plus grand au plus petit élément :

```
a = [1, 2, 4, 6, 3]
print(a)
a.sort()
print(a)
a.sort(reverse=True)
print(a)
```

```
[1, 2, 4, 6, 3]
[1, 2, 3, 4, 6]
[6, 4, 3, 2, 1]
```

- **sorted()** : La fonction `sorted()` **trie également une liste**. Contrairement à la méthode précédente `.sort()`, cette fonction renvoie **la liste triée** et ne modifie pas la liste initiale (La fonction **`sorted()`** supporte aussi l'argument **`reverse=True`**)

```
a = [1, 2, 4, 6, 3]
print(a)
print(sorted(a))
```

```
[1, 2, 4, 6, 3]
[1, 2, 3, 4, 6]
```

- **.reverse():** La méthode `.reverse()` **inverse** une liste :

```
a = [1, 2, 4, 6, 3]
print(a)
a.reverse()
print(a)
```

```
[1, 2, 4, 6, 3]
[3, 6, 4, 2, 1]
```

- **.count():** La méthode `.count()` **compte le nombre** d'éléments (**passés en argument**) dans une liste :

```
a = [1, 2, 4, 6, 3, 1, 4, 1]
print(a)
print(a.count(1))
```

```
[1, 2, 4, 6, 3, 1, 4, 1]
3
```

- **Test d'appartenance :** L'opérateur `in` teste si un élément fait partie d'une liste.

```
a = [1, 2, 4, 6, 3, 1, 4, 1]
print(3 in a)
```

- **Remarque :** on peut directement utiliser la fonction `list()` qui prend n'importe quel **objet séquentiel** (liste, chaîne de caractères, etc.) et qui renvoie une liste :

```
seq = "CAAAGGTAACGC"
print(list(seq))
```

```
['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
```

- ❑ **Liste de compréhension** représente une manière originale et très puissante de **générer des listes**.
- ❑ La syntaxe de base consiste au moins en une boucle **for** au sein de crochets précédés **d'une variable** (qui peut être la variable d'itération ou pas)

```
a = [i for i in range(10)]  
b = [2 for i in range(10)]  
print(a)  
print(b)
```

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

- ❑ **Exemple** : créer une liste qui contient les nombres de 0 à 30 (inclus) qui sont pairs.
- ❑ **Solution**:

```
a=[i for i in range(31) if i % 2 == 0]  
print(a)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
```

- Les **dictionnaires** se révèlent très pratiques lorsque vous devez manipuler des **structures complexes** à décrire et que les listes présentent leurs limites.
- Les **dictionnaires** sont des **collections non ordonnées** d'objets
- Il ne s'agit pas d'objets séquentiels comme les listes ou chaînes de caractères, mais plutôt d'objets dits de **correspondance** (**mapping objects**) ou **tableaux associatifs**. En effet, on accède aux valeurs d'un **dictionnaire** par des **clés**.
- **Exemple:**

```
personne = {}  
personne['nom'] = 'ahemd'  
personne['age'] = 31  
personne['profession'] = 'Ingénieur'  
  
print("Dictionnaire:", personne)
```

```
personne = {'nom': 'ahemd', 'age': 31, 'profession': 'Ingénieur'}
```

```
Dictionnaire: {'nom': 'ahemd', 'age': 31, 'profession': 'Ingénieur'}
```

- On peut ajouter une clé et une valeur supplémentaire : `personne['ville']='al hoceima'`

- **Itération sur les clés pour obtenir les valeurs** : Si on souhaite voir toutes les associations **clés / valeurs**, on peut itérer sur un dictionnaire de la manière suivante :

```
personne = {'nom': 'ahemd', 'age': 31, 'profession': 'Ingénieur'}
personne['ville'] = 'al hoceima'
for key in personne:
    print(key, " : ", personne[key])
```

```
nom : ahemd
age : 31
profession : Ingénieur
ville : al hoceima
```

- ❑ Les méthodes **.keys()** et **.values()** renvoient les **clés** et les **valeurs** d'un **dictionnaire**:
- ❑ il existe la méthode **.items()** qui renvoie **un nouvel objet dict_items** :

```
personne = {'nom': 'ahemd', 'age': 31, 'profession': 'Ingénieur'}  
print(personne.keys())  
print(personne.values())  
print(personne.items())
```

```
dict_keys(['nom', 'age', 'profession'])  
dict_values(['ahemd', 31, 'Ingénieur'])  
dict_items([('nom', 'ahemd'), ('age', 31), ('profession', 'Ingénieur')])
```

- ❑ **Remarque:** **dict_keys**, **dict_values** et **dict_items** ne sont pas **indexables** (on ne peut pas **retrouver** un élément par indice, par exemple dico.keys()[0] renverra une erreur).

- ❑ **Existence d'une clé ou d'une valeur** : Pour vérifier si une **clé existe** dans un **dictionnaire**, on peut utiliser le **test d'appartenance** avec l'opérateur **in** qui renvoie un **booléen** :
- ❑ **Exemple:**

```
personne = {'nom': 'ahemd', 'age': 31, 'profession': 'Ingénieur'}  
if 'nom' in personne:  
    print("la cle nom existe")
```

```
C:\Users\DELL\Desktop\T  
la cle nom existe
```

```
personne = {'nom': 'ahemd', 'age': 31, 'profession': 'Ingénieur'}  
print('nom' in personne.keys())
```

- ❑ La méthode `.get()` extrait la valeur associée à une **clé** mais ne **renvoie** pas **d'erreur** si la clé n'existe pas.
- ❑ On peut également indiquer à `.get()` une **valeur par défaut** si la clé n'existe pas
- ❑ **Exemple:**

```
personne = {'nom': 'ahemd', 'age': 31, 'profession': 'Ingénieur'}  
print(personne.get('nom'))  
print(personne.get('ville'))  
print(personne.get('ville', "la valeur n'exsiste pas "))
```

```
ahemd  
None  
la valeur n'exsiste pas
```

- ❑ **Tri par clés :** On peut utiliser la fonction `sorted()` vue précédemment avec les listes pour trier un dictionnaire par ses clés :

```
personne = {'nom': 'ahemd', 'age': 31, 'profession': 'Ingénieur'}  
print(sorted(personne))
```

```
['age', 'nom', 'profession']
```

- ❑ **Tri par valeurs** : Pour **trier** un **dictionnaire** par ses **valeurs**, il faut utiliser la fonction **sorted** avec l'argument **key** :
- ❑ L'argument **key=dico.get** indique qu'il faut réaliser le **tri** par les **valeurs** du dictionnaire:

```
personne = {'age1': 40, 'age2': 31, 'age3': 35}
print(sorted(personne, key=personne.get))
for key in sorted(personne, key=personne.get):
    print(key, personne[key])
```

```
['age2', 'age3', 'age1']
age2 31
age3 35
age1 40
```

- ❑ l'argument **reverse=True** fonctionne également :

```
personne = {'age1': 40, 'age2': 31, 'age3': 35}
for key in sorted(personne, key=personne.get, reverse=True):
    print(key, personne[key])
```

- ❑ **Les fonctions min() et max()** acceptent également l'argument **key=**. On peut ainsi obtenir la clé associée au **minimum** ou au **maximum** des valeurs d'un **dictionnaire** :

```
personne = {'age1': 40, 'age2': 31, 'age3': 35}
print(max(personne, key=personne.get))
print(min(personne, key=personne.get))
```

```
age1
age2
```

- ❑ **Liste de dictionnaires** : En créant une liste de dictionnaires qui possèdent les mêmes clés, on obtient une **structure** qui ressemble à une base de données :

```
personne = [{'nom': 'mohamed', 'age': 21, 'profession': 'Ingénieur'},  
            {'nom': 'ahmed', 'age': 31, 'profession': 'Technicien'}]  
print(personne)
```

```
[{'nom': 'mohamed', 'age': 21, 'profession': 'Ingénieur'}, {'nom': 'ahmed', 'age': 31, 'profession': 'Technicien'}]
```

- ❑ La **fonction dict()** va convertir **l'argument** qui lui est passé en **dictionnaire**. Il s'agit donc d'une **fonction de casting**.
- ❑ L'argument qui lui est passé doit avoir une forme particulière : un **objet séquentiel** contenant **d'autres objets séquentiels de 2 éléments**. Par exemple, une liste de listes de **2 éléments** :

```
jours = [{"lundi", 1}, {"mardi", 2}, {"mercredi", 3}, {"jeudi", 4}]  
print(dict(jours))
```

```
{'lundi': 1, 'mardi': 2, 'mercredi': 3, 'jeudi': 4}
```

- ❑ **Tuples** sont des **objets séquentiels** correspondant aux **listes** (itérables, ordonnés et **indexables**) mais ils sont toutefois **non modifiables**.
- ❑ L'intérêt des **tuples** par rapport aux listes réside dans **leur immutabilité**.
- ❑ Cela, accélère considérablement la manière dont **Python** accède à chaque élément et ils prennent moins de place en mémoire.
- ❑ On utilise les **parenthèses** au lieu des **crochets** pour les créer

```
t = (1, 2, 3)
print(t)
print(type(t))
print(t[2])
print(t[0:2])
```

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

- ❑ **L'affectation et l'indigage** fonctionnent comme avec les listes.
- ❑ Si on essaie de **modifier** un des **éléments** du tuple, Python renvoie un message d'erreur.
- ❑ **Remarque** : Si vous voulez ajouter un élément (ou le modifier), vous devez créer un nouveau tuple :

□ Exemple:

```
t = (1, 2, 3)
print(t)
print(id(t))
t = t + (2,)
print(t)
print(id(t))
```

```
(1, 2, 3)
2426163306368
(1, 2, 3, 2)
2426162987632
```

□ Remarque :

- ▣ Les **opérateurs** **+** et ***** fonctionnent comme pour les listes (**concaténation** et **duplication**) :
- ▣ On peut utiliser la fonction `tuple(sequence)`. C'est-à-dire qu'elle prend en argument un **objet** de type **container** et renvoie le **tuple correspondant** (**opération de casting**)

□ Exemple:

```
print(tuple([1, 2, 3]))
print(tuple("ATGCCGCGAT"))
```

```
(1, 2, 3)
('A', 'T', 'G', 'C', 'C', 'G', 'C', 'G', 'A', 'T')
```

- ❑ Les objets de type `set` représentent un **autre type de container** qui peut se révéler très pratique.
- ❑ Ils ont la particularité d'être **modifiables, non ordonnés, non indexables** et de ne contenir qu'une **seule copie maximum** de chaque élément. Pour créer un nouveau `set` on peut utiliser **les accolades** :

```
s = {4, 5, 5, 12}
print(s)
print(type(s))
```

```
{5, 12, 4}
<class 'set'>
```

- ❑ La fonction interne à Python `set()` convertit **un objet itérable** passé en argument en un **nouveau set** (**opération de casting**) :

```
print(set([1, 2, 4, 1]))
print(set((2, 2, 2, 1)))
print(set(range(5)))
```

```
{1, 2, 4}
{1, 2}
{0, 1, 2, 3, 4}
```

❑ Remarque :

- ❑ Il est **impossible** de récupérer un **élément** par sa **position**. Il est également impossible de **modifier** un de ses **éléments** par **l'indexation**.

- ❑ les **sets** ne supportent pas les opérateurs **+** et *****.

❑ Les **sets** ne peuvent être modifiés que par des méthodes spécifiques.

- ❑ La méthode **.add()** **ajoute** au set **l'élément passé en argument**. Si l'élément est déjà présent dans le set, il n'est pas ajouté puisqu'on a au plus une copie de chaque élément.
- ❑ La méthode **.discard()** **retire** du set **l'élément passé en argument**. Si l'élément n'est pas présent dans le set, il ne se passe rien, le set reste intact.

❑ Comme les **sets** ne sont pas **ordonnés** ni **indexables**, il n'y a pas de méthode pour **insérer** un élément à une position précise contrairement aux listes.

❑ Exemple :

```
s = set(range(5))
print(s)
s.add(4)
print(s)
s.add(472)
print(s)
s.discard(0)
print(s)
```

```
{0, 1, 2, 3, 4}
{0, 1, 2, 3, 4}
{0, 1, 2, 3, 4, 472}
{1, 2, 3, 4, 472}
```


- ❑ Les containers de type **set** sont très utiles pour **rechercher** les **éléments uniques** d'une **suite d'éléments**. Cela revient à éliminer tous les doublons.
- ❑ **Exemple**: Ecrire un compteur de lettres en combinaison avec une liste de compréhension, le tout en une ligne !

```
seq = "atctcgatcgatcgcgctagctagctcgccatacgtacgactacgt"  
print(set(seq))  
print( [(base, seq.count(base)) for base in set(seq)] )
```

```
{'t', 'c', 'a', 'g'}  
[('t', 11), ('c', 15), ('a', 10), ('g', 10)]
```

- ❑ La méthode **.issubset()** indique si un **set** est **inclus** dans un autre **set**.
- ❑ La méthode **isdisjoint()** indique si un **set** est **disjoint** d'un autre **set**, c'est-à-dire, s'ils n'ont aucun élément en commun indiquant que leur intersection est nulle. **Exemple** :

```
s1 = set(range(10))  
s2 = set(range(3, 7))  
s3 = set(range(15, 17))  
print(s1)  
print(s2)  
print(s3)  
print( s2.issubset(s1) )  
print(s3.isdisjoint(s1))
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}  
{3, 4, 5, 6}  
{16, 15}  
True  
True
```

- ❑ En programmation, les **fonctions** sont très utiles pour **réaliser plusieurs fois** la même opération au **sein d'un programme**.
- ❑ Elles rendent également le code plus **lisible** et plus **clair** en le fractionnant en blocs logiques.
- ❑ Une fonction est un sous-programme constitué d'une suite d'instructions indépendantes.
- ❑ Elle prend éventuellement quelques paramètres et retourne un résultat.
- ❑ Une fonction se caractérise par son **nom** et le **type** de la **valeur retournée**.

- Pour définir une **fonction**, **Python** utilise le mot-clé **def**.
- Si on souhaite que la fonction renvoie quelque chose, il faut utiliser le mot-clé **return**.

- **Exemple:**

```
def carre(x):  
    return x**2  
print(carre(2))
```

- De même que pour les boucles et les tests, l'**indentation** de ce **bloc d'instructions** (qu'on appelle le corps de la fonction) est **obligatoire**.

- Une particularité des **fonctions** en **Python** est que vous **n'êtes pas obligé** de préciser le **type** des **arguments**.
- Python est en effet connu comme étant un langage au **typage dynamique**, c'est-à-dire qu'il reconnaît le type des variables au moment de l'exécution.

```
def fois(x, y):  
    return x * y  
  
print(fois(2, 3))  
print(fois(3.1415, 5.23))  
print(fois("to", 2))  
print(fois([1, 3], 2))
```

```
6  
16.4300450000000003  
toto  
[1, 3, 1, 3]
```

- Lorsqu'on définit une fonction `def fct(x, y):` les arguments `x` et `y` sont appelés **arguments positionnels** (en anglais positional arguments).
- Il est strictement obligatoire de les **préciser** lors de **l'appel** de la **fonction**.
- De plus, il est nécessaire de **respecter** le même ordre lors de **l'appel** que dans la **définition** de la fonction.

- Un énorme avantage en **Python** est que les **fonctions** sont capables de **renvoyer plusieurs objets** à la fois.

- **Exemple :**

```
def carre_cube(x):  
    return x ** 2, x ** 3  
  
print(carre_cube(2))
```

```
(4, 8)
```

- En réalité **Python** ne renvoie qu'un seul objet, mais celui-ci peut être **séquentiel**, c'est-à-dire contenir lui même d'autres objets.
- Dans notre exemple **Python** renvoie un objet de type **tuple**
- Il pourrait de renvoyer une liste :

- **Exemple :**

```
def carre_cube2(x):  
    return [x ** 2, x ** 3]  
  
print(carre_cube2(2))
```

```
[4, 8]
```

- Un argument défini avec une syntaxe `def fct(arg=val):` est appelé **argument par mot-clé** (en anglais keyword argument).
- Le passage d'un tel **argument** lors de l'appel de la fonction est **facultatif**.
- Ce type d'argument ne doit pas être **confondu** avec les arguments positionnels, dont la syntaxe est `def fct(arg):`
- **Exemple:**
- Il est aussi possible de passer un ou plusieurs **argument(s)** de manière facultative et de leur attribuer une **valeur** par **défaut**
- Il est bien sûr possible de passer plusieurs arguments par mot-clé :

```
def fct(x=1):  
    return x  
  
print(fct())  
print(fct(10))
```

```
1  
10
```

```
def fct(x=0, y=0, z=0):  
    return x, y, z  
  
print(fct())  
print(fct(10))  
print(fct(10, 8))  
print(fct(10, 8, 3))
```

```
(0, 0, 0)  
(10, 0, 0)  
(10, 8, 0)  
(10, 8, 3)
```

- On peut préciser l'**argument** par **mot-clé** `z` et **garder** les valeurs des autres **arguments** par **défaut**
- Python permet même de rentrer les arguments par mot-clé dans un ordre arbitraire :

□ **Exemple:**

```
def fct(x=0, y=0, z=0):  
    return x, y, z  
  
print(fct(z=10))  
print(fct(z=10, x=3, y=80))  
print(fct(z=10, y=80))
```

```
(0, 0, 10)  
(3, 80, 10)  
(0, 80, 10)
```

- **Remarque :** les arguments positionnels doivent toujours être placés avant les arguments par mot-clé :

```
def fct(a, b, x=0, y=0, z=0):  
    return a, b, x, y, z  
  
print(fct(1, 1))  
print(fct(1, 1, z=5))  
print(fct(1, 1, z=5, y=32))
```

```
(1, 1, 0, 0, 0)  
(1, 1, 0, 0, 5)  
(1, 1, 0, 32, 5)
```

□ Remarque :

- L'utilisation **d'arguments par mot-clé** permet de **modifier** le **comportement** par **défaut** de nombreuses fonctions.
- Par exemple, si on souhaite que la fonction `print()` n'affiche pas un retour à la ligne, on peut utiliser l'argument `end`

```
print("Message ", end="")  
print(":")
```

Message :

- Lorsqu'on manipule des fonctions, il est essentiel de bien comprendre comment se comportent les variables :
- Une **variable** est dite **locale** lorsqu'elle est créée dans une fonction. Elle n'existera et ne sera visible que lors de l'exécution de ladite fonction.
- Une **variable** est dite **globale** lorsqu'elle est créée dans le programme principal. Elle sera visible partout dans le programme.

```
def carre(x):  
    y = x**2  
    return y  
  
# programme principal  
z = 5  
resultat = carre(z)  
print(resultat)
```


- ❑ Les **modules** sont des **programmes Python** qui contiennent des **fonctions** que l'on est amené à **réutiliser** souvent (on les appelle aussi **bibliothèques** ou **libraries**).
- ❑ Les développeurs de Python ont mis au point de nombreux **modules** qui effectuent une **quantité phénoménale** de **tâches**.
- ❑ Pour cette raison, prenez toujours le réflexe de vérifier si une partie du code que vous souhaitez écrire n'existe pas déjà sous forme de **module**.
- ❑ La plupart de ces **modules** sont déjà **installés** dans les versions standards de Python.
- ❑ Vous pouvez accéder à une documentation exhaustive sur le site de **Python**.

- Pour importer des **modules** en Python, vous pouvez utiliser l'instruction **import**. Voici comment procéder :

import nom_du_module

- Exemple :

```
import math  
print(math.pi)
```

```
3.141592653589793  
8
```

```
import random  
print(random.randint(0, 10))
```

- Il existe un autre moyen d'importer une ou plusieurs **fonctions** d'un module, à l'aide du mot-clé **from**, on peut importer une **fonction spécifique** d'un **module** donné.

```
from random import randint  
print(randint(0, 10))
```

```
from random import *  
print(randint(0, 10))
```

- **Remarque** : Il est inutile de répéter le **nom** du **module** dans ce cas, seul le nom de la **fonction** en question est requis.

□ **Remarque :**

- Dans la pratique, plutôt que de charger toutes les **fonctions** d'un **module** en une seule fois :

```
from random import *
```

- C'est mieux de charger le *module* seul puis d'appeler explicitement les **fonctions** **voulues** de la manière suivante :

```
import random
```

- Si vous souhaitez utiliser un **alias** pour référencer le **module**, vous pouvez le faire comme suit :

```
import nom_du_module as alias
```

- **Exemple:**

```
import math as m  
print(m.pi)
```

```
3.141592653589793
```

- **Remarque :** pour **vider** de la **mémoire** un **module** déjà chargé, on peut utiliser l'instruction **del** :

```
import random  
  
print(random.randint(0, 10))  
del random
```

- ❑ Les **fonctions** sont utiles pour réutiliser une fraction de code plusieurs fois au sein d'un même programme sans avoir à **dupliquer** ce code.
- ❑ On peut imaginer qu'une fonction bien écrite pourrait être judicieusement **réutilisée** dans un autre programme Python. C'est justement l'intérêt de créer un module.
- ❑ En général, les **modules** sont **regroupés** autour d'un **thème** précis.
- ❑ En Python, la **création** d'un **module** est très simple.
 - ▣ Il suffit d'écrire un ensemble de fonctions (et/ou de constantes) dans un fichier,
 - ▣ puis d'enregistrer ce dernier avec une extension **.py** (comme n'importe quel script Python).
- ❑ Pour **appeler** une **fonction** ou une **variable** de ce **module**, il faut que le **fichier .py** soit dans le répertoire courant (dans lequel on travaille) ou bien dans un répertoire listé par la variable d'environnement PYTHONPATH de votre système d'exploitation.

□ Exemple:

```
DATE = 18032024

def bonjour(nom):
    """Dit Bonjour."""
    return "Bonjour " + nom

def ciao(nom):
    """Dit Ciao."""
    return "Ciao " + nom

def hello(nom):
    """Dit Hello."""
    return "Hello " + nom
```

```
import message as msg
print(msg.hello("world"))
print(msg.ciao("Ensah"))
print(msg.bonjour("Monsieur"))
print(msg.DATE)
```

```
Hello World
Ciao Ensah
Bonjour Monsieur
18032024
```

- ❑ Les **modules** sont un des moyens de regrouper plusieurs fonctions.
- ❑ On peut regrouper des **modules** dans ce qu'on appelle des **packages**.
- ❑ Un **package** sert à **regrouper** plusieurs **modules**. Cela permet de ranger plus proprement les **modules**, **classes** et **fonctions** dans des **emplacements séparés**.
- ❑ Si on veut accéder, on doit fournir un chemin vers le **module**.
- ❑ De ce fait, les risques de **conflits** de **noms** sont moins importants et surtout, tout est bien plus ordonné.

- **Exemple:**
- Imaginons que vous installiez un jour une bibliothèque pour écrire une interface graphique. En s'installant, la bibliothèque ne va pas créer ses modules au même endroit. Ce serait un peu désordonné...
- On peut ranger tout cela d'une façon plus claire : d'un côté, on peut avoir les différents objets graphiques de la fenêtre, de l'autre les différents événements (clavier, souris,...), ailleurs encore les effets graphiques...
- Dans ce cas, on va sûrement se retrouver face à un package portant le nom de la bibliothèque.
- Dans ce package se trouveront probablement d'autres packages, un nommé evenements, un autre objets, un autre encore effets.
- Dans chacun de ces packages, on pourra trouver soit d'autres packages, soit des modules et dans chacun de ces modules, des fonctions.

- Exemple de hiérarchie:
- Pour notre bibliothèque imaginaire, la hiérarchie des répertoires et fichiers ressemblerait à cela :
- Un répertoire du nom de la bibliothèque contenant :
- Un répertoire evenements contenant :
 - ▣ un module clavier;
 - ▣ un module souris;
 - ▣ ...
- Un répertoire effets contenant différents effets graphiques ;
- Un répertoire objets contenant les différents objets graphiques de notre fenêtre (boutons, zones de texte, barres de menus...).

- Si on veut créer nos propres **packages**, on doit commencer par créer, dans le **même dossier** que votre **programme Python**, un **répertoire** portant le nom du **package**.
- Dans ce répertoire, vous pouvez soit :
 - ▣ mettre vos modules, vos fichiers à l'extension **.py**;
 - ▣ créer des **sous-packages** de la même façon, en créant un répertoire dans votre **package**.
- **Remarque** : Ne mettez pas d'espaces dans vos noms de packages et évitez aussi les caractères spéciaux. Quand vous les utilisez dans vos programmes, ces noms sont traités comme des noms de variables et ils doivent donc obéir aux mêmes règles de nommage.