

# Python base

---

[m2iformation.fr](http://m2iformation.fr)



# Présentation de Python et ses versions

- Le langage de programmation Python a été créé en 1989 par Guido Van Rossum
- La dernière version de Python est la version 3. Plus précisément, la version 3.14 qui a été publiée en Octobre 2025.
- La version 2 de Python est obsolète et n'est plus maintenue depuis le 1er Janvier 2020
- La [Python Software Foundation](#) est l'association qui organise le développement de Python et anime la communauté de développeurs et d'utilisateur.

# Présentation de Python et ses versions

- Python est :
  - Multiplateforme
  - Un langage de haut niveau
  - Un langage interprété
  - Orienté objet
- Usage de Python :
  - Scripts pour automatiser des tâches
  - Analyses de données, Développement web ect...

# Environnement de développement

- Par défaut, Python est déjà installé sur Linux et MacOS
- Pour Windows, il faudra le télécharger à cette adresse :
  - [Download Python | Python.org](https://www.python.org/downloads/)
  - /!\ Attention, penser à cocher les bonnes cases à l'installation (notamment l'ajout au PATH sur windows)
- Pip est le gestionnaire de paquets de Python et qui est systématiquement présent depuis la version 3.4.

# Environnement de développement

- Un script python est un fichier avec l'extension .py
- Pour démarrer un script python, on utilise la commande py (Windows) ou python/python3 (MacOS et Linux) et le nom du script
- Exemple
  - `python mon_script.py`

# Environnement de développement

- L'interpréteur de Python est l'application qui permet de convertir les instructions python en un langage compréhensible par l'ordinateur.
- Il peut être utilisé pour exécuter une instruction.
- Pour ouvrir l'interpréteur, il suffit de taper dans un terminal (powershell, bash,...) py ou python3.
- Pour quitter l'interpréteur, Ctrl + D ou la fonction exit() de python.

# Environnement de développement (IDE)

- Un IDE (Integrated Development Environment) est un environnement de développement intégré, il offre un ensemble d'outils tel que :
  - Un éditeur de code
  - Un compilateur
  - Un débogueur ect...

# Environnement de développement (IDE)

- Il convient d'installer l'un des IDE suivant :
  - [Pycharm](#) (JetBrains, conçu spécialement pour python, avec une version payante améliorée).
  - [Visual Studio Code](#) (Microsoft, utile dans de nombreux domaines et avec de nombreuses extensions disponibles)
- Lors de l'installation, il peut être utile pour les 2 logiciels de cocher les cases permettant de faire "Ouvrir le projet avec ..." et l'ajout au PATH

# Les normes et conventions en Python

- La syntaxe python est soumise à des conventions.
- Un bon développeur s'assurera qu'il les suit en écrivant ses scripts python.
- Elles permettent de normaliser le langage et de faciliter la lecture.
- Python suit la norme [PEP8](#)
- Les IDE ont souvent un auto-formatage qui respectera une grande partie de ces règles. Les raccourcis pour formater automatiquement changent selon le logiciel.

# Les variables

- Les variables de type **numériques** servant à stocker des nombres.  
Ces variables peuvent être de plusieurs sous-types :
  - les integers **int**, qui servent à stocker des nombres **entiers**
  - les **floats**, servant à stocker des **nombre à virgule flottante** (décimaux)

```
mon_int = 514 # Variable de type integer  
mon_float = 3.14 # Variable de type float
```

# Les variables

- Les chaînes de caractères strings **str** qui permettent de stocker du **texte**.
- Les **booléens**, permettant de stocker des valeurs binaires (**Vraie=True** ou **Fausse=False**)
- Le vide, **None** en python, est un type à part qui ne représente 'Rien', il n'est **ni un 0, ni un False**

```
mon_bool = True # Variable de type booléens  
ma_string = "Je suis une chaîne de caractère" # Variable de type string.
```

# Les variables

- L'opérateur **=** en python est l'opérateur **d'assignation/affectation**, il permet d'affecter à une variable une valeur donnée.
  - Ex: **ma\_variable = 3** permet d'assigner la valeur entière 3 à la variable 'ma\_variable'.
- Autrement, lorsque l'on mettra **ma\_variable** dans une instruction Python, on récupérera la valeur contenue dans cette variable
  - Ex: **ma\_variable2 = ma\_variable + 1** permet d'assigner la valeur entière 4 à la variable 'ma\_variable2'.

# Le mot clé **del**

- Le mot clé **del** permet de **supprimer de la donnée** en python
- Il peut être utile lorsque l'on cherche à **libérer de la mémoire** ou à se débarrasser de certaines variables/fonctions/classes/...

# Un peu de lexique

- Une **fonction** est un morceau de code déjà écrit, il prend des **paramètres** et retourne une **valeur**.
- On pourra **exécuter** ce morceau de code en ajoutant des parenthèses avec les **valeurs passées en paramètres** après le nom de la fonction.

```
# Exemple de fonction  
print("Un message à afficher")
```

# Un peu de lexique

- Une **méthode** est similaire à une fonction, la différence est qu'elle **s'applique** à une valeur/un objet donné, on ajoutera un ":" après celui-ci suivi du **nom de la méthode** et des **parenthèses**.
- Lorsque l'on utilise ces 2 concepts, les **valeurs** que l'on met **entre les parenthèses** seront appelées **paramètres** ou **arguments**.

```
# Exemple de méthode  
test_maj = "test".upper()
```

# Un peu de lexique

- Les mots **console** et **terminal** reviennent à peu près à la même chose, il s'agit d'une **fenêtre** permettant la **communication** entre l'utilisateur et le programme par du **texte**.
- Un **script python** est un fichier contenant des instructions.
- Un **programme** ou une **application** correspond au processus, résultat de l'interprétation du script par l'ordinateur.
- un **module python** est le contenu de l'interprétation d'un script python à l'exécution, nous y reviendrons plus tard...

# Récupérer et afficher des valeurs

- Pour qu'il y ait un **dialogue** possible entre l'utilisateur et l'ordinateur (application console), on a recours à ce qui s'appelle des affichages et des **récupérations de valeurs**.
- **L'affichage** se fait par la fonction **print()**, qui affichera les valeurs passées en paramètre sur le terminal.
- **La récupération** se fait par la fonction **input()**, elle récupère une saisie de l'utilisateur sous forme de **str**. il est possible d'y ajouter une chaîne à afficher en paramètre.

# Récupérer et afficher des valeurs

```
ma_recuperation = input("Veuillez entrer une valeur : ")
print("Vous avez entrer comme valeur : ", ma_recuperation)
```

- Ces deux processus amènent rapidement à **deux composantes essentielles** de la programmation console :
  - **L'affichage** : le **formatage** des str.
  - **la récupération** : le **casting** des variables.

# Récupérer et afficher des valeurs

- Si l'on veut **présenter** de façon claire des **informations** à l'utilisateur, il faut souvent se servir du **formatage**. Il existe plusieurs méthodes.
- Le **%-formating**, maintenant déprécié et utilisé de nos jours dans de rares cas (formatage de dates, requêtes SQL)
- les **f-strings** sont des chaînes de caractères pour **formater directement le texte** en y incluant entre accolades les variables :

```
print(f"La valeur de nombre_a vaut {nombre_A:0.2f}") # nombre_a = 3.14
```

# Fonctionnement des f-strings

- Lorsque l'on utilise un f-string, on retrouve souvent ce genre de syntaxe :

```
variable = 55.2091  
f"{variable:^7.2f}" # 55.21
```

- Ici la valeur sera :
  - 7: Dans un espace de 7 caractères minimum au total (virgule, décimales, ...) on ajoutera le nombre d'espaces nécessaire si pas assez de caractères.
  - 2: avec toujours 2 chiffres après la virgule (arrondi si besoin)

# Les raw-strings

- Similaire au f-strings, il existe aussi en python les **raw-strings**.
- Ce sont des chaînes dans lesquelles les **caractères spéciaux** comme le **backslash** ne sont pas interprétés.
- Ils facilitent la saisie des chemins de fichier ou de regex par exemple.

```
print("\n{1}")
print(f"\n{1}")
print(r"\n{1}")
```

# Cast des variables

- Lorsque l'on utilise un langage de programmation, on a fréquemment besoin de **passer d'un type de variable vers un autre**. Pour ce faire, on sert de ce qui s'appelle le "**cast**" (En français **transtypage**). Pour réaliser un **transtypage**, il faut utiliser **la fonction de cast** qui porte **le nom du type vers lequel on veut passer**.

# Cast des variables

```
ma_string = "599.98"
mon_prix = float(ma_string)
print(f"Ma string vaut {ma_string} et est un type {type(ma_string)}")
# Ma string vaut 599.98 et est de type <class 'str'>

print(f"Mon prix vaut {mon_prix} et est un type {type(mon_prix)}")
# Mon prix vaut 599.98 et est de type <class 'float'>
```

# Cast des variables

- De plus, lorsque l'on cherche à obtenir un nombre de l'utilisateur, on peut également directement caster l'input de la sorte :

```
mon_nombre = int(input("Veuillez entrer un nombre : ")) # 25

print(f"Mon nombre vaut {mon_nombre} et est un type {type(mon_nombre)}")
# Mon prix vaut25 et est de type <class 'int'>
```

- Attention ! Le casting peut être la source de nombreux problèmes générant ce que l'on appelle des **exceptions** ! Nous verrons comment traiter les exceptions plus tard.

# Cast en booléens

- Lorsque l'on fait un **cast** en **bool**, python applique certaines règles:
  - Les valeurs **None**, **False**, **0**, **0.0** et **""** donnent forcément **False**
  - **Toutes les autres valeurs donnent True**
- En réalité toute valeur correspondant au vide pour son type sera considérée comme False dans une condition (cf : **partie block conditionnels**)

## Exercice :

- Écrire un programme, qui, à partir de la saisie d'un nom et prénom, affiche le message suivant :
  - Bonjour M. Ou Mme {Prénom} {Nom}
- Il peut être utile de chercher en ligne les méthodes lower, upper, capitalize et/ou title pour forcer la casse.

# Les opérateurs arithmétiques

- Il existe plusieurs types **d'opérateurs**: unaires, binaires, arithmétiques, logiques, d'affectation, relationnels, ect... Il est possible d'utiliser les **opérateurs arithmétiques** sur les variables pour les manipuler. Il est possible de les **combiner** avec l'**opérateur d'assignation** pour effectuer l'opération directement sur une variable donnée.

```
mon_resultat = 4 + 4 # on affecte à la variable mon_resultat 4 + 4 soit 8
mon_resultat = mon_resultat - 4 # on affecte à la variable mon_resultat sa valeur - 4 soit 4
mon_resultat *= 2 # On multiplie la valeur de la variable par 2 (opérateur unaire)
mon_resultat /= # On divise (division entière)la valeur de la variable par 2
mon_resultat /= # On divise (division décimale)la valeur de la variable par 2
mon_resultat **= 3 # On passe la variable à la puissance 3
mon_resultat %= 3 # On consert le reste de la division de la variable par 3
```

# Les opérateurs arithmétiques

- Il faut également savoir que l'opérateur **d'addition** + permet à deux variables de type **string** de s'ajouter l'une à la suite de l'autre, ce qui s'appelle la **concaténation**. Celui de **multiplication** \* entre un **string** et un **int** permettra de faire ce qu'on appelle la **réPLICATION**.

## Exercice :

Écrire un programme qui, à partir de la saisie d'un rayon et d'une hauteur, calcule le volume d'un cône droit

# Les opérateurs relationnels

- Il est également possible de comparer des valeurs via les opérateurs relationnels qui sont :
  - ">" : Supérieur à
  - "<" : Inférieur à
  - ">=" : Supérieur ou égal à
  - "<=" : Inférieur ou égal à
  - "==" : Égal à
  - "!=" : Différent de

# Les opérateurs logiques

- Les valeurs de type booléennes peuvent être manipulées via les opérateurs logiques. En python, les 3 principaux sont **NOT**, **AND** et **OR**. On peut donner leurs résultats possibles sous forme de tables de vérités. Il existe notamment d'autres opérateurs.
- Ces opérateurs vont aussi donner comme résultat une variable de type **booléen**.

# Les opérateurs logiques

```
variable_bool_A = True  
variable_bool_B = False  
  
variable_bool_C = variable_bool_A and variable_bool_B # False  
  
variable_bool_D = variable_bool_A or variable_bool_B # True
```

Table de vérité de ET

a	b	a ET b
0	0	0
0	1	0
1	0	0
1	1	1

Table de vérité de OU

a	b	a OU b
0	0	0
0	1	1
1	0	1
1	1	1

## Exercice :

Écrire un programme qui, à partir de la saisie de l'âge de l'utilisateur, affiche True s'il est majeur et False s'il est mineur (sans structure conditionnelle if).

# Les structures conditionnelles

- Si l'on veut permettre à nos programme de prendre **des chemins différents** de manière **conditionnelle**(saisies, calculs, ...) on a recours aux **structures conditionnelles**
- Pour réaliser une structure conditionnelle, on se sert des **clauses/mots clés** suivants :
  - **if** : initialiser la structure de contrôle, on donne une **condition** et on **exécute** le bloc d'instruction si elle est **vraie**.
  - **elif** : ajouter une **autre condition** dans le cas où la précédente **n'aura pas été validée**. On peut enchaîner ainsi **autant de structures elif que l'on souhaite**.
  - **else** : toujours la **dernière partie** d'une structure de contrôle, son bloc est exécuté dans le cas où **aucune des conditions précédentes n'a été validée**.

# Les structures conditionnelles

```
mon_age = int(input("Veuillez donner votre âge : "))

if mon_age >= 21 :
    print("Vous êtes majeur aux USA")
elif mon_age >= 18 :
    print("Vous êtes majeur en France")
else:
    print("Vous êtes mineur")
```

- Les clauses **elif** et **else** sont facultatives dans la structure conditionnelle, on en mets que si on en a réellement besoin.

# Exercice

- Écrire un programme qui prend en entrée une température **temp** et qui renvoie l'état de l'eau à cette température c'est-à-dire "SOLIDE", "LIQUIDE" ou "Gazeux".
- On prendra comme conditions les suivantes :
  - Si la température est strictement **négatives** alors l'eau est à l'état **solide**.
  - Si la température est **entre 0 et 100** (compris) l'eau est à l'état **liquide**.
  - Si la température est strictement **supérieur à 100** l'eau est à l'état **gazeux**.
- Il est possible de réaliser cet exercice sans if imbriqué grâce au elif

# Exercice

- Écrire un programme qui permet de tester si un profil est valable pour une candidature ou non selon ces trois critères :
  - **L'âge minimum** pour le poste est **30** ans
  - Le **salaire maximum** possible est **40 000** euros.
  - Le **nombre** d'années d'expérience **minimum** est de **5** ans.
- On affichera différents messages pour chaque condition non respectée.
- Il est possible de réaliser cet exercice avec une seule structure conditionnelle ne comportant qu'une condition par clause (pas de and/or)

# Instruction pass

- L'instruction **pass** est une instruction à part de python.
- Elle ne fait **absolument rien !**.
- Il est régulier que l'on s'en serve de manière provisoire dans un bloc (if, else, for, fonction...) où l'on ne compte pas mettre d'instructions pour le moment.

# match case

- Lorsque l'on travaille avec **la structure conditionnelle**, il est fréquent que l'on ait beaucoup de **elif** qui utilisent **la même variable**.
- Depuis la version **3.10** de python, il existe une nouvelle structure, le **pattern matching** ou structure **match...case**.

```
# Sans match case
if var == 1:
    print("une")
elif var == 2:
    print("deux")
else:
    print("ni une, ni deux")

# Avec match case
match var:
    case 1:
        print("une")
    case 2:
        print("deux")
    case _:
        print("ni une, ni deux")
```

# Les ternaires

Avancé

- En python, il est possible d'utiliser ce qu'on appelle les **ternaire**, il s'agit d'une **expression** comportant une **condition**. On peut le comparer à une **structure conditionnelle if**.
- Il se structure comme suit :

```
temp = int(input("Saisir la température de l'eau : "))
etat = "solide" if temp < 0 else ("liquide" if temp <= 100 else "gazeux")
```

# Les structures itératives

- Il existe en Python deux façons de faire des **boucles/structures d'itération**. Les instructions du bloc seront exécutées à chaque **itération** de celle-ci.
  - La boucle "**while**" (Tant que ...) qui sera exécutée **tant que la condition spécifiée est vraie**
  - La boucle "**for...in...**" (Pour chaque... dans...) qui sera exécutée **pour chaque élément** d'un ensemble de type **conteneur ou interval** (fonctionne aussi avec les str). Elle met chaque élément un à un dans une **variable**.

```
for _ in range(0, 10):
    print("Je me répète !")

for element in [0, 1, 2, 3, 4, 5]:
    print(element)

for item in range(1, 11)
    print("Je suis l'itération n° : ", item)
```

- Lorsque l'on ne se sert jamais de la variable en question, le norme est de la nommer par "\_"

# Les structures itératives

- Lorsque l'on utilise une structure itérative, on peut également utiliser des **mots clés** durant l'itération, tels que :
  - "continue" : On passe à **l'itération suivante** en se replaçant au **début de la boucle**. On **ignore** alors tout ce qui aurait dû se dérouler **après le mot-clé**.
  - "break" : On **sort immédiatement de la boucle** sans effectuer les instructions après le mot-clé et dans les itérations suivantes.

```
while True:  
    valeur = input("Saisir STOP pour arrêter le programme :")  
    if valeur == "STOP":  
        break  
    elif valeur.upper() == "Stop":  
        print("EN UPPERCASE")  
        continue  
    else:  
        pass # Ce bloc est inutile, pass ne fait rien.
```

# Exercice

- Écrire un programme en python qui affiche les tables de 1 à N. N : est un entier supérieur à 0 saisi par l'utilisateur.
- Gérer l'affichage en ajoutant des espaces et en retournant à la ligne après chaque table.
- /!\ Une table va de 1 à 10.

Table de multiplication										
1	2	3	4	5	6	7	8	9	10	
1	2	3	4	5	6	7	8	9	10	
2	4	6	8	10	12	14	16	18	20	
3	6	9	12	15	18	21	24	27	30	
4	8	12	16	20	24	28	32	36	40	
5	10	15	20	25	30	35	40	45	50	
6	12	18	24	30	36	42	48	54	60	
7	14	21	28	35	42	49	56	63	70	
8	16	24	32	40	48	56	64	72	80	
9	18	27	36	45	54	63	72	81	90	
10	20	30	40	50	60	70	80	90	100	

# Exercice

- Écrire un programme qui permet d'afficher un triangle isocèle formé d'étoiles \*.
- La hauteur du triangle (le nombre de lignes) sera saisie, comme dans l'exemple ci-contre.
- Il existe plusieurs méthodes pour arriver au résultat.
- Quelques pistes : f-strings, mathématiques, for imbriqués, incrémentation et décrémentation.

saisir la hauteur du triangle : 10



# Exercice

- On dispose d'une feuille de papier d'épaisseur 0.1 mm.
- Combien de fois doit-on la plier au minimum pour que l'épaisseur dépasse 400m ?
- Écrire un programme en Python pour résoudre ce problème.
- Une fois fini, aborder le problème à l'inverse.
- Combien de fois doit-on déplier une feuille de 400m au minimum pour que l'épaisseur dépasse 0.1mm.

# Exercice

- Réaliser un programme permettant à l'utilisateur d'entrer comme données :
  - Une population initiale.
  - Un taux d'accroissement
  - Une population visée.
- Ce programme permettra à l'utilisateur de savoir en combien de temps la population visée sera atteinte.

# Les Fonctions

- En programmation, les **fonctions** sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme.
- Une fonction effectue une tâche. Pour cela, elle **reçoit** éventuellement des **arguments** et **renvoie** une **valeur** ou **None** (Rien).
- Chaque fonction effectue en général une tâche **unique et précise**. Si cela se complique, il est plus judicieux d'écrire **plusieurs fonctions** (qui peuvent éventuellement s'appeler les unes les autres).
- Les valeurs passées à l'exécution de la fonction s'appellent des **arguments**.
- Les variables entre parenthèses qui **contiendront** ces valeurs sont les **paramètres**.

# Les Fonctions

- 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**.
- Le nombre **d'arguments** que l'on peut passer à une fonction est **variable** et dépend du **nombre de paramètres**.
- Il est possible de passer un ou plusieurs argument(s) de manière **facultatives** et de leur attribuer une valeur par **défaut**. Pour cela, on utilise le **=**.

```
def return_number(nombre: int):  
    return nombre  
  
res = return_number(2) # retourne 2  
  
def carre(nombre=3):  
    return nombre ** 2  
  
res = carre() # retourne 9
```

# Les Fonctions

- 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 la dite fonction.
- Une variable dite **globale** lorsqu'elle est créée **dans le programme principal**. Elle sera visible partout dans le programme.
- Lorsque l'on essaie de modifier une variable globale à l'intérieur d'une fonction, il sera obligatoire d'utiliser le mot clé **global**

```
a = 10
def fonction():
    global a
    a += 1
```

# Exercice

- Écrire une fonction qui prend en paramètre :
  - prénom
  - nom
- Elle retournera une chaîne avec le prénom et le nom séparé d'un espace, exemple "John Doe"
- Vous afficherez le résultat de cette fonction à l'aide de la fonction `print()`

# Exercice

- Écrire la fonction "soustraire" qui prend en paramètre :
  - nombre a
  - nombre b
- Elle retournera un entier qui sera la soustraction de ces deux nombres
- exemple : `soustraire(2, 1) # résultat = 1`
- De plus, lors de l'exécution, la fonction affichera "Je soustrais 2 et 1"
- Vous afficherez le résultat à l'aide de la fonction `print()`

# Exercice

- Écrire une fonction quelle\_heure.
- Cette fonction aura un paramètre heure de type str.
- Ce paramètre aura "12h00" comme valeur par défaut.
- La fonction ne retournera aucun résultat mais affichera l'heure avec la fonction print()
- exemple : quelle\_heure() # résultat : "il est 12h00"
- exemple : quelle\_heure("14h00") # résultat : "il est 14h00"

## Exercice

- Écrire une fonction `compter_lettre_a`.
- Cette fonction prendra en paramètre une chaîne
- Créer une boucle qui parcourt les lettres de la chaîne et compte le nombre de lettres égales à "a".
- La fonction renverra un entier.
- exemple : `compter_lettre_a("abba")` # résultat : 2
- exemple : `compter_lettre_a("mixer")` # résultat : 0
- Écrire une autre fonction sans boucle qui utilisera **count** à la place.

# Exercice

- Écrire un programme qui permet de saisir une chaîne d'ADN ainsi qu'une séquence d'ADN et qui retourne le d'occurrences de la séquence dans la chaîne
- Cette séquence sera composée uniquement de la combinaison de lettre suivante 'a', 't', 'c', 'g'.
  1. Écrire une fonction vérification\_adn qui permet de renvoyer la valeur True si la chaîne d'ADN est valide, False si elle est invalide
  2. Écrire une fonction saisie\_adn qui récupère une saisie, vérifie sa validité et renvoie une chaîne d'ADN valide sous forme de chaîne
  3. Écrire une fonction proportion qui reçoit deux paramètres une chaîne d'ADN et une séquence d'ADN Elle renverra le d'occurrences de la séquence dans la chaîne
  4. Créer des instructions pour pouvoir tester le programme

# Modules

- Un **module** est un ensemble d'instructions provenant d'un **script** et qui peut être (ré)utilisé par d'**autres scripts**
- Intérêts : faciliter la réutilisation, la lisibilité, le débogage, le travail d'équipe
- Python vient avec un ensemble de modules natifs : scrypt, csv, datetime, math, ...
- Pour avoir la liste complète des modules fonction **help('modules')**.
- Python nous donne la possibilité de créer nos propres modules.

# Modules

- Un **module** contient donc **l'ensemble des variables et des fonctions**, définies par les instructions du **script**. Il est entièrement exécuté au moment de l'instruction **import**

```
from math import pi

def circonference_cercle(rayon):
    return 2 * pi * rayon
```

- Ici le module nommé **circle** contiendra 2 éléments :
  - La variable **pi**
  - La fonction **circonference\_cercle**
- On peut utiliser la fonction **dir(module)** pour en connaître le contenu.

# Modules

- **L'importation** permet à un script d'utiliser le code d'un **autre module**.
- Syntaxes de l'importation :
  - **import nom\_module**
  - **from nom\_module import fonction1** ect...
  - **from nom\_module import \***
- Syntaxe d'accès à un membre d'un module importé
  - **module.fonction** avec la méthode 1
  - **nom\_membre** avec la méthode 2 et 3
  - Il est possible d'ajouter un alias à un module ou un membre
    - **import nom\_module as mod\_1**
    - **from nom\_module import fn1 as f1**

```
import circle
result = circle.circonference(10)

from circle import circonference
result = circonference(10)
```

# La variable `__name__` (variable Dunder)

- On retrouve souvent cette structure pour les scripts python, elle comporte beaucoup d'avantages quand on travaille avec des **imports**
- `__name__` est une **variable** prédéfinie dans chaque module, elle contient :
  - La chaîne "`__main__`" si on est dans le module principal, lancé directement depuis le script
  - Le **nom du module** quand on est dans un module importé **import**
- De ce fait, le bloc `if __name__ == "__main__"` n'est **exécuté** que dans le cas où l'on est dans le **module principal**. Les modules importés ne l'exécuteront pas.
- On peut voir ça comme du code verrouillé qui ne s'exécute que si on lance directement le script.

```
import math

def addition(a,b):
    return a + b

def main():
    print(addition(40, 3))

if __name__ == "__main__":
    main()
```

# Exercice

- Restructurer le script ADN avec la structure vue précédemment
- Écrire un script qui utilise les fonctions de l'exercice ADN.

# Packages

- un **package** est comme un dossier contenant des **sous-packages** et/ou des **modules**.
- Nous pouvons également en utiliser, un package du **Python Package Index (PyPI)**, installable facilement avec **pip**.
- pour importer un package, nous utilisons l'**import (package.module)**.
- Un package doit avoir le fichier **init.py**, même si vous le laisser vide.
- Mais lorsque nous importons un package, seuls ses modules immédiats sont importés, pas les sous-package. Si vous essayez d'y accéder, cela déclenchera un `AttributeError`.

# Manipuler les fichiers

- On peut manipuler les fichiers via la fonction `open()`
- Elle prend en premier paramètre un chemin de fichier (**path**) et en second paramètre un **mode**, composé de 2 parties:
  - Le mode d'ouverture :
    - r : Lecture
    - w : Écriture
    - a : Ajout
  - Le type d'ouverture:
    - t : Ouverture sous format texte (par défaut)
    - b : Ouverture en mode binaire.
- Il est important de penser à fermer notre fichier en fin d'utilisation sous peine d'avoir des problèmes d'accès. Pour ce faire, on utilise `.close()` sur notre variable résultat de la fonction `open()`.

# Écrire et lire dans un fichier

- Une fois notre fichier ouvert, on peut le lire ou le modifier. Pour ce faire, il existe en Python plusieurs méthodes.
  - `read()` : Pour lire **l'ensemble du fichier** tel qu'il est écrit.
  - `readline()` : Pour lire **ligne par ligne** le fichier (le curseur de fichier passera à la ligne suivante après la méthode). Cette méthode prendra en compte les caractères spéciaux tels que le caractère de retour à la ligne `\n`.
  - `readlines()` : Pour obtenir une **liste contenant les lignes du fichier**. Cette méthode prendra en compte les caractères spéciaux tels que le caractère de retour à la ligne `\n`.
  - `write()` : Permet **d'écrire du texte**.
  - `writelines()` : Permet **d'écrire une liste de lignes**.

# Exercice

- Écrire un programme permettant à un utilisateur de sauvegarder un texte secret dans un fichier.
- Si le fichier n'existe pas, il devra être créé avec un nouveau secret.
- L'utilisateur pourra :
  - Voir le secret
  - Modifier le secret
  - Quitter le programme (Cette action sauvegardera le fichier)
- Pour éviter tout problème, il est conseillé de ne lire et écrire le fichier qu'une seule fois à l'entrée et la sortie du programme.

# Qu'est-ce qu'un conteneur ?

- Un **conteneur** est un **objet** permettant de stocker d'autres **objets**.
- Pour une **liste**, il est possible de stocker plusieurs **valeurs** au sein de la même variable.
- Les **conteneurs** sont **dynamiques**, ils peuvent contenir **plusieurs types de données** (int, str, float, list, object...)
- Pour plusieurs conteneurs, on peut **accéder aux valeurs** par utilisation d'un **index** ou d'une **clé**.

```
ma_liste = [1, 2, 3, 4, 5]
print(ma_liste) # [1, 2, 3, 4, 5]

mon_dict = {'Key1' : 123, 'Key2' : '456', 'Key3' : [7, 8, 9] }
print(mon_dict) # {'Key1' : 123, 'Key2' : '456', 'Key3' : [7, 8, 9] }

ma_tuple = (1, 'blabla', 3.14)
print(ma_tuple) (1, 'blabla', 3.14)

ma_liste = [1, 2, 3, 4, 5]
print(ma_liste[2]) # 3

mon_dict = {'Key1' : 123, 'Key2' : '456', 'Key3' : [7, 8, 9] }
print(mon_dict["key2"]) # 456
```

# Les listes

- La **liste** est le type de **conteneur** le plus utilisé.
- Elle permet de **manipuler** facilement ses données via l'utilisation de ses **méthodes**.
- Les **méthodes** des listes les plus utilisées sont les suivantes :
  - **sort()** : trie les éléments de la liste.
  - **append(element)** : ajouter un élément à la fin de la liste.
  - **extend(list)** : ajouter une liste à la fin de la liste.
  - **pop(index)** : retirer un élément de la liste à l'index donné.
  - **remove(element)** : retirer le premier élément de la liste qui correspond.
  - **count(element)** : nombre d'occurrences d'un élément.

```
ma_liste = []
print(ma_liste) # []

ma_liste = [1, 2, 3]
print(ma_liste) # [1, 2, 3]

ma_liste = [2, 1, 3]
print(ma_liste) # [2, 1, 3]

ma_liste.sort()
print(ma_liste) # [1, 2, 3]

ma_liste.append(4)
print(ma_liste) # [1, 2, 3, 4]

ma_liste.extend([5, 6])
print(ma_liste) # [1, 2, 3, 4, 5, 6]

ma_liste.remove(4)
print(ma_liste) # [1, 2, 3, 5, 6]

ma_liste.pop(2)
print(ma_liste) # [1, 2, 5, 6]
```

# l'itération sur une liste

- l'**itération** est la capacité de **parcourir** (via généralement une boucle) une **série de valeurs** contenues dans un conteneur afin de les afficher ou d'en modifier les valeurs de façon séquentielle.
- Pour parcourir une liste, on utilise généralement une boucle **for**, telle que :

```
ma_liste = [1, 2, 3, 4, 5]
print(ma_liste) # [1, 2, 3, 4, 5]

for item in ma_liste:
    print(item)

    ...
1
2
3
4
5
    ...
```

# Exercice

- Via l'utilisation d'une variable de type **liste**, vous devrez réaliser un logiciel permettant à l'utilisateur d'entrer une **série de notes**, dont le nombre possible à entrer sera soit (au choix de l'utilisateur) :
  - saisie avant la saisie de notes
  - permissif et pourra aller jusqu'à entrer une note négative qui stoppera la saisie des notes.
- Une fois la **saisie des notes terminée**, l'utilisateur aura à sa disposition un **affichage** lui permettant d'avoir la **note max**, la **note min** ainsi que la **moyenne** (possible de faire un menu pour choisir)

```
Veuillez entrer une note entre 0 et 20 compris (une note négative stoppera la saisie) : 12
Veuillez entrer une note entre 0 et 20 compris (une note négative stoppera la saisie) : 11
Veuillez entrer une note entre 0 et 20 compris (une note négative stoppera la saisie) : 9
Veuillez entrer une note entre 0 et 20 compris (une note négative stoppera la saisie) : 8
Veuillez entrer une note entre 0 et 20 compris (une note négative stoppera la saisie) : 7
Veuillez entrer une note entre 0 et 20 compris (une note négative stoppera la saisie) : -5
La note maximale est de 12.00 / 20
La note minimale est de 7.00 / 20
La moyenne est de 9.40 / 20
```

# Modules natifs

- Comme vu avant Python vient avec un ensemble de module de base
- Quelques exemples d'utilisation
- module csv :

```
import csv
fichier = open("noms.csv", "rt") # la fonction open pour ouvrir un fichier et r pour lecture seul t pour text brut
lecteurCSV = csv.reader(fichier, delimiter=";") # Ouverture du lecteur CSV en lui fournissant le caractère séparateur
for ligne in lecteurCSV:
    print(ligne) # Exemple avec la 1e ligne du fichier d'exemple : ['Nom', 'Prénom', 'Age']
fichier.close()

# Ecriture
fichier = open("annuaire.csv", "wt", newline=";") # On peut changer le newline
ecrivainCSV = csv.writer(fichier, delimiter="|") # On peut changer le délimiteur
ecrivainCSV.writerow(["Nom","Prénom","Téléphone"]) # On écrit la ligne d'en-tête avec le titre des colonnes
ecrivainCSV.writerow(["Martin","Julie;Clara","0399731590"]) # attention au caractères spéciaux (,,:« '.)
fichier.close()
```

# Exercice

- Écrire un script qui demande les informations d'un produit :
  - Titre
  - Prix
  - Stock
- Il les ajoute ensuite dans un fichier produits.csv

# Les lambdas

Avancé

- Les lambdas sont des fonctions simplifiées à l'extrême et anonymes.
- Elles n'ont pas de nom et s'utilisent en général comme arguments d'autres fonctions (cf diapo filter, map, reduce).
- Elles doivent rester très simples (généralement 1 instruction).

```
fct = lambda x : x**2  
  
def fct2(x):  
    return x**2  
  
print(fct(2))  
print(fct2(2))
```

# Sorted, Filtered, Map et Reduce

Avancé

- Pour aller plus loin dans l'usage des listes, il existe certaines fonctions utiles.
- Ces 4 fonctions utilisent les **fonctions ou lambdas** et nous simplifient beaucoup le travail avec les listes.
  - **sorted** : trier la liste selon certains critères.
  - **filter** : filtrer les éléments de la liste.
  - **map** : créer une nouvelle liste avec tous les éléments transformés par une fonction.
  - **reduce** : réduire la liste à une seule valeur.

# Les tuples

- Le tuple permet de **regrouper** des données, on appelle ça du **packing/construction**.
- Les données sont **non-modifiables** et identifiées par leurs **indices/index**.
- Syntaxes de définition :
  - **mon\_tuple = ()** ou **mon\_tuples = tuple()** ou **mon\_tuple=(1,2,3)** ou **mon\_tuple=1,2,3**.
  - Avec les mêmes méthodes que pour les listes, on pourra itérer sur les tuples et récupérer les valeurs à des index précis.
  - Les opérations sur un tuple :
    - **len(tuple)** => nombre d'éléments d'un tuple.
    - **tuple.count(element)** => nombre d'occurrences d'un élément dans le tuple.
    - **tuple.index(element)** => index de la première occurrence de l'élément.
  - Avec une assignation à plusieurs variables, python propose **l'unpacking**.
  - exemple : `var1, var2 = (1, 2)`

# Exercice

- Écrire un programme se servant d'une fonction retournant, à partir de deux nombres lui étant envoyés en paramètres :
  - La somme
  - La différence
  - Le quotient
  - Le produit
- Vous testerez cette fonction dans le cadre d'un programme console demandant à l'utilisateur deux valeurs et lui permettant d'obtenir les 4 résultats en même temps.

# les sets

- Un **set** est un ensemble d'éléments **uniques** et **ordonnés**, les doublons sont **impossibles**, les valeurs doivent donc être **immutable**.
- Lors de **l'ajout** ou du **retrait** d'un élément d'un set, le conteneur se voit ainsi automatiquement **réordonné**. L'ordre définit par python n'est pas toujours très sensé...
- Lorsque l'on **cast** une list en set, on obtient une série d'éléments **sans doublons** qui ne peuvent plus être modifiés via leur **index** (les sets ne permettant pas la modification des éléments via cette méthode).
- Les sets contiennent des méthodes semblables aux listes mais n'en disposent pas de beaucoup.

```
mon_set = {1, 2, 3, 5, 5, 6}
print(mon_set) # {1, 2, 3, 5, 6}
mon_set.add(4)
print(mon_set) # {1, 2, 3, 4, 5, 6}
mon_set.pop()
print(mon_set) # {2, 3, 4, 5, 6}
# mon_set[2] = 5 n'est pas possible pour un set
```

```
ma_list = [1, 1, 24, 3, 10, 3, 4, 5, 5, 54, 5, 6]
print(ma_list) # [1, 1, 24, 3, 10, 3, 4, 5, 5, 54, 5, 6]
mon_set_2 = set(ma_list)
print(mon_set_2) # {1, 3, 4, 5, 6, 10, 54, 24}
```

# Méthodes des sets

- **add(element)** : ajout d'élément.
- **update(set)** : fusion de 2 sets.
- **remove(element)** : supprime l'élément s'il est présent, sinon erreur.
- **discard(element)** : supprime l'élément s'il est présent, sinon ne fait rien.
- **isdisjoint(set)** : si aucun élément n'est commun entre les deux.
- **issubset(set2)** : si le set est compris dans le set2 .
- **issuperset(set2)** : si les set2 est compris dans le set.
- On retrouve aussi les méthodes d'**union |**, **d'insertion &**, de **différence -** et de **différence symétrique ^**.

# exercice

- Via l'utilisation d'une variable de type set contenant des noms de familles, vous devrez réaliser une application permettant à l'utilisateur :
  - de les stocker
  - de les afficher
  - de les éditer
  - de les supprimer
- Pour ce faire, l'utilisateur aura à sa disposition un menu permettant de naviguer entre les différentes fonctionnalités du programme, comme dans l'exemple ci-dessous.

```
== MENU PRINCIPAL ==
1. Voir les noms de famille
2. Ajouter un nom de famille
3. Editer un nom de famille
4. Supprimer un nom de famille
0. Quitter le programme
Votre choix : 1

== LISTE NOMS DE FAMILLE ==
LUCIEN
```

# Les dictionnaires

- un **dictionnaire** est un conteneur se servant d'une association de **clés** et de **valeur**.
- Il est possible d'accéder aux valeurs qui le constituent via l'utilisation de la clé associée entre crochets. Avec le mot clé **del**, on peut supprimer une entrée.
- Certaines méthodes du dictionnaire produisent des types spéciaux qu'il faudra **cast en list**.
  - **.values()** : récupère les valeurs.
  - **.keys()** : récupère les clés.
  - **.items()** : récupère des tuples (clés, valeurs).
- Via **L'unpacking** des tuples et la méthode **.items()**, il est possible d'afficher les informations du dictionnaire plus facilement.

```
mon_dict = {"k1": "valeur un", "k2": 25443, "k3": 3.14, "k4": {1: 'blabla'}}
print(mon_dict)
print(mon_dict['k3']) # 3.14
print(mon_dict['k4'][1]) # blabla
print(mon_dict.values()) # ['valeur un', 25443, 3.14, {1: "blabla"}]
print(mon_dict.keys()) # ["k1", "k2", "k3", "k4"]
print(mon_dict.items()) # {"k1": "valeur un", "k2": 25443, "k3": 3.14, "k4": {1: 'blabla'}}

for key, value in mon_dict.items():
    print(f"key : {key}, value : {value}") # key : k1, value : valeur un ect...
```

# Les dictionnaires

- Pour parcourir un dictionnaire, on utilise généralement une boucle for, on peut également accéder en complément de la valeur. Les clés des dictionnaires sont forcément de types immutables

```
mon_dict = {'key1': 123, 'key2': '456', 'key3': [7, 8, 9]}\nprint(mon_dict) # {'key1': 123, 'key2': '456', 'key3': [7, 8, 9]}\nfor key, value in mon_dict.items():\n    print(f"{key}: {value}")
```



```
{'key1': 123, 'key2': '456', 'key3': [7, 8, 9]}\nkey1: 123\nkey2: 456\nkey3: [7, 8, 9]\n|
```

# exercice

- Avec des variables de type dictionnaire dans une liste, vous réaliserez un logiciel pour stocker une série d'adresses avec :
  - numéro de voie
  - complément
  - intitulé de voie
  - commune
  - code postal
- Pour ce faire, vous utiliserez des clés de type string qui représenteront les différentes lignes de l'adresse dans le dictionnaire.
- Le logiciel devra permettre l'ajout, l'édition, la suppression et la visualisation des données par l'utilisateur.

```
==> MENU PRINCIPAL ==>
1. Voir les adresses
2. Ajouter une adresse
3. Editer une adresse
4. Supprimer une adresse
0. Quitter le programme
Votre choix : 2

==> AJOUTER UNE ADRESSE ==>
Veuillez entrer le numéro de voie SVP : 96
Veuillez entrer le complément d'adresse SVP : Apt 47
Veuillez entrer l'intitulé de voie SVP : Rue des Fleurs
```

# list/set/dict comprehension

Avancé

- Il est possible de **générer et d'itérer** sur des conteneurs à l'aide de la **comprehension**
- Pour la **list comprehension**, la syntaxe est la suivante, un **itérable** est un objet sur lequel on peut itérer:
  - var = [expression for element in iterable]
- Il est aussi possible d'ajouter un filtre avec un if après l'itérable (équivalent à la fonction filter)

```
liste_d = [x for x in range(1, 11) if x % 2 == 0]
print(liste_d)

# équivalent
liste_a = []
for x in range(1, 11):
    if x % 2 == 0:
        liste_a.append(x)
print(liste_a)
```

```
# list comprehension avec les carrés de 0 à 9
ls = [x**2 for x in range(10)]

# dict comprehension avec lettre et leur valeur ascii
dic = {chr(n): n for n in range(65, 91)}
print(dic)

# tuple comprehension avec reduction d'une chaîne
chaine = "abracadabra"
s = {char for char in chaine}
print(s)
```

# Les dictionnaires

- La **mutabilité** est la capacité d'une variable à être modifiée.
- Il ne faut pas la confondre avec la **réassignation**, qui stocke simplement une autre variable à un autre emplacement mémoire.
- Les types non-mutables/immutable sont les **bool**, **str**, **int**, **bytes**, **range**, **tuple** et **frozenset**.
- A contrario les list, dict et set sont par exemple mutables, il est possible de les modifier.
- Il faut cependant faire attention à leur utilisation au sein d'une fonction visant à les altérer, car leur valeur pourrait changer sans qu'on ne veuille.
- L'emplacement mémoire d'une variable mutable ne change pas après sa modification.

```
mon_nombre = 5
print(id(mon_nombre)) # 2358276981104
print(mon_nombre) # 5

mon_nombre += 2
print(id(mon_nombre)) # 2358276981168
print(mon_nombre) # 7
```

```
ma_liste = [1, 2, 3]
print(id(ma_liste)) # 2408503638912
print(ma_liste) # [1, 2, 3]

ma_liste.append(4)
print(id(ma_liste)) # 2408503638912
print(ma_liste) # [1, 2, 3, 4]
```

# list/set/dict comprehension

- Le **condensat** est la valeur obtenue lorsque l'on passe une variable dans un **algorithme de condensation**.
- suite au passage à travers un tel algorithme, **plusieurs valeurs** peuvent obtenir **la même valeur condensat**.
- On appelle ce phénomène une **collision**, et il est plus ou moins fréquent en fonction de l'algorithme de hash utilisé.
- Seuls les objets mutables sont **hashables**, et à condition qu'il ne contiennent pas d'objet immuable.
- Par exemple, il est possible de hasher un **int**, **un string** et **un tuple** :

```
print(hash(10)) # 10
print(hash(2305843009213693961)) # 10, donc collision avec 10
print(hash('toto')) # 1693940491935614836
print(hash((1, 2, 3))) # 529344067295497451
```

# Exercice

- Une année s'est écoulée et la nouvelle édition de la course de module de Tatooine est encore plus captivante.
- Cette année, la position de chaque concurrent est stockée dans une liste. (on y mettre le nom des concurrents)
- Parmi les moments phares de cette édition il y a :
  - Une panne moteur fait passer le premier concurrent à la dernière position.
  - Le second concurrent accélère et prend la tête de la course.
  - Le dernier concurrent sauve l'honneur et dépasse l'avant-dernier module de la course.
  - Un tir de blaster élimine le module en tête de la course.
  - Dans un spectaculaire retournement de situation, un module qu'on pensait éliminé fait son grand retour à la dernière position.
- Créer la fonction `panne_moteur`, modifiant la liste passée en argument de manière à ce que le premier module passe dernier, le deuxième premier et ainsi de suite.
- Créer la fonction `passe_en_tete`, modifiant la liste passée en argument de manière à ce que le premier module passe deuxième et le deuxième premier.
- Créer la fonction `sauve_honneur`, modifiant la liste passée en argument de manière à ce que le dernier module passe avant dernier et l'avant dernier dernier.
- Créer la fonction `tir_blaster`, enlevant le premier concurrent de la liste passée en argument.
- Compléter la fonction `retour_inattendu`, ajoutant un concurrent à la fin de la liste passée en argument.

# Les \*args et \*\*kwargs

- En python, il est possible d'ajouter des paramètres spéciaux précédés avant leurs noms par une ou deux étoiles.
- Leurs noms sont conventionnés, il est important de les nommer args(arguments) et kwargs(keyword arguments)
- Ils permettent d'avoir des fonctions au nombre d'argument variable.

```
def ma_fonction(argument_classique, argument_par_défaut="valeur par défaut", *args, **kwargs):  
    print(argument_classique)  
    print(argument_par_défaut)  
    print(args)  
    print(kwargs)
```

## \*args

- Le paramètre \*arg se transformera en tuple qui aura tous les arguments supplémentaires non nommés en son sein, il sera possible d'y accéder par leur index[]

```
def ma_fonction_avec_args(*args):  
    for arguments in args:  
        print(arguments)  
  
ma_fonction_avec_args(1, "5", True, "Salut", "\na\nb\nc", "Hello World !")  
ma_fonction_avec_args()  
ma_fonction_avec_args("aaa")
```

## \*\*kwargs

- Le paramètre \*\*kwargs se transformera en un dictionnaire qui contiendra un ensemble clé-valeur qui aura tous les arguments ayant un nom associé à une valeur via la syntaxe nom = valeur

```
def ma_fonction_avec_kwargs(**kwargs):
    print(kwargs)
    for kwarg_key, kwarg_value in kwargs.items():
        print(kwarg_key, kwarg_value)

ma_fonction_avec_kwargs(argument1="test", argument2=True, arg3=300)
```

## Exercice

- Réaliser une fonction qui permet, à partir d'une suite de nombres envoyés en paramètres, de retourner une chaîne de caractère correspondant à une syntaxe de ce type :
  - 1-2-3-4...-X
- Vous testerez cette fonction dans le cadre d'un programme de type console, après avoir récupéré ou généré une suite de nombres qui sera envoyée à votre fonction.

# les générateurs

Avancé

- Les **générateurs** sont des **fonctions particulières** qui utilisent le mot clé **yield** dans leur corps.
- Elles sont capables de "**mettre en pause**" leur exécution et de retourner **plusieurs valeurs une à une** grâce à **yield**.
- Il existe une **syntaxe courte** nommée **generator expression** similaire à la **list comprehension**.

```
def gen_int(n):
    for i in range(n):
        yield i

gen_int5 = gen_int(5)
print(next(gen_int5))
print(gen_int5.__next__())

gen_int10 = (i for i in range(10))
print(next(gen_int10))
```

# Exercice

- Réaliser un générateur des lettres de l'alphabet, soit en minuscules, soit en majuscules, en fonction d'un paramètre envoyé à sa création.
- Vous testerez ce générateur dans le cadre d'un programme de type console.

# Le principe des décorateurs

- les décorateurs sont des **fonctions particulières**que l'on peut **appliquer à d'autres fonctions**.
- Ils permettent de réaliser des tâches **avant et après l'exécution de la fonction** et d'en contrôler le comportement.
- On peut s'en servir aussi pour factoriser du code commun à 2 fonctions.
- On les utilise avec la syntaxe suivante :

```
@mon_decorateur  
def hello_world():  
    print("Hello world!")
```

# Définir un décorateur

Avancé

- Lorsque l'on crée des fonctions, il est fréquent que l'on souhaite avoir une fonction similaire à une autre, mais ayant un comportement légèrement différent. Dans ce genre de cas, il faut en général surcharger les fonctions / méthodes, ou rajouter des expressions dans ces fonctions le temps nécessaire.
- En python, il existe ce que s'appelle des décorateurs de fonction, qui permettent d'altérer le fonctionnement des fonctions ou des méthodes de sorte que l'on peut appeler les versions modifiées à la volée sans avoir à retirer les ajouts si l'on veut utiliser de nouveau la version de base des fonctions.

```
def mon_decorateur(fonction):  
  
    def wrap_func():  
        print("Code avant la fonction")  
        fonction()  
        print("Code après la fonction")  
        pass  
  
    return wrap_func  
  
# En commentant simplement ce décorateur, on repasse à la fonction de base  
@mon_decorateur  
def fonction_de_base():  
    print("Code de la fonction")  
    pass  
  
fonction_de_base()
```

# Décorateur avec paramètres

Avancé

- Il est également possible d'utiliser des paramètres sans un décorateur. Le plus souvent, on se sert ainsi des paramètres de type \*arg et \*\*kwargs pour permettre plus de flexibilité.
- Par exemple, ici, nous avons une fonction qui permet d'ajouter des variantes à la décoration.

```
def decorator(*args, **kwargs):
    print("Dans le décorateur")

    def inner(func):
        # code functionality here
        print("Dans la fonction interne")
        print("J'aime ce fruit : ", kwargs['fruit'])

        func()

    return inner
```

```
@decorator(fruit="Banane") # J'aime ce fruit : Banane
def my_func():
    print("Dans la fonction de base")
```

# Exercice

Avancé

- Dans un programme de type console, vous devrez montrer un exemple d'utilisation d'un décorateur qui permettra d'ajouter un nouvel affichage en plus à une fonction permettant déjà d'afficher un message simple dans la console. Le résultat devra donner le résultat ci-dessous.

```
Avant décoration :  
Je suis la fonction de base  
  
Après décoration :  
Je décore la fonction !  
Je suis la fonction de base
```

# Décorateurs multiples

Avancé

- De plus, en Python, on peut décorer une fonction déjà décorée, via l'utilisation de plusieurs décorateurs. De ce fait, on a par exemple ici une fonction décorée puis qui se voit être elle-même décorée.

```
@mon_second_decorateur  
@mon_decorateur  
def fonction_de_base():  
    print("Code de la fonction")  
    pass
```

```
Code avant la fonction 2  
Code avant la fonction  
Code de la fonction  
Code après la fonction  
Code après la fonction 2
```

```
def mon_second_decorateur(fonction):  
  
    def wrap_func():  
        print("Code avant la fonction 2")  
        fonction()  
        print("Code après la fonction 2")  
        pass  
  
    return wrap_func  
  
def mon_decorateur(fonction):  
  
    def wrap_func():  
        print("Code avant la fonction")  
        fonction()  
        print("Code après la fonction")  
        pass  
  
    return wrap_func
```

# Exercice

Avancé

- Via l'utilisation d'une IHM (Interface Homme Machine), vous devrez montrer le fonctionnement des décorateurs multiples et des décorateurs paramétrés. Pour ce faire, vous réaliserez une fonction n'ayant pour fonctionnalité qu'un simple affichage dans la console et qui une fois décorée pourra afficher un message supplémentaire, qui sera personnalisable en fonction du décorateur que l'on choisira d'appliquer. LE décorateur multiple aura pour fonctionnement de décorer une fonction déjà décorée par le décorateur paramétré (ce dernier ajoutant un message personnalisable à la fonction)

Avant décoration :

Je suis la fonction de base

Après décoration personnalisable :

J'écris un message personnalisé : Message perso

Je suis la fonction de base

Après décoration multiple :

Je décore la fonction !

J'écris un message personnalisé : Message perso

Je suis la fonction de base

# Les fichiers JSON

- Pour manipuler des fichiers JSON, il va nous falloir faire appel au module **json**:
  - Ce module est présent de base dans python
- Via ce module, nous disposons ensuite de 4 méthodes principales :
  - **json.dump()** : Qui va sauvegarder des données dans un flux données
  - **json.load()** : Qui va chercher les données dans le flux et les retourner avec typage pour correspondre à python
  - **json.dumps()** : Pour récupérer une chaîne de caractère correspondant au JSON dans le but de l'afficher ou de l'envoyer
  - **json.loads()** : Pour récupérer des données correspondantes à un JSON sous la forme d'une chaîne de caractère.

# Exercice

Avancé

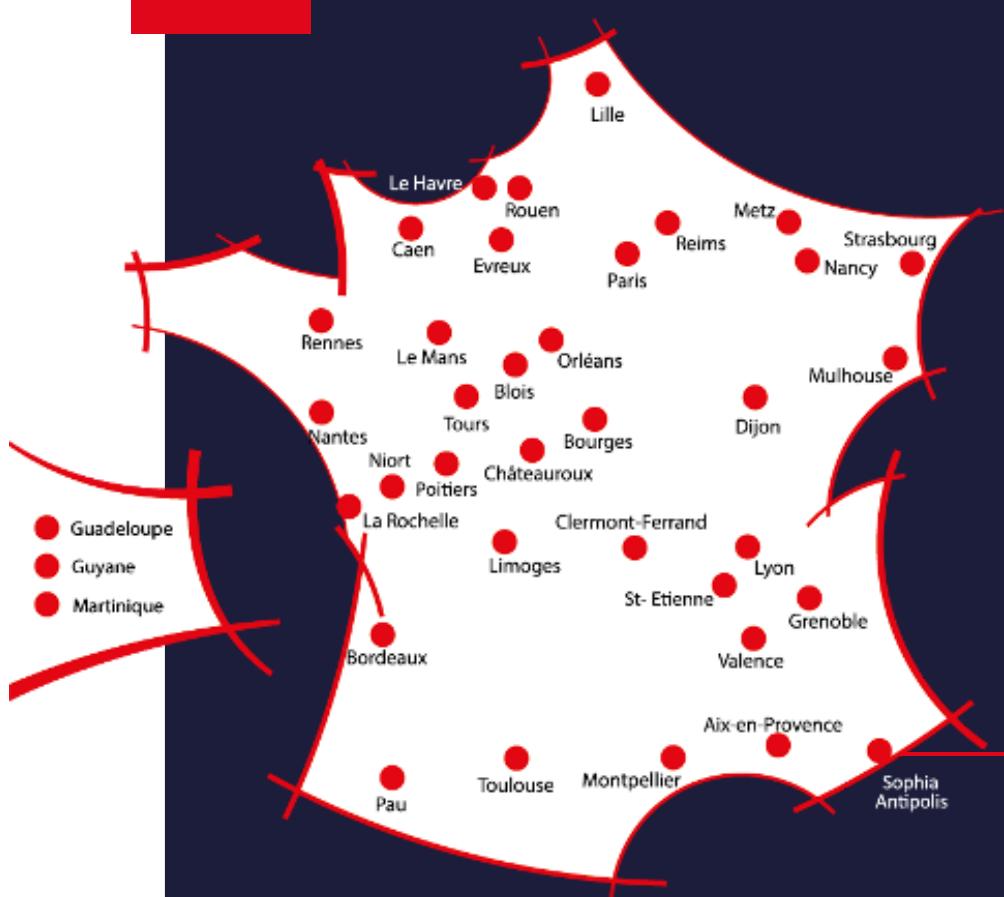
- Par l'utilisation d'un fichier JSON qui sera ouvert, lu et écrit, vous devrez réaliser un logiciel servant à un utilisateur pour stocker des informations sur des chansons. Ces chansons devront posséder comme informations un titre, un artiste, une catégorie et un score (sur 5). Lors de l'ouverture, le programme ouvrira automatiquement le fichier music.json (ou le créera dans le cas où il n'existerait pas) dans le but de l'alimenter de la liste des chansons pour l'utilisateur. La localisation du fichier devra être à la racine du programme dans le dossier nommé datas.

```
== MENU PRINCIPAL ==
1. Ajouter une chanson
2. Voir les chansons
3. Editer une chanson
4. Supprimer une chanson
0. Quitter le programme
Faites votre choix : 1

== AJOUTER UNE CHANSON ==
Titre de la chanson : Titre
Artiste de la chanson : Artiste
Catégorie de la chanson : Catégorie
Score de la chanson (sur 5) : 4
```

**Merci pour votre attention**

**Des questions ?**



Découvrez également  
l'ensemble des stages à votre disposition  
sur notre site [m2iformation.fr](http://m2iformation.fr)

[m2iformation.fr](http://m2iformation.fr)

