Les bases de Python

Ihab ABADI - Antoine DIEUDONNE - Arthur DENNETIERE - Guillaume MAIRESSE

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 3.11 a été publiée en Octobre 2022.
- 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'utilisateurs.

Présentation de Python et ses versions

- Python est :
  - est multiplateforme
  - est gratuit
  - est un langage de haut niveau
  - est un langage interprété
  - est orienté objet
- Usage de Python:
  - Scripts pour automatiser des taches
  - Analyses de données
  - Calcule numérique
  - Développement web
  - …

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
  - /!\ 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 python ou python3, Résultat triple chevrons.
- Pour quitter l'interpréteur, Ctrl + D ou la fonction exit() de python

```
ihab@MacBook-Pro-de-ihab ~ % python3
Python 3.7.9 (v3.7.9:13c94747c7, Aug 15 2020, 01:31:08)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Les Environnements de Développement (IDE)

- Il convient d'installer l'un des IDE suivant :
  - Pycharm (JetBrains, conçut spécialement pour python, avec une version payante améliorée)
  - Visual Studio Code (Microsoft, utile dans de nombreux domaines et avec de nombreuses extensions disponible) /!\
- 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

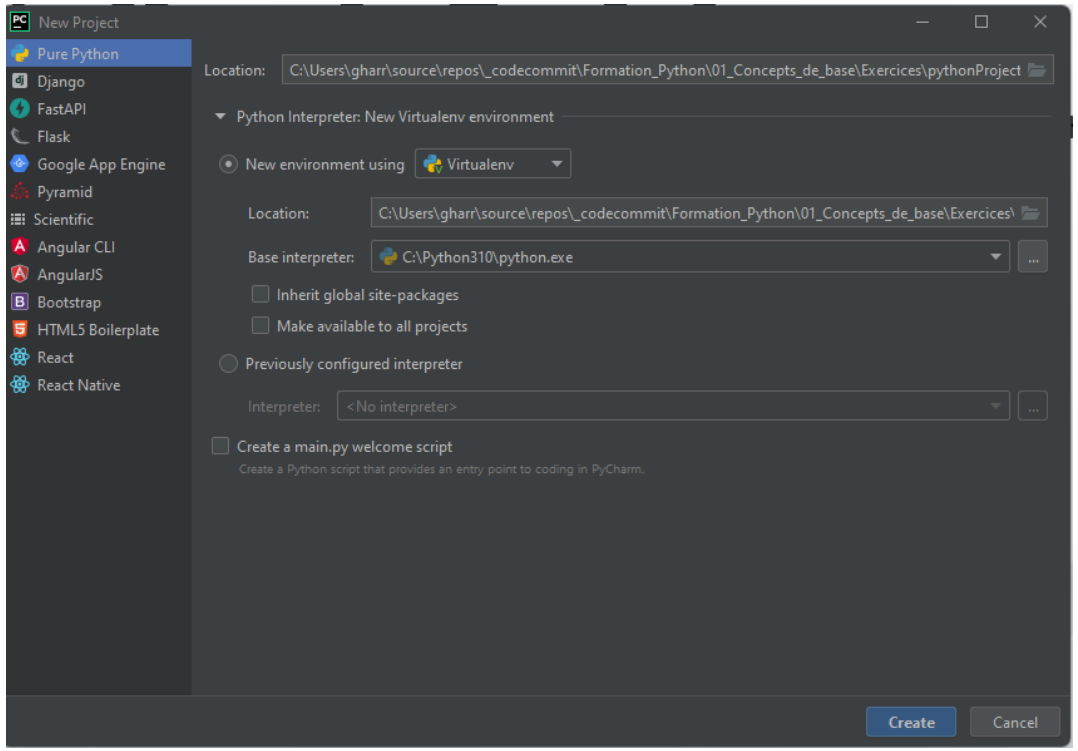
PyCharm

Lors du lancement de PyCharm, appuyez sur **New Projet**, puis sélectionnez l'emplacement de stockage du projet Python sur votre ordinateur via l'input **Location**.

Vous pouvez ensuite spécifiez l'environnement virtuel de lancement de l'interpréteur Python, comme cela est marqué dans l'exemple ci-après (celui-ci étant ici configuré sur **Virtualenv**, l'environnement virtuel de base).

Un environnement virtuel permet d'installer des modules et des packages de façon spécifique au projet sans les installer de façon globale (sur la machine en elle-même).

D'autres paramètres sont disponibles, mais nous n'en ferons pas usage dans ce cours. Appuyez simplement sur **Create** dans le but de valider la création du projet Python.

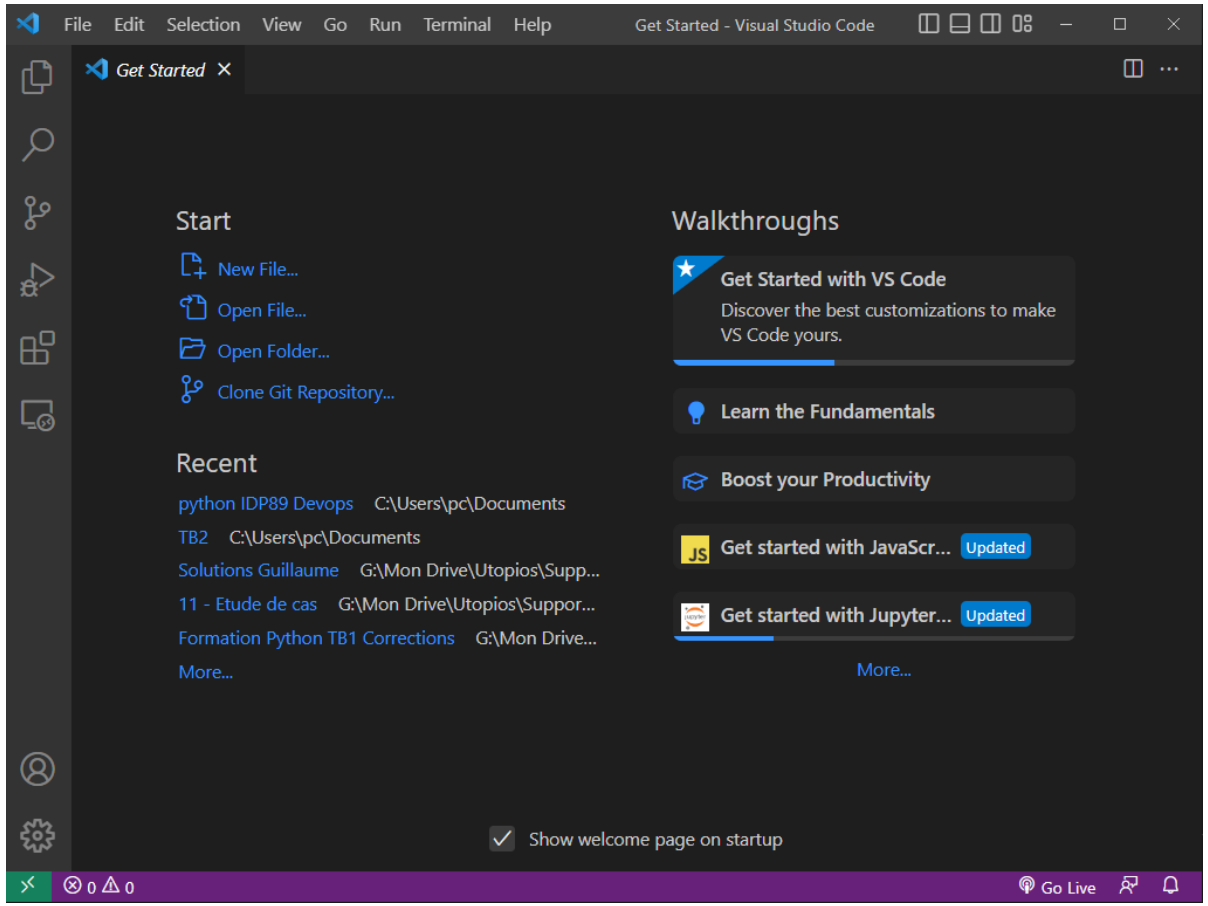


PyCharm

- Pycharm génère un environnement virtuel stocké dans venv (packages et modules installés).
- Il permet d'ajouter des scripts python dans notre dossier (click droit, add python file).
- Il permet l'exécution d'un script par un **clic droit + run**, **CTRL + SHIFT + F10** ou encore en ajoutant une **configuration de lancelment** (en haut à droite).

Visual Studio Code

- Dans le menu latéral on trouve l'explorer, il permet d'avoir un aperçu des fichiers du dossier ouvert
- Lorsque l'on se situe sur un fichier python, un bouton 'play' est affiché en haut à droite, il lance un terminal avec une commande équivalente à **py mon_script.py**



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 faciliter la lecture.
- Python suit la norme PEP8
- Les IDE ont souvent un auto-formatage qui respectera une grande partie de ces règles. Le raccourcis pour formater automatiquement change selon le logiciel.

Les variables

Les **variables** ont pour but de **stocker** des informations dans la mémoire vive de l'ordinateur.

Les **variables** peuvent être de plusieurs **types**, qui sont parmi les plus fréquents :

- 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 **float**s, servant à stocker des **nombres à virgule** flottante (décimaux)
- Les chaines de caractères strings/**str** qui permettent de stocker du **texte**.
- Les **bool**éens, 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_int = 514  # Variable de type integer
mon_float = 3.14  # Variable de type float
mon_complex = 547J  # Variable de type complex

mon_int_ex: int = 514
mon_float_ex: float = 3.14
mon_complex_ex: complex = 547J
```

```
ma_string = "Je suis une string"
ma_string_2= 'Je suis aussi une string'
ma_string_multiline = """Je suis une string gardant
    L'indentation et les sauts
    de lignes"""
```

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.
- 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 mettra **entre les parenthèses** seront appelées **paramètres** ou **arguments**.

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

# 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 chaine à afficher en paramètre.

```
# Une récupération passe par l'utilisation de la fonction input()
# qui récupèrera toujours une chaine de caractères
ma_recuperation = input("Veuilliez entrer une valeur SVP : ")

# Un affichage console passe par l'utilisation de la fonction print()
# Qui peut avoir plusieurs paramètres dans le but d'avoir una variable en plus du texte
print("Vous avez entrer comme valeur : ", ma_recuperation)
```

Ces deux processus amènent rapidement à **deux composantes essentielles** de la programmation console, qui sont :

- pour **l'affichage :** le **formatage** des str
- pour **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 :

- La méthode **.format()** qui est liée au type string. Pour s'en servir il suffit de placer **deux accolades {}** dans une chaine de caractère qui serviront **d'emplacement** prévus pour **l'affichage** des variables paramètres de la méthode format.
  Ces marqueurs peuvent être **numérotés** (en commençant par 0) pour indiquer quel paramètre de la méthode sera affiché.
  On pourra ajouter un formatage spécifique qui est le même que pour les f-strings (cf. diapo suivante).

```
nombre_A = 3.141592653589793
print("nombre_a = {0:0.2f}".format(nombre_A)) # nombre_a = 3.14

nombre_B = 2
nombre_C = 21
print("{0:2d} {1:3d} {2:4d}".format(nombre_B, nombre_B**2, nombre_B**3))  #  2   4    8
print("{0:2d} {1:3d} {2:4d}".format(nombre_C, nombre_C**2, nombre_C**3))  # 21 441 9261
```

- Le **%-formating**, maintenant déprécié et utilisé de nos jours que dans de rares cas (formatage de dates, de requêtes SQL)
- Les **f-strings** sont des chaines 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}")  # La valeur de nombre_a vaut 3.14
```

Fonctionnement des f-strings

AVANCÉ

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

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

- Il s'agit en réalité d'un formatage rapide et simple d'utilisation pour les valeurs utilisées par le f-string
- Ici la valeur sera :
  - f : avec un formatage dédié aux réels
  - ^ : centrée
  - 7 : Dans un espace de 7 caractère 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)
- Documentation sur les f-strings

Les raw-strings

- Similairement au f-strings, il existe aussi en python les **raw-strings**

- Ce sont des chaines où les **caractères spéciaux** comme le **backslash ** ne sont pas interprétés

- Ils facilitent la saisie des chemins de fichier ou des regex par exemple

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

print(r"\"")
# print(r""")
print(r'\'')
# print(r'\')
print(r'\\')
print('\\')
```

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 se 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**.

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

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

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("Veuilliez entrer un nombre SVP : "))  # 25
print(f"Mon nombre vaut {mon_nombre} et est de type {type(mon_nombre)}")
# Mon nombre vaut 25 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 un False dans une condition (cf : partie block conditionnels)

Exercice – (5 min)

EXERCICE

- Ecrire 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, etc…
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
print(mon_resultat)  # 8
mon_resultat = mon_resultat - 4  # mon affecte à la variable mon_resultat sa valeur moins 4
print(mon_resultat)  # 4
mon_resultat *= 2  # on multiplie la valeur de la variable mon_resultat par 2
print(mon_resultat)  # 8
mon_resultat /= 2  # On divise (division entière) la valeur de mon_resultat par 2
print(mon_resultat)  # 4.0
mon_resultat //= 2  # On divise (division décimale) la valeur de mon_resultat par 2
print(mon_resultat)  # 2.0
mon_resultat **= 3  # On passe mon_resultat à la puissance 3
print(mon_resultat)  # 8.0
mon_resultat %= 3  # On consert le reste de la division de mon_resultat par 3
print(mon_resultat)  # 2.0
```

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 – (5 min)

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
- « **==** » : Egal à
- « **!=** » : Différent de

Le résultat des opérations utilisant ces opérateurs sera un **bool**éen (**True**/**False**) permettant de savoir si le résultat de la comparaison est vrai ou faux

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 **bool**éen.

```
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 – (5 min)

EXERCICE

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

Autres opérateurs

On retrouve d'autres types d'opérateurs notables :

- **D'identité** : « **is** » et « **is not** » permettront de vérifier le type d'une variable (similaire à la fonction **isinstance()**)
- **D'appartenance** : « **in** » et « **not in** » permettront de savoir si une variable est membre d'une autre
- Liés au **binaire** : les opérateurs **&, |, ^, ~, << et >>** permettent de travailler avec les valeurs binaires de nos variables
  /!\ il ne faut pas les confondre avec les opérateurs logiques AND, OR et NOT

Les structures conditionnelles

Si l'on veut permettre à notre 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 enchainer 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**.

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

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 ne les met que si on en a réellement besoin.

Exercice – (10 min)

EXERCICE

- Ecrire 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 **négative** 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 **supérieure à 100** l'eau est à l'état **gazeux**

- Il est possible de réaliser cet exercice sans if imbriqués grâce au elif

Exercice – (15 min)

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)

L’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.

Le 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**

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

```
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 :

```
variable = <Valeur si vrai> if <condition> else <valeur si faux>
```

- Exemple avec l'exercice température vu précédemment:

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

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 mettra **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, la norme est de la nommer par un underscore « _ ».

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 du 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 – (15 min)

EXERCICE

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

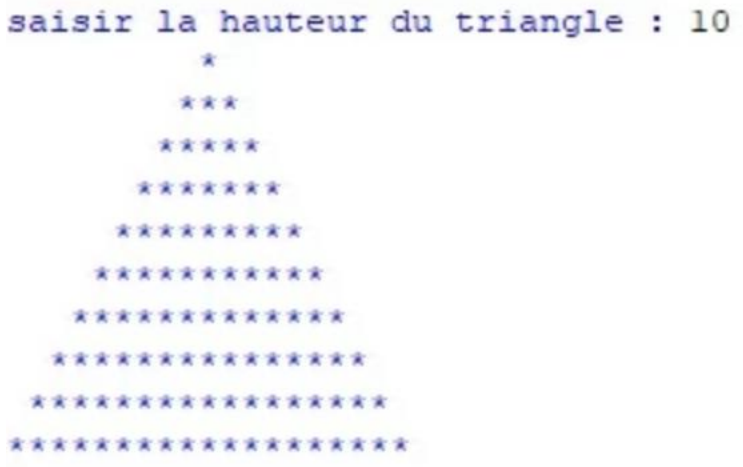
Table de multiplication

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 – (15 min)

EXERCICE

- Écrire un programme qui permet d'afficher un triangle isocèle formé d'étoile *.
- 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



Exercice – (15 min)

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 400 m ?
- 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 400 m au minimum pour que l’épaisseur dépasse 0.1mm ?

Exercice – (15 min)

EXERCICE

- Réalisez 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 les 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 dépends du **nombre de paramètres**.
- Une particularité des fonctions en Python est que vous n'êtes **pas obligé** de **préciser le type** des **paramètres**, dès lors que les **opérations** que vous effectuez avec ces eux sont **valides**. Python est en effet connu comme étant un langage au « **typage dynamique** ».
- Il est aussi possible de passer un ou plusieurs argument(s) de manière **facultative** et de leur attribuer une valeur par **défaut**. Pour cela on utilise le **=**.

```
def carre(nombre: int):
    return nombre**2

res = carre(2) # retourne 4
```

```
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 ladite fonction.
- Une variable est 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 – (5 min)

EXERCICE

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

Exercice – (5 min)

EXERCICE

- Ecrire 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 soustrait 2 et 1 »
- Vous afficherez le résultat à l'aide de la fonction print()

Exercice – (5 min)

EXERCICE

- Ecrire 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 – (5 min)

EXERCICE

- Ecrire une fonction compter_lettre_a
- Cette fonction prendra en paramètre une chaîne
- Créer une boucle qui parcourt les lettres de la chaine 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

- Ecrire une autre fonction sans boucle qui utilisera **count** à la place.

Exercice – (20 min)

EXERCICE

Ecrire un programme qui permet de saisir une chaine 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. Ecrire une fonction *vérification_adn* qui permet de renvoyer la valeur True si la chaine d'ADN est valide, False si elle est invalide
2. Ecrire une fonction *saisie_adn* qui récupère une saisie, vérifie sa validité et renvoie une chaine d'ADN valide sous forme de chaine
3. Ecrire une fonction *proportion* qui reçoit deux paramètres : une chaine 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 : crypt, 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**
- Exemple: fichier **circle.py**

```
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 connaitre le contenu

Modules

- **L'importation** permet à un script d'utiliser le code d'un **autre module**
- Syntaxes d'importation :
  1. `import <nom_du_module>`
  2. `from <nom_du_module> import <fn1>, <fn2>, …`
  3. `from <nom_du_module> import *`
- Syntaxe d'accès à un membre d'un module importé
  - `<module>.<mon_membre>` avec la méthode 1 (le module est dans une variable)
  - `<Mon_membre>` avec la méthode 2 et 3
- Il est possible d'ajouter un alias à un module ou un membre
  - `import <nom_du_module> as mod_1`
  - `from <nom_du_module> import <fn1> as f1, <fn2> as f2`

```
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 chaine **"__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 ce script

```python
# IMPORTS
import math

# DEFINITION DES FONCTIONS
def addition(a, b):
    return a + b


# fonction main()
# réutilisable dans un autre module
def main():
    print(addition(40, 3))


if __name__ == "__main__":
    #appel de la fonction main
    main()
```

Exercice – (10 min)

EXERCICE

- Restructurer le script ADN avec la structure vue précédemment
- Ecrire 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 laissez vide
- Mais lorsque nous importons un package, seuls ses modules immédiats sont importés, pas les sous-packages. 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 : Ecriture
  - 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().

Ecrire 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 – (10 min)

EXERCICE

- Ecrire 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.**

```python
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)
```

```python
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**.

```
ma_list = []
print(ma_list)  # []
ma_list = [1, 2, 3]
print(ma_list)  # [1, 2, 3]
```

- 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'occurrence d'un élément
  - **index(element)** : index de la première occurrence

```
ma_list = [2, 1, 3]
print(ma_list)  # [2, 1, 3]
ma_list.sort()
print(ma_list)  # [1, 2, 3]
ma_list.append(4)
print(ma_list)  # [1, 2, 3, 4]
ma_list.extend([5, 6])
print(ma_list)  # [1, 2, 3, 4, 5, 6]
ma_list.remove(4)
print(ma_list)  # [1, 2, 3, 5, 6]
ma_list.pop(2)
print(ma_list)  # [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]
1
2
3
4
5
```

Exercice – 20 min

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) :
  - **saisi** avant la saisie des 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)

```
Veuilliez entrer une note entre 0 et 20 compris (une note négative stoppera la saisie) : 12
Veuilliez entrer une note entre 0 et 20 compris (une note négative stoppera la saisie) : 11
Veuilliez entrer une note entre 0 et 20 compris (une note négative stoppera la saisie) : 9
Veuilliez entrer une note entre 0 et 20 compris (une note négative stoppera la saisie) : 8
Veuilliez entrer une note entre 0 et 20 compris (une note négative stoppera la saisie) : 7
Veuilliez 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 délimiter
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 – (10 min)

EXERCICE

- Ecrire 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, Filter, 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 des listes

- **sorted** : trier la liste selon certains critères
- **filter** : filtrer les éléments de la liste selon un prédicat (fonction)
- **map** : créer une nouvelle liste avec tous les éléments transformés par une fonction
- **reduce** (module **functools**) : réduire la liste à une seule valeur à l'aide d'une fonction

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:
- **nom_tuple = ()** ou **nom_tuple = 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'élément d'un tuple
  - **tuple.count(element)** => nombre d'occurrence 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 aussi l'**unpacking.**
  Exemple : **var1, var2 = (1, 2)**

Exercice – (15 min)

EXERCICE

- Ecrire un programme se servant d'une fonction retournant, à partir de deux nombres lui étant envoyés en paramètres :
  - La somme
  - La difference
  - 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 cependant…

```
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
```

- 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.

```
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 si il est présent, sinon erreur
- **discard(element)** : supprime l'élément si 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 le set2 est compris dans le set
- On retrouve aussi les méthodes d'**union |**, d'**intersection &,** de **différence -** et de **différence symétrique ^.**

Exercice – 20 min

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 la **clé** associée entre crochets.
A l'aide du mot clé **del**, on peut supprimer une entrée.

```
mon_dict = {'k1': 'valeur un', 'k2': 258963, 'k3': 3.14, 'k4': {1: 'blabla'}}
print(mon_dict)  # {'k1': 'valeur un', 'k2': 258963, 'k3': 3.14, 'k4': {1: 'blabla'}}
print(mon_dict['k3'])  # 3.14
print(mon_dict['k4'][1])  # blabla
```

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, valeur)

```
print(mon_dict.values())  # dict_values(['valeur un', 258963, 3.14, {1: 'blabla'}])
print(mon_dict.keys())  # dict_keys(['k1', 'k2', 'k3', 'k4'])

print(mon_dict.items())  # dict_items([('k1', 'valeur un'), ('k2', 258963), ('k3', 3.14), ('k4', {1: 'blabla'})])
```

Via **l'unpacking** des tuples et la méthode **.items()**, il est possible d'afficher les informations du dictionnaire plus facilement :

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

Les dictionnaires

Pour parcourir un dictionnaire, on utilise généralement une boucle **for**, telle que :

on peut également accéder à la clé en complément de la valeur, de cette façon :

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

```
{'key1': 123, 'key2': '456', 'key3': [7, 8, 9]}
key1: 123
key2: 456
key3: [7, 8, 9]
```

Les clés des dictionnaires sont forcément des types **immutables**.

Exercice – (20 min)

EXERCICE

Avec des variables de type **dictionnaire** dans une liste, vous réaliserez un logiciel pour **stocker** une série **d'adresses**

Celles-ci devront avoir :

- **un numéro de voie**
- **un complément**
- **un intitulé de voie**
- **une commune**
- **un 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 ===
Veuilliez entrer le numéro de voie SVP : 96
Veuilliez entrer le complément d'adresse SVP : Apt 47
Veuilliez 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 **iterable** est un objet sur lequel on peut **itérer** (ex: list, range, …) :
  <var> = [<expression> for <element> in <iterable>]
- Il est aussi possible d'ajouter un filtre avec un **if** après l'iterable (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 chaine
chaine = "abracadabra"
s = {char for char in chaine}
print(s)
```

Mutable / Immutable

- La **mutabilité** est la capacité d'une variable a 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 le veuille !
- **L'emplacement mémoire** d'une variable **mutable ne change pas** après sa modification, comme dans l'exemple ci-contre

```
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]
```

Le condensat (Hash)

- 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 de 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'**ils ne contiennent pas d'objet immutable.**
- Par exemple, il est possible de hasher un **int**, un string et un **tuple**, tel que :

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

Exercice – (20 min)

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 mettra 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_inatendu**, **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(agument1="test", argument2=True, arg3=300)
```

Exercice – (15 min)

EXERCICE

- Réaliser une fonction qui permet, à partir d'une suite de nombres envoyés en paramètres, de retourner une chaine 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 envoyé 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 – (15 min)

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 fonction / méthodes, ou rajouter des expressions dans ces fonctions le temps nécessaire.

En Python, il existe ce qui 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 des paramètres

AVANCÉ

Il est également possible d'utiliser des paramètres dans un décorateur. Le plus souvent, on se sert ainsi des paramètres de type *args et **kwargs pour permettre plus de flexibilité.

Par exemple, ici, nous avons une fonction qui permet d'ajouter des variantes à la décoration :

```python
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
```

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

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

Exercice – 10 min

AVANCÉ

EXERCICE

- 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 à son tour :

```python
@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
```

```python
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 – (15min)

AVANCÉ

EXERCICE

- Via l'utilisation d'une IHM (Interface Homme Machine), vous devrez montrer le fonctionnement des décorateurs multiples et des décorateurs paramétrés. Pour se 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 fonction 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 le Python

Via ce module, nous disposons ensuite de 4 méthodes principales :

- **json.dump()** : Qui va sauvegarder des données dans un flux de données (un fichier ouvert)
- **json.load()** : Qui va chercher les données dans le flux et les retourner avec typage pour correspondre au Python
- **json.dumps()** : Pour récupérer une chaine 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 chaine de caractère

Exercice – (60 min)

AVANCÉ

EXERCICE

- 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, un score (sur 5) et une durée (en minutes et secondes). Lors de l'ouverture le programme ouvrira automatiquement le fichier music.json (ou le créera dans le cas où il n'existe pas) dans le but d'alimenter la liste des chansons pour l'utilisateur. La localisation du fichier devra être à la racine du programme dans un 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
```

Programmation Orientée objet

Les classes et instances

- Une **classe** est le "**moule**" servant à la fabrication des **objets/instances** du **type correspondant à cette classe**.
- On peut voir la classe comme le **récapitulatif** des **éléments** que vont posséder tous les **objets** de ce type. Par exemple un Chien a un âge, un nom, une race, etc…
- Tous ces éléments se retrouveront dans la classe et seront (ou non) définis à **l'instanciation** d'un objet de ce type.
- En python on définit une class avec le bloc **class**. Tout les blocs **def** à l'intérieur créeront des **méthodes relatives à la classe** et non des fonctions.

```
class Chien:
    """Représentation d'un chien"""

    def __init__(self, nom, age, race):
        self.nom = nom
        self.age = age
        self.race = race

    def aboyer(self):
        print(f"Wouf Wouf {self.nom}")
```

Le constructeur __init__

Le **constructeur** est le point d'entrée pour la création d'une **instance/objet** du **type de la classe** (**instanciation**).
Il s'agit d'une **méthode** dite **Dunder** ou **Magique** (dont le nom commence et fini par deux caractères **underscore** : __nom_methode__)

```
class Chien:
    def __init__(self, nom, age, race):
        self.nom = nom
        self.age = age
        self.race = race
```

Le constructeur est appelé lorsque l'on souhaite instancier une classe, on écrit le **nom de la classe** et non __init__:

```
chien_1 = Chien("Rex", 12, "Berger Allemand")
```

Une fois la **variable** renvoyant vers **l'instance** de Chien créée, on peut la manipuler et utiliser ses méthodes :

```
chien_1.aboyer() # Wouf wouf Rex
chien_2.aboyer() # Wouf wouf Milou
```

Le paramêtre de méthode ‘self’

- Pour référencer **l’objet instancié** lors de la déclaration des méthodes d’une classe, on utilise un **paramètre supplémentaire** en **premier**.
- La norme est de le nommer **self.**
- Lors de l’appel de ces méthodes, **on ne devra pas renseigner cet argument**.

```
def __init__(self, nom, age, race):
    self.nom = nom
    self.age = age
    self.race = race

def aboyer(self):
    print(f"Wouf Wouf {self.nom}")
```

```
chien_1 = Chien("Rex", 12, "Berger Allemand")
```

```
chien_1.aboyer() # Wouf wouf Rex
chien_2.aboyer() # Wouf wouf Milou
```

Les attributs

Un **attribut** de classe est une **variable** qui est associée à cette classe.
Il est en général **définit** et **affecté** dans le **constructeur**.
Il peut être **accédé** et **réaffecté** via la notation **variable_objet.attribut**.

```
chien_1.age = 6
print(f"Le chien s'appelle {chien_1.nom}, il a {chien_1.age} ans")
print(f"{chien_1} est donc né en {date.today().year - chien_1.age }")
```

Un **objet** est une valeur référence, c'est-à-dire qu'il est **mutable** et qu'on peut ainsi **le passer en paramètre de fonction ou de méthode** et voir **s'opérer des changements** en cas de modifications éventuelles de ses attributs.

```
def change_nom(chien, nouveau_nom):
    chien.nom = nouveau_nom

chien_1.nom = "Rex"
print(chien_1.nom) # Rex
change_nom(chien_1, "Bill")
print(chien_1.nom) # Bill
```

Les attributs implicites

AVANCÉ

Ces attributs sont créés **par défaut** lors de la manipulation des classes.
Ils utilisent via la syntaxe **Dunder** (double underscore).

Pour la classe on a :

- **__name__ :** le nom de la classe
- **__doc__ :** commentaire associé à la classe
- **__dict__ :** le dictionnaire des attributs statiques
- **__bases__ :** un tuple des classes dont celle-ci hérite
- **__module__ :** contient le nom du module dans lequel la classe a été définie

Pour l'instance on a :

- **__class__ :** la classe de l'objet
- **__dict__ :** la liste des attributs d'instance

```
class MaClasse:
    """une classe"""
    test = 0
    def __init__(self):
        self.test1 = 1

cl = MaClasse()
# Classe
print(MaClasse.__name__)    # MaClasse
print(MaClasse.__doc__)     # une classe
print(MaClasse.__dict__)    # {'test': 0, .....}
print(MaClasse.__bases__)   # (<class 'object'>,)
print(MaClasse.__module__)  # __main__

# Instance
print(cl.__class__)             # <class '__main__.MaClasse'>
print(cl.__class__.__name__)    # MaClasse
print(cl.__dict__)              # {'test1': 1}
print(cl.__doc__)               # une classe
```

Les méthodes

Une **méthode** est l'équivalent d'une **fonction** qui est **associée** à un **objet** ou à une **classe**.
Pour faire **appel** à une méthode, il faudra utiliser la notation **Classe.méthode()** ou **objet.méthode()**.
Une méthode peut accéder aux **attributs** de **l'objet auquel elle est associée** en passant encore une fois par le **paramètre self**, qu'elle doit avoir en tant que **premier paramètre** :

```
def aboyer(self):
    print(f"Wouf wouf {self.nom}")
```

Une méthode peut réaliser tout ce qu'une fonction faisait de base, mais est en général utilisée pour **éviter d'avoir à passer en argument** des **valeurs** qui sont **déjà dans les attributs de l'objet**:

```
def afficher(self):
    print(f"Mon chien a {self.age} ans, il s'appelle {self.nom} de la race {self.race}")
```

Une méthode participe ainsi activement à la réalisation d'un code plus propre et à la mise en place du **DRY (Don't Repeat Yourself)** dans le cadre d'un programme.

Exercice – (15 min)

EXERCICE

1. Créer une classe **Gateau**
2. Ajouter les attributs suivants et les initialiser dans le constructeur :
   - **nom gâteau** : str
   - **temps cuisson** : int
   - **liste ingrédients** : list de str
   - **étapes recettes** : list de str
   - **nom du créateur** : str
3. Ajouter une méthode qui affiche les ingrédients de la recette
4. Ajouter une méthode qui affiche les étapes de la recette
5. Instancier un objet gâteau qui affiche les ingrédients ainsi que les étapes de préparation du gâteau

Exercice – (30 min)

EXERCICE

1. Créer une classe nommée CompteBancaire qui représente un compte bancaire, ayant pour attributs :
   - numeroCompte (type numérique)
   - nom (nom du propriétaire du compte du type chaine)
   - solde
2. Créer un constructeur ayant comme paramètres : numero_compte, nom, solde
3. Créer une méthode Versement() qui gère les versements
4. Créer une méthode Retrait() qui gère les retraits
5. Créer une méthode Agios() permettant d'appliquer les agios à un pourcentage de 5 % du solde
6. Créer une méthode afficher() permettant d'afficher les détails sur le compte

Les propriétés

AVANCÉ

- Les propriétés sont **trois méthodes magiques** qui sont appelées en cas de **récupération (getattr)**, **d'affectation (setattr)** ou de **suppression (delattr)** d'un attribut.

- Il est ainsi possible de **surcharger/override** ces méthodes magiques pour **en modifier le fonctionnement**.

- Cela évite ainsi d'avoir à répéter des lignes de codes et également la création de méthodes destinées à contrôler et à modifier les affectations ou les récupérations d'attributs d'objets (appellés getters et setters).

```python
ma_temperature = Temperature(37.5)
ma_temperature.celcius = 25
print(ma_temperature.fahrenheit)  # 99.5
```

```python
class Temperature:
    def __init__(self, value):
        self.value = value

    def __getattr__(self, name):
        if name == 'celsius':
            return self.value
        if name == 'fahrenheit':
            return self.value * 1.8 + 32
        raise AttributeError(name)

    def __setattr__(self, name, value):
        if name == 'celsius':
            self.value = value
        elif name == 'fahrenheit':
            self.value = (value - 32) / 1.8
        else:
            super().__setattr__(name, value)
```

Les propriétés

AVANCÉ

- Python fournit également un décorateur **@property**. Il permet d'appeler une **méthode** comme si on tentait **d'accéder** à un **attribut de l'objet** portant le **même nom**.

- Le décorateur **@<prop>.setter** permet d'appeler la méthode **méthode** comme si on tentait **de définir l'attribut de l'objet** portant le **même nom**.

```
@property
def nom(self):
    return self._nom

@nom.setter
def nom(self, nom):
    self._nom = nom
```

```
objet.nom = "le nom"
print(objet.nom)
```

```
@property
def age(self):
    today = date.today()
    age = today.year - self.birth_date.year  - ((today.month, today.day) < (self.birth_date.month, self.birth_date.day))
    return age
```

Les attributs de classe

En plus des **attributs** liés à **un objet/instance**, il est possible de faire appel à ce qu'on appelle des **attributs de classe**.
Ils sont **partagés** par l'ensemble **des objets de ce type**, il sont **liés à la classe elle-même**.
On peut par exemple se servir des attributs de classe pour compter facilement les objets instanciés de cette classe ou pour accéder à des valeurs communes à tous les éléments de ce type.
Pour accéder à un attribut de classe, on doit se servir de la syntaxe **Classe.attribut**.

```
class Chien:
    instances_chien = 0
    nom_latin = "Canis lupus familiaris"

    def __init__(self, age, nom, race):
        Chien.instances_chien += 1
        self.age = age
        self.nom = nom
        self.race = race
```

```
print(f"Il y a {Chien.instance_chien} instances de chiens dont le nom latin est : {Chien.nom_latin}")
#Il y a 2 instances de chiens dont le nom latin est : Canis lupus familiaris
```

Les méthodes de classe

- Une **méthode de classe** est **une méthode** qui est **liée à la classe** et non à **l'objet**.
- Pour en définir une on utilise le décorateur **@classmethod**
- Elle a accès **à l'état de la classe** par le biais d'un paramètre que l'on nomme **cls** par convention, il est en **premier** (à la place de self) et **pointe vers la classe** et non l'objet.
- On accèdera aux attributs de classe avec **cls.attribut**
- Pour faire appel à une méthode de classe on utilise la syntaxe suivante : **Classe.méthode_de_classe**

```
@classmethod
def afficher_nombre_chiens(cls):
    print(f"Il y a {cls.nombre_chiens} instanciés")
```

Les méthodes statiques

- Une **méthode statique** ne reçoit pas de premier argument implicite (self ou cls).
- Une méthode statique est également une méthode qui est **liée à la classe** et non à l'objet.
- Cette méthode ne peut pas accéder ou modifier l'état de la classe.
- Elle est destinée à avoir un comportement qui ne change pas, elle est comme une fonction classique en soit.

Pour faire une méthode statique, il faut donc utiliser le décorateur **@staticmethod** et on l'appellera dans le cœur de notre programme via (comme pour les méthodes de classe) la syntaxe **Classe.méthode()** :

```
@staticmethod
def seuil_chien(max):
    print(f"Il y a {max - Chien.nombre_chiens} disponibles dans le refuge")
```

```
Chien.seuil_chien(10) # Il y a 8 places disponibles dans le refuge
```

Exercice – 45 min

EXERCICE

1. Créer une classe **WaterTank** qui possédera **les attributs d'instance** suivants :
   a. **Poids** de la citerne à **vide** : float
   b. **Capacité maximale** : float
   c. **Niveau de remplissage** : float
2. Créer les **méthodes** suivantes propre à chaque instance de classe :
   a. Méthode indiquant le **poids total**
   b. Méthode pour **remplir la citerne** avec un **nombre de litre d'eau**
   c. Méthode pour **vider la citerne** d'eau d'un **nombre de litre d'eau**
3. Créer un **attribut de classe** qui contiendra **la totalité des volumes d'eau** des citernes
4. Testez votre programme

Différences entre méthode de classe et méthode statique

Méthode de classe	Méthode statique
Une méthode de classe prend comme **premier paramètre cls** (la classe)	Une méthode statique n'a **pas d'arguments par défaut**
Une méthode de classe peut **accéder et modifier l'état d'une classe via le paramètre cls**	Une méthode statique ne peut pas **accéder ou modifier l'état d'une classe sans utiliser la syntaxe avec le nom de la classe**
La méthode class prend **la classe comme paramètre pour connaître l'état de cette classe (cls)**	Les méthodes statiques **ne connaissent pas l'état de la classe**. Ces méthodes sont utilisées pour effectuer certaines tâches utilitaires en prenant certains paramètres, comme des fonctions.
Utilisation du décorateur **@classmethod**	Utilisation du décorateur **@staticmethod**

L'héritage

L'héritage est un mécanisme fortement utilisé dans la programmation orienté objet.
Une classe peut **hériter** d'une **autre classe** et possédera **les méthodes** et **les attributs** de celle-ci.
On parle alors de **classe fille/enfant** et de **classe mère/parent**.

Pour **réaliser un héritage** en Python il suffit **d'ajouter des parenthèses** après le nom de la classe que l'on créé et d'y **ajouter la classe dont l'on souhaite hériter**.

Par exemple ici **Chien va hériter de Mammifère** et ainsi avoir accès à ses méthodes et attributs . Cela est correct sémantiquement car on peut dire qu'**un Chien EST un Mammifère**.

```python
class Mammifere:
    nom_latin = "Mamma"
    nombre_mammifere = 0

    def __init__(self):
        Mammifere.nombre_mammifere += 1

class Chien(Mammifere):
    nom_latin = "Canis Lupus Familiaris"

    def __init__(self, nom, age, race):
        super().__init__()
        self.nom = nom
        self.age = age
        self.race = race

mon_chien = Chien("Rex", 4, "Berger Allemand")
print(Mammifere.nombre_mammifere) # 1
```

L'utilisation de la méthode super()

Lors d'un **héritage**, il est possible **d'accéder aux attributs et aux méthodes de la classe mère**.
Si l'on souhaite avoir accès à la méthode **calc_age(annee)** de la classe **Mammifère** pour se servir du résultat dans la classe enfant, on doit utiliser le mot-clé **super()** pour **accéder à la classe parent**, puis la syntaxe **super().nom_méthode()** pour en **appeler la méthode**.

Le mot clé **super()** est également utilisé dans le cadre d'un **constructeur** pour faire appel au **constructeur de la classe parent** qui pourrait avoir besoin de paramètres, comme ci-dessous :

```
class Personne:
    def __init__(self, nom, prenom, age):
        self.nom = nom
        self.prenom = prenom
        self.age = age

class Enfant(Personne):
    def __init__(self, nom, prenom, age, jouet):
        super().__init__(nom, prenom, age)
        self.jouet = jouet
```

```
class Mammifere:
    nom_latin = "Mamma"
    nombre_mammifere = 0

    def __init__(self):
        Mammifere.nombre_mammifere += 1

    def calculer_age(self, annee) -> int:
        return date.today().year - annee

class Chien(Mammifere):
    nom_latin = "Canis Lupus Familiaris"

    def __init__(self, nom, age, race):
        super().__init__()
        self.nom = nom
        self.age = age
        self.race = race

    def age_chien(self) -> int:
        return super().calculer_age(self.age)
```

Exercice – (15min)

EXERCICE

1. Écrire une classe **Rectangle** en langage Python, permettant de construire un rectangle doté **d'attributs longueur et largeur**.
2. Créer une méthode **perimetre()** permettant de calculer le périmètre du rectangle et une méthode **surface()** permettant de calculer la surface du rectangle
3. Créer une classe fille **Parallélépipède héritant de la classe Rectangle** et dotée en plus d'un **attribut hauteur** et d'une autre méthode **volume()** permettant de calculer le volume du Parallélépipède.
4. Surcharger les méthodes **périmètre()** et **surface()** du **Parallélépipède** pour avoir les bon résultats pour un parallélépipède (longueurs des arrêtes et surfaces des faces).

/!\ Sémantiquement ce modèle n'est pas correct car un parallélépipède N'EST PAS un rectangle donc l'héritage n'a pas de sens.

La classe object

Chaque classe du Python va **automatiquement hériter** d'une classe qui se nomme « **object** ». Cette classe comporte **une série de méthodes et d'attributs** qui seront ainsi automatiquement hérités par les classes enfants.

L'exemple le plus courant est sans doute celui de l'héritage de la méthode magique **__repr__** qui est la méthode utilisée lorsque l'on souhaite récupérer la **représentation de l'objet**. Il y a aussi la méthode **__str__** qui sera appelée lors d'un **cast en str**.
Ou encore l'utilisation des méthodes magiques **__getattr__** et/ou **__ setattr__** qui sont les deux méthodes utilisées par les objets pour setter ou getter les attributs qui les constituent (on peut les surcharger dans le but de réaliser des propriétés comme nous l'avons vu précédemment).

```python
class Personne:
    def __init__(self, nom, prenom, age):
        self.nom = nom
        self.prenom = prenom
        self.age = age

    def __repr__(self):
        return f"La personne s'appelle {self.prenom} {self.nom} et a {self.age} ans"

personne_1 = Personne("Dupont", "Jean", 40)
print(personne_1) # La personne s'appelle Jean Dupont et a 40 ans
```

Le polymorphisme

Le **polymorphisme** est une autre notion clé de la programmation orienté objet.
Le polymorphisme consiste en l'utilisation d'une **version différente d'une méthode**, on appelle ça aussi **surcharger/override** une méthode.
On peut réutiliser la méthode du parent avec **super().nom_méthode()**.

On voit ici que lors du parcours de la liste on appellera les **méthodes jouer()** des classes que l'on est en train de parcourir.
La méthode jouer de la classe Enfant va **remplacer** celle de la classe Personne.

```
liste_personnes = [
    Personne("Jean", "Dupont", 30),
    Enfant("Titou", "Enfant", 5, "Légo")
]
for personne in liste_personnes:
    personne.jouer()
```

```
class Personne:
    def __init__(self, nom, prenom, age):
        self.nom = nom
        self.prenom = prenom
        self.age = age

    def jouer(self):
        print("L'adulte n'a plus le temps de jouer")

class Enfant(Personne):
    def __init__(self, nom, prenom, age, jouet):
        super().__init__(nom, prenom, age)
        self.jouet = jouet

    def jouer(self):
        print(f"L'enfant joue avec {self.jouet}")
```

Duck typing

Le Duck Typing est un concept de Python provenant de l'expression anglophone « **If it walks like a duck, and it quacks like a duck, then it must be a duck** » (ou en français : Si ça marche comme un canard, que cela cancane comme un canard, alors cela doit être un canard).

Selon cette expression, il est **inutile de tester les types des classes** avant **d'appeler des méthodes qui leur sont accessibles**. Par exemple, il est possible d'utiliser la méthode **len()** donnant la taille de l'objet sur plusieurs types de variables, qu'elles soient ou non des conteneurs.

```
class CoinCoin:
    def __len__(self):
        return 42

ma_liste = [1, 4, 23]
mon_dic = {"nom":"toto",
    "prenom":"titi",
    "age": 12}
ma_chaine = "Hello World !"
mon_canard = CoinCoin()

print(len(ma_liste)) # 3
print(len(mon_dic)) # 3
print(len(ma_chaine)) # 13
print(len(mon_canard)) # 42
```

Visibilité en Python : Name Mangling

AVANCÉ

En Python, tout comme dans beaucoup de langages servant à réaliser de l'Orienté Objet, on a recours à ce qui s'appelle la **visibilité des attributs et des méthodes** dans le but de **sécuriser nos classes**. Malheureusement pour nous, il n'existe pas de mot-clés **private, protected, public**, etc… comme dans d'autres langages de programmation tels que le Java ou le C#.

A côté de cela, il existe en Python une **convention de nommage** permettant facilement aux développeurs Python de repérer si une variable ou une méthode est de type **publique, privée ou protégée**. Cette convention se base sur le « **name mangling** », une autre propriété du Python qui fait que lorsque l'on essaie d'accéder à une variable par la notation « **objet.__attribut** », l'interpréteur va en réalité transformer la chose en « **_nom-classe__nom-attribut** ». Ce processus est prévu dans le but de sécuriser les accès aux attributs de type privés, qui peuvent cependant encore être accédés via la syntaxe « **objet._nom-classe__nom-attribut** »

Ainsi, on a donc recours à ces conventions de nommage :

- Les attributs et méthodes publiques sont nommés « **ainsi** »
- Les attributs et méthodes protégés sont nommés « **_ainsi** »
- Les attributs et méthodes privés sont nommés « **__ainsi** »

Exercice – (15 min)

EXERCICE

1. Créer une classe **Personne**, contenant le *nom* de la personne, son **prénom**, son **numéro de téléphone** et son **email**. Une méthode **__str__** pour afficher les données de la personne.
2. Créer une classe **Travailleur** , qui hérite de la classe Personne et étend avec les attributs **nom d'entreprise**, **adresse entreprise** et **téléphone professionnel**. Une méthode **__str__** pour afficher les données et **qui réutilise celle de Personne**.
3. Créer une classe **Scientifique** qui hérite de la classe Travailleur et étend avec les attributs de type list **disciplines** (physique, chimie, mathématique, …) et **types du scientifique** (théorique, expérimental, informatique...) Une méthode __str__ pour afficher les données et **qui réutilise celle de Travailleur**.

L'héritage multiple

En Python, il est possible pour **une classe d'hériter de plusieurs classes**, ce qui peut conduire à des **situations délicates** que l'interpréteur solutionne en usant de ce que l'on appelle le « **MRO** » (Method Resolution Order). Il s'agit d'une liste contenant **l'ordre d'apparition des classes** servant pour l'héritage d'une classe. Pour accéder à cette MRO, il est possible d'avoir recours à la méthode **.mro()**, tout simplement.

Lors d'un **héritage multiple**, il est possible d'avoir comme situation un héritage dit « **en diamant** », car **une classe** est héritée par **deux classes** qui seront à leur tour héritées par **une même classe**. Dans ce genre d'héritage, il faudra bien faire attention à se servir du constructeur de la super-classe via l'utilisation du mot-clé **super()**, qui va en réalité chercher dans la MRO le constructeur dont on a besoin pour éviter les conflits. De plus, en cas de polymorphisme, c'est via la MRO que l'on saura laquelle des deux méthodes redéfinies sera utilisée.

```
class Toutou(Animal, Carnivore):
    """Un chien qui est à la fois un animal et un carnivore"""

toutou = Toutou()
toutou.se_nourrir()
print(toutou.point_de_vie)
```



```
class EtreVivant:
    def __init__(self):
        self.point_de_vie = 100

    def se_nourrir(self):
        self.point_de_vie += 1

class Animal(EtreVivant):
    def dormir(self):
        self.point_de_vie += 1

    def se_nourrir(self):
        self.point_de_vie += 5

class Carnivore(EtreVivant):
    def chasser(self):
        self.point_de_vie -= 1

    def se_nourrir(self):
        self.point_de_vie += 10
```

Exercice – (30 min)

EXERCICE

```python
class Address:
    def __init__(self, street, city):
        self.street = str(street)
        self.city = str(city)
    def show(self):
        print(self.street)
        print(self.city)
```

```python
class Person:
    def __init__(self, name, email):
        self.name = name
        self.email= email
    def show(self):
        print(self.name + ' - ' + self.email)
```

1. Créer la classe **Contact** qui **hérite à la fois de Address et Person**, cette classe doit implémenter la méthode **show()**
2. Créer une classe **Notebook** qui contient un **dictionnaire** qui associe les **noms des personnes** à un **objet Contact**. (Pas besoin d'héritage)
   a. Cette classe devra avoir une méthode **show()**
   b. Cette classe doit avoir une méthode **add(self, name, email, street, city)**
3. Tester le code suivant :

```python
notes = Notebook()
notes.add('Alice', '<alice@example.com>', 'Lv 24', 'Sthlm')
notes.add('Bob', '<bob@example.com>', 'Rtb 35', 'Sthlm')
notes.show()
```

Résultat attendu :

```
=== Alice ===
Alice - <alice@example.com>
Lv 24
Sthlm

=== Bob ===
Bob - <bob@example.com>
Rtb 35
Sthlm
```

TP – (60 min) – sur deux slides

TP

- Dans cet exercice on s'intéresse à créer **des classes pour gérer les vols d'une compagnie aérienne** qui organise des vols **entre des villes**.
- Plus précisément on s'intéressera aux plans de vol entre les différentes villes.
- Càd les vols disponibles ainsi que l'heure de départ.
- Créer une classe **Vol_direct** qui représentera un vol direct entre deux villes (pas d'escale dans une ville intermédiaire), on doit :
  - Définir le constructeur de cette classe qui a **quatre attributs** :
    - **Dep** et **arr** qui désigne respectivement la ville de départ et la ville d'arrivée
    - **jour** qui désigne le jour de la semaine (lundi, mardi, ...)
    - **heure** (**un entier entre 0 et 24** qui représente l'heure de départ)
  - Écrire une méthode **affiche()** qui affiche une chaine bien formatée de la forme :
    « **Ce vol part de Paris vers Marseille le lundi à 9 heure** »
- Créer une classe **Vols** qui représente tous les vols le long de la semaine en utilisant la classe Vol_direct. Pour ce faire on doit :
  - Définir le **constructeur** de cette classe avec **un seul attribut qui est une liste de vols**
  - Écrire une méthode **Liste_successeurs** qui retourne une **liste** contenant les **villes arrivées d'une ville de départ passée comme paramètre**
  - Écrire une méthode **Appartient** qui vérifie **si une ville appartient au plan du vol** que ce soit comme ville d'arrivée ou de départ
  - Écrire une méthode **Affiche** qui affiche tous les vols directs.

TP – (60 min) – Suite

TP

Ecrire un **programme principal** permettant de :

- Créer une **liste** nommée **lv** d'objets Vol_direct, on suppose avoir définie les 3 fonctions suivantes :
  - **Saisie_Jour** qui retourne un jour valide,
  - **Saisie_Heure** qui retourne une heure valide
  - **Saisie _Ville** qui retourne un nom de ville valide.
- Créer un **objet Vol** nommé **v** à partir de la liste déjà créée
- Afficher tous les vols
- Saisir une ville qui doit appartenir au plan du vol puis calculer et afficher la liste de ses successeurs

```
=== Liste des vols ===
Ce vol part de Paris vers Marseille le 17 à 4 heure
Ce vol part de Paris vers Lyon le 21 à 8 heure
Ce vol part de Marseille vers Lyon le 11 à 17 heure
Ce vol part de Paris vers Bruxelles le 4 à 20 heure

La ville Paris fait partie du plan de vol !
La ville Bruxelles fait partie du plan de vol !
La ville Bordeaux ne fait pas partie du plan de vol !

La liste des destinations à partir de Paris est : {'Lyon', 'Marseille', 'Bruxelles'}
```

Méta-classe

AVANCÉ

La méta-classe est un concept avancé en Python qui n'est que très rarement utilisé directement par les développeurs.
En Python, les classes sont elles-mêmes des objets qui **héritent de type**. Il est possible de spécifier le type dont doit hériter l'objet qui représente la classe. On parle alors de **méta-classe**. Une méta-classe est une classe qui décrit une classe. Cela signifie que tous les attributs et toutes les méthodes d'une méta-classe seront les attributs et les méthodes de la classe.

L'usage de la méta-classe permet de réaliser des implémentations qui ne sont pas possibles avec une simple classe. Par exemple, décorateur **@property** pour créer une propriété.

```python
class MetaClasseCompteur(type):
    """Une méta classe pour aider à compter les instances créées."""

    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
        cls._nb_instances = 0

    @property
    def nb_instances(cls):
        return cls._nb_instances

    def plus_une_instance(cls):
        cls._nb_instances += 1

class MaClasse(metaclass=MetaClasseCompteur):

    def __init__(self):
        MaClasse.plus_une_instance()

print(MaClasse.nb_instances) # 0
o1 = MaClasse()
o2 = MaClasse()
o3 = MaClasse()
print(MaClasse.nb_instances) # 3
```

Classe abstraite

En Python, le module **abc** permet de simuler le fonctionnement d'une **classe abstraite** (qui **ne doit pas être instanciable et est destinée uniquement à l'héritage**). Le nom de ce module est la contraction de « abstract base classes ».
Ce module fournit une classe **ABC** et une méta-classe appelée **ABCMeta** qui permettent de **transformer une classe Python en classe abstraite**.

Ce module fournit également le décorateur **@abstractmethod** qui permet de déclarer comme abstraite une méthode, une méthode statique, une méthode de classe ou une propriété. Cela signifie qu'il **n'est pas possible de créer une instance** d'une classe qui hérite d'une classe abstraite **tant que toutes les méthodes abstraites ne sont pas implémentées**.

```python
from abc import ABCMeta, abstractmethod


class Animal(metaclass=ABCMeta):

    @abstractmethod
    def crier(self):
        pass

class Chien(Animal):

    def crier(self):
        print("whouaf whouaf !")


# a = Animal() impossible car abstraite, à la place on se sert de Chien
c = Chien()
c.crier()
```

Exercice – (30 min)

AVANCÉ

EXERCICE

- Créer une classe **Interface** héritant de **ABC**.
- La classe interface implémentera la méthode magique __subclasshook__ qui permet de s'assurer que les méthodes dans la séquence __methods__ sont présentes dans la classe.
- Créer une classe **Container** qui impose l'existence de la méthode __contains__.
- Créer une classe **Sized** qui hérite l'existence de la méthode __len__.
- Créer une classe **SizedContainer** qui impose l'existence de la méthode __len__ et __contains__.
- Créer une classe **Iterable** qui impose l'existence de la méthode __iter__.

Qu’est-ce qu’une exception ?

Une **exception** est un **problème** qui apparait **lors de l’exécution du programme**. On parle d ’exception car c’est **un cas que le programme n’a pas pu gérer**, littéralement une exception en français.
Les exemples d’exceptions les plus courants sont les **exceptions de format**, les **exceptions de fichier introuvable** ou de **connexion impossible en base de données**.

Pour **réaliser un programme fonctionnel**, il faut **prendre en compte les erreurs** que pourraient causer les utilisateurs et faire en sorte **qu’elles soient non bloquantes**. En effet, lorsque l’on teste notre programme, on peut voir **les exceptions se lever**, et ainsi prendre conscience du problème. Ce n’est pas le cas pour les utilisateurs lambdas qui voient simplement le programme se stopper ou figer…

Pour éviter cela, on réalise donc **un bloc de récupération des exceptions** dans le but d’afficher des messages personnalisés ou de stocker les problèmes dans un fichier de log qui pourra par la suite être envoyé aux développeurs dans un soucis de maintenance du logiciel.

Attraper une exception

Pour attraper une exception, il faut faire appel à un bloc de type **try…except…else…finally.**

Ce bloc est donc constitué de quatre grandes parties, dont les deux dernières ne sont pas toujours utilisées ensemble :

- Le bloc **try** sert à contenir l'ensemble du **code que l'on souhaite exécuter** et **qui pourrait poser problème** lors de l'exécution
- **Le ou les** blocs **except** servent **à récupérer l'exception** dans le but de **la traiter** (ou non) de façon à ce **qu'elle ne bloque pas le fonctionnement du programme**. Il peut y avoir autant de blocs except que l'on veut, mais attention à bien mettre le bloc de récupération global après ceux concernant les exceptions spécifiques !
- Le bloc **else** sert **à exécuter du code** dans le cas où **aucune exception n'a été récoltée**
- Le bloc **finally** sert quant à lui à **exécuter du code à la fin de l'ensemble du bloc** try…except…else…finally dans le but d'être sur par exemple de fermer un fichier ou une connexion à une base de données peu importe s'il y a eu un souci ou non


```
try:
    age = int(input("Saisir votre Age : "))
except ValueError:
    print("Saisie invalide !")
except Exception:
    print("Une autre exception a été levée")
else:
    print("Saisie valide !")

try:
    age = int(input("Saisir votre Age : "))
except Exception as ex:
    print(ex)
    print("Saisie invalide !")
else:
    print("Saisie valide !")

try:
    age = int(input("Saisir votre Age : "))
except:
    print("Saisie invalide !")
else:
    print("Saisie valide !")
finally:
    print("après le try, avec ou sans exeception levées")
```

Exceptions personalisées

- Pour créer une exception nous-même, il nous suffit de créer une classe qui héritera d'**Exception** ou de **BaseException**
- Il est aussi possible de **lever une exception** avec le mot clé **raise**

```
def input_age():
    try:
        age = int(input("Saisir votre Age : "))
        if age<= 0 or age >=120:
            raise AgeInvalideException("Age invalide")
    except ValueError as ve:
        print(ve)
        print("Saisie invalide !")
        return -1
    except AgeInvalideException as aie:
        print(aie)
        return -1
    else:
        print("Age valide !")
        return age
```

```
class AgeInvalideException(Exception):
    pass

class MaSuperException(Exception):
    def __init__(self, *args):
        super().__init__("Une super exception a été levée", *args)
```

Exercice – (20 min)

EXERCICE

Via la gestion des exceptions et la levée d'exceptions personnalisées, vous devrez réaliser un programme en console qui **demandera à l'utilisateur un login** ne devant **comporter que des lettres** et **un mot de passe** ne **comportant que des chiffres**. Dans le cas contraire, vous devrez **lever une exception** qui **ne devra pas stopper** le fonctionnement du programme mais **s'afficher afin d'informer à l'utilisateur que ses informations sont incorrectes**

```
Veuilliez entrer un login SVP (celui-ci ne doit posséder que des lettres minuscules) : aaa
Veuilliez entrer un mot de passe SVP (celui-ci ne devra comporter que des chiffres) : dd
Le mot de passe ne dois posséder que des nombres !

Veuilliez entrer un login SVP (celui-ci ne doit posséder que des lettres minuscules) : Aa
Il ne dois y avoir que des minuscules dans le login !
Veuilliez entrer un mot de passe SVP (celui-ci ne devra comporter que des chiffres) : 47

Veuilliez entrer un login SVP (celui-ci ne doit posséder que des lettres minuscules) : aa
Veuilliez entrer un mot de passe SVP (celui-ci ne devra comporter que des chiffres) : 47
```

Les méthodes magiques

Nous avons déjà traité d'une méthode spéciale : la méthode **__init__()** qui désigne le constructeur. Pour des usages plus avancés, on peut définir les méthodes **__new__()** et **__del__()** pour réaliser les traitements de création et de suppression des objets.

On peut également déclarer la méthode **__repr__()** qui doit retourner la chaîne de caractères correspondant à la représentation de l'objet. Cette méthode est appelée directement par la fonction **repr()**. C'est également cette méthode qui est utilisée par la console Python pour afficher un objet et elle passe au dessus de __str__().

A côté de ça, la méthode magique **__hash__()** sert à renvoyer un hashage de notre objet. Elle est essentielle si l'on veut se servir de notre objet en tant que clé de dictionnaire par exemple (les clés de dictionnaire étant en réalité des hashing de variables). Ici, on se sert d'un tuple pour prendre l'ensemble des valeurs de la classe qui seront passées de la sorte dans la fonction de hashing.

```python
class Chien:
    def __init__(self, nom, age, race):
        self.nom = nom
        self.age = age
        self.race = race


    def crier(self):
        print("whouaf whouaf !")

    def __repr__(self):
        return f"{self.nom} a {self.age} ans et est de race : {self.race}"


mon_chien = Chien("Rex", 4, "Berger Allemand")
print(mon_chien)
```

```python
def __hash__(self):
    return hash((self.nom, self.age, self.race))
```

Les méthodes magiques conversions

Il est possible de réaliser des **cast** lorsque l'objet est passé en paramètre de certaines fonctions. La **conversion en valeur booléenne** est également utilisée lorsqu'un objet **doit être évalué comme expression booléenne** dans une structure **if** ou **while**.

Pour notre classe Chien, nous pourrions considérer qu'un chien est évalué à True si son âge, son nom et sa race sont vrai :

```
def __bool__(self):
    return len(self.nom) > 0 and len(self.race) > 0 and self.age != 0
```

```
if mon_chien:
    print("Mon Chien est vrai")
```

Méthode spéciale	fonction de conversion
`__str__(self)`	`str`
`__bytes__(self)`	`bytes`
`__bool__(self)`	`bool` ou expression booléenne
`__int__(self)`	`int`
`__float__(self)`	`float`
`__complex__(self)`	`complex`
`__dict__(self)`	`dict`

Les méthodes magiques conversions
Opérateurs unaires et arithmétiques

Si les objets doivent pouvoir être utilisés avec les opérateurs unaires **+val**, **-val** ou
s'ils peuvent être passés en paramètre de la fonction **abs()**, vous devez fournir respectivement une implémentation des méthodes **__pos__(self)**, **__neg__(self)**, **__abs__(self)**.

Si les objets doivent pouvoir être utilisés dans des **opérations arithmétiques**, alors vous pouvez fournir une implémentation pour les méthodes suivantes :

Méthode spéciale	opérateur ou fonction
`__add__(self, o)`	`+`
`__sub__(self, o)`	`-`
`__mul__(self, o)`	`*`
`__matmul__(self, o)`	`@`
`__truediv__(self, o)`	`/`
`__floordiv__(self, o)`	`//`
`__mod__(self, o)`	`%`
`__divmod__(self, o)`	`divmod()`
`__pow__(self, o, modulo)`	`**` ou `pow()`

Les méthodes magiques conversions
Opérateurs unaires et arithmétiques

```python
def __mul__(self, other):
    if isinstance(other, Chien):
        if random.randint(0, 1):
            return Chien("Nouveau", 0, self.race)
        else:
            return Chien("Nouveau", 0, other.race)
```

```python
class Vecteur:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __neg__(self):
        return Vecteur(-self.x, -self.y)
```

```python
mon_chien = Chien("Rex", 4, "Berger Allemand")
mon_chien_bis = Chien("Bernie", 7, "Labrador")
mon_nouveau_chien = mon_chien_bis * mon_chien
print(mon_chien)  # Rex a 4 ans et est de race : Berger Allemand
print(mon_chien_bis)  # Bernie a 7 ans et est de race : Labrador
print(mon_nouveau_chien)  # Nouveau a 0 ans et est de race : Labrador
```

Les méthodes magiques comparaisons

Par défaut, l'opérateur d'égalité **==** permet de comparer l'unicité en mémoire des objets. Ainsi les deux chiens ci-dessous ne sont pas égaux :

```
mon_chien = Chien("Rex", 4, "Berger Allemand")
mon_chien_bis = Chien("Rex", 4, "Berger Allemand")
print(mon_chien_bis == mon_chien)  # False
```

En effet, nous créons deux objets distincts que nous affectons respectivement à la variable **mon_chien** et à la variable **mon_chien_bis**.
Mais il serait intéressant de considérer que deux chiens sont égaux s'ils ont les mêmes valeurs pour leurs champs. Nous pouvons modifier ce comportement par défaut en fournissant notre propre méthode d'égalité :

```
def __eq__(self, other):
    if isinstance(other, Chien):
        return self.race == other.race and self.age == other.age and self.nom == other.nom
```

```
mon_chien = Chien("Rex", 4, "Berger Allemand")
mon_chien_bis = Chien("Rex", 4, "Berger Allemand")
print(mon_chien_bis == mon_chien)  # True
```

Il existe aussi les méthodes magiques de comparaison **lt (<), le (<=), gt (>), ge (>=) et ne (!=).**

Les méthodes magiques conteneurs

Si vos objets doivent se comporter comme un **conteneur** (c'est-à-dire comme une liste ou un dictionnaire), vous pouvez fournir l'implémentation des méthodes Dunder suivantes :

Méthode spéciale	Cas d'utilisation
`__len__(self)`	utilisation de la méthode `len()`
`__getitem__(self, key)`	o[key]
`__setitem__(self, key, value)`	o[key] = value
`__delitem__(self, key)`	del o[key]
`__contains__(self, key)`	key in o

Les méthodes magiques conteneurs

```python
class Vecteur:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __len__(self):
        return 2

    def __getitem__(self, k):
        if k == 'x' or k == 0:
            return self.x
        if k == 'y' or k == 1:
            return self.y
        raise KeyError(k)

    def __setitem__(self, k, v):
        if not isinstance(v, (int, float)):
            raise TypeError
        if k == 'x' or k == 0:
            self.x = v
        elif k == 'y' or k == 1:
            self.y = v
        else:
            raise KeyError(k)
```

```python
v = Vecteur(2, 5)
print(len(v))  # 2
print(v['x'])  # 2
print(v[0])  # 2
print(v['y'])  # 5
print(v[1])  # 5

v[0] = -2
v[1] = -5
print(v)  # Vecteur(-2,-5)
```

Les méthodes magiques classes abstraites

Beaucoup de méthodes abstraites n'ont de sens que lorsqu'elles sont implémentées ensemble par la même classe. Par exemple les méthodes **__len__(self)** ou **__getitem__(self, key)** permettent de définir une séquence puisqu'il est possible de connaître la taille et l'élément associé à une clé.

Le module **container.abc** fournit des classes abstraites qui définissent différents contrats. Il existe par exemple la classe abstraite **Sequence** qui déclare les deux méthodes de manière abstraite.

```
from collections.abc import Sequence
s = Sequence()

# Traceback (most recent call last):
#   File "C:\Users\gharr\source\repos\training_python\demoFormation\examples.py", line 3, in <module>
#     s = Sequence()
# TypeError: Can't instantiate abstract class Sequence with abstract methods __getitem__, __len__
```

Les méthodes magiques classes abstraites

Toutes les classes du module **colletions.abc** sont des classes abstraites qui sont là pour guider le développeur qui voudrait créer sa propre classe et qui souhaiterait que **les objets de cette classe se comportent suivant un contrat**. En héritage d'une des classes du module colletions.abc, cela permet au développeur de vérifier que son implémentation est conforme au contrat.

TP – (60 min)

TP

1- Créer une classe **Intervalle** possédant une méthode **__init__** permettant d'initialiser **une borne inférieure** et **une borne supérieure** pour un objet de type Intervalle.

Vérifier que les bornes sont numériques, positives, non nulles et placées dans le bon ordre, sinon générer une exception de type « **IntervalError** » affichant le message d'erreur **« Erreur : Bornes invalides ! ».**

Le type « IntervalError » est une **Exception à définir.**

2 - En effet, avec les contrôles définis dans la méthode __init__, on ne peut plus créer un intervalle mal formé. Toutefois, il est toujours possible à un programmeur d'écrire directement a.borne_sup = -2, ce qui mettra -2 dans la borne supérieure de l'intervalle Modifier la portée des attributs borne_inf et borne_sup afin qu'ils ne soient visibles que depuis les méthodes de la classe, mais pas de l'extérieur. (private)

3. Pour modifier une valeur de l'intervalle, écrire dans la classe Intervalle une méthode **modif_borne_sup** qui permettra de protéger la borne supérieure en ne pouvant y écrire que des nombres supérieurs à la borne inférieure.

4. Ajoutez une méthode **modif_borne_inf** à la classe Intervalle. Faites attention à ce qu'**une valeur négative ne puisse pas être enregistrée**.

5. Écrivez deux méthodes d'accès **lire_inf(self)** et un **lire_sup(self)** qui retourneront les valeurs des bornes.

6. Écrire une méthode spéciale **__str__(self)** permettant de retourner **une chaine indiquant les valeurs des deux bornes de l'intervalle**.

7. Écrire une méthode spéciale **__contains__(self, val)** qui teste si une valeur val **appartient ou non à l'intervalle** (utilisé par l'opérateur **in**).

8. Écrire une méthode spéciale **__add__(self, autre)** qui retourne **un nouvel Intervalle addition des deux intervalles**. Exemple : [2,5] + [3,4] = [5,9].

9. Écrire une méthode spéciale **__sub__(self, autre)** qui retourne **un nouvel Intervalle soustraction des deux intervalles**.

10. Écrire une méthode spéciale **__mul__(self, autre)** qui retourne **un nouvel Intervalle multiplication des deux intervalles**. Exemple : [2,5] * [3,4] = [6,20]

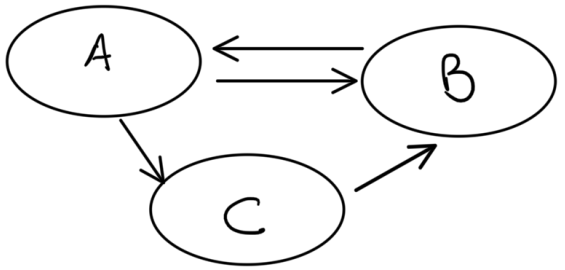
11. Écrire une méthode spéciale **__and__(self, autre)** (&) qui retourne **l'intersection des deux intervalles** et « **None** » si leur **intersection est vide**. Exemple: [2,5]∩[3,6]=[3,5]

TP Global

AVANCÉ

TP

- Afin de mettre en pratique les compétences acquises lors du module « Programmation orienté objet », nous souhaitons modéliser en POO **un graphe** avec des **nœuds** et des **bords** comme le diagramme ci-dessous.



- Créez les classes nécessaires.
- Nous souhaitons réaliser une **application de planification de voyage**.
- Cette application modélisera un ensemble de **villes** ainsi que **les moyens de transport** possibles entre celle-ci.
- En utilisant **le diagramme ci-dessous**, ainsi que les classes créées dans la question 1, créez l'ensemble de classes nécessaires pour notre application.
- En utilisant la fonction short_path fourni dans le module suivant : https://github.com/utopios/practice_python/blob/main/short_path.py
  - Trouvez le chemin **le plus rapide entre Lille et Lyon**.
  - Trouvez le chemin **le moins coutant entre Lille et Lyon**.