

Formation Python Module 1

Ihab ABADI / UTOPIOS

Sommaire

- Présentation de Python et ses versions
- Présentation de l'environnement de développement python avec visual studio code ou Pycharm.
- Structure d'un projet.
- Syntaxe de base de Python.
 - Variables
 - Structures conditionnelles.
 - Structures itératives.
- Fonctions
- Notion de module et package
- Les conteneurs.
- Listes
- Tuples
- Sets
- Dictionnaires.
- Mutable et Muable
- Les décorateurs

Sommaire

- La POO en python les bases.
 - Notion de classe et instances.
 - Attributs
 - Méthodes
 - Constructeurs.
 - Les propriétés avec les annotations et sans.
 - Attributs et méthodes de classes
- Héritage et polymorphisme.
 - L'héritage
 - Constructeur et héritage
 - Héritage et mutualisation de code
 - La classe object
 - Le polymorphisme
 - Visibilité des attributs et des méthode classe enfant
 - Héritage multiple

Sommaire

- Méta-classe
- Classes et méthodes abstraites
- Gestion des exceptions.
- Les méthodes magiques.
 - Méthodes magiques élémentaires
 - Méthodes magiques pour les conversions
 - Méthodes magiques pour les opérateurs unaires
 - Méthodes magiques pour les opérations arithmétiques
 - Méthodes magiques pour la comparaison
 - Méthodes magiques pour les conteneurs
 - Méthodes magiques et classes abstraites
 - Autres méthodes magiques

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 3.10 a été publiée en Octobre 2021.
- 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 tâches
 - Analyses de données
 - Calculs numériques
 - Développement web
 - ...

Environnement de développement.

- Python est déjà installé sur Mac et Linux, pour l'installation sur windows:
- <https://www.python.org/ftp/python/3.11.0/python-3.11.0b4-amd64.exe>
- Pip est le gestionnaire de paquets de Python et qui est systématiquement présent depuis la version 3.4.
- Pycharm ou vscode

Environnement de développement.

- Un script python est un fichier avec l'extension .py.
- Pour démarrer un script python on utilise la commande python ou python3 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 instruction python en un langage compréhensible par l'ordinateur.
- Il peut être utiliser pour exécuter une instruction
- Pour ouvrir l'interpréteur il suffit de taper dans un terminal (powershell, bash,...) python ou python3, Résultat triple chevrons.
- Pour quitter l'interpréteur, Ctrl+d ou la fonction exit()

```
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.
>>> █
```

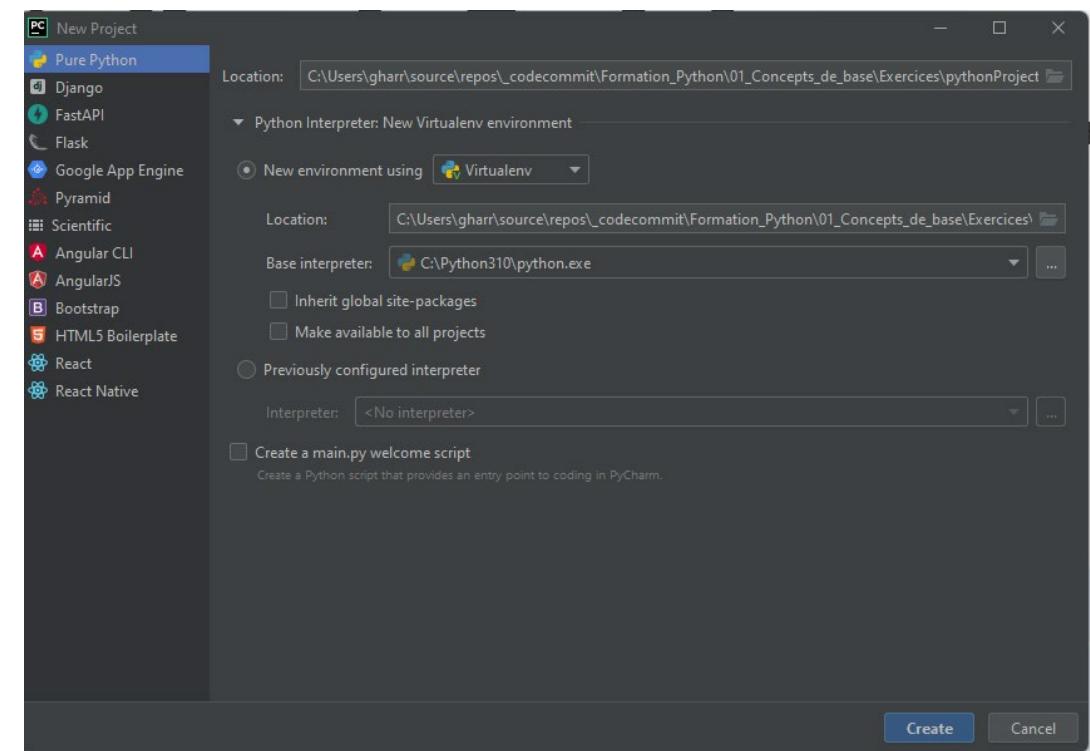
Environnement de développement.

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 feront pas usage dans ce cours. Appuyez simplement sur **Create** dans le but de valider la création du projet Python.



Environnement de développement.

- Pycharm génère un environnement virtuel stocké dans venv (packages et modules installés).
- Pycharm permet d'ajouter des scripts python dans notre dossier (click droit, add python file).
- Pycharm permet l'exécution d'un script par un click droit – run ou (CTRL + SHIFT + F10).

Les variables

Dans tout langage de programmation, nous avons besoin de variables dans le but de stocker des informations dans la mémoire vive de l'ordinateur (cette dernière étant régulièrement consultée pour faire tourner un programme et exécuter des instructions plus ou moins complexe par nos scripts).

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

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

```
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
```

- Les chaînes de caractères « **strings** » qui permettent de stocker du texte.

```
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"""


```

Récupérer et afficher des valeurs

Evidemment, dans tout programme, il faut qu'il y ait un dialogue possible entre l'utilisateur et l'ordinateur. Pour réaliser ce dialogue, dans le cadre d'un programme de type console, on a recours à ce qui s'appelle des affichages et des récupérations de valeurs. Un affichage correspond simplement en la capacité de la console à présenter à l'utilisateur une chaîne de caractère plus ou moins formatée présentant des variables. A contrario, une récupération est la capacité de la console à interpréter les informations envoyées par l'utilisateur pour les transformer en un type de variable et la stocker dans la mémoire vive de l'ordinateur.

```
# Une récupération passe par l'utilisation de la fonction input()
# qui récupèrera toujours une chaîne de caractères
ma_recuperation = input("Veuillez 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 une 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, qui sont pour l'affichage le **formatage des strings**, et pour la récupération le **casting des variables**.

Récupérer et afficher des valeurs

Lorsque l'on veut présenter de façon harmonieuse les informations stockées en mémoire à l'utilisateur, il faut souvent se servir de ce que l'on appelle le formatage d'une chaîne de caractère. Pour ce faire, il existe plusieurs méthodes. La plus ancienne est la méthode **.format()** qui est liée au type string. Pour s'en servir, il n'y a rien de plus simple, il suffit de placer des marqueurs dans une chaîne de caractère (deux accolades) qui serviront d'emplacement prévus pour l'affichage des variables paramètres de la méthode format.

Ces marqueurs peuvent posséder des numéros (qui commencent par 0 généralement) pour indiquer quelle variable paramètre de la méthode de formatage sera affichée ici. Ils peuvent également contenir des valeurs d'espacement et des valeurs de formatage supplémentaire pour afficher par exemple un certain nombre de chiffres après la virgule, ou la valeur selon un format spécifique :

```
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
```

Enfin, depuis les versions récentes de Python, il existe ce qui s'appelle les f-strings. Ces chaînes de caractère permettent de formater directement le texte en y incluant entre accolades les variables que l'on souhaite y placer :

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

Casting des variables

Lorsque l'on utilise un langage de programmation, il est fréquent que l'on ait besoin de s'assurer du typage de nos variables. En plus de cela, Python est un langage où le type des variables est donné de façon dynamique, ce qui complique encore plus la chose.

Pour éviter les problèmes de typages, on se sert de ce qui s'appelle le « **casting** ». Pour réaliser un casting, il n'y a rien de plus simple. Il suffit d'ajouter le type des variables que l'on souhaite puis de passer entre parenthèse la variable que l'on souhaite caster.

Par exemple, on peut s'assurer de la récupération d'un nombre en castant une variable de type **string** en **int** de la sorte :

```
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("Veuillez 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 problème générant ce que l'on appelle des **exceptions** ! Nous verrons comment traiter les exceptions plus tard.

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 ».

Exercice – (5 min)

EXERCICE

- Ecrire un programme, qui à partir de la saisie d'un nom, d'une race et d'un âge, affiche le message suivant :

Le chien « nom » de race « race » a « âge » ans

Les opérateurs

Lorsque l'on utilise un langage de programmation dans le but de réaliser un logiciel, on a fréquemment recours aux opérateurs. Il en existe de plusieurs types, parmi lesquels les opérateurs unaires, binaires, arithmétiques, logiques, d'affectation, etc...

Par exemple, pour donner une valeur à une variable, on utilise l'opérateur d'affectation, qui n'est pas celui d'égalité (ce dernier se notant « == »), comme ci-après :

```
mon_nombre = 4 # On affecte à la variable mon_nombre la valeur 4
print(mon_nombre) # 4
```

Il est ensuite possible d'utiliser les opérateurs arithmétiques sur les variables pour les manipuler. Comme opérateurs arithmétiques, il existe :

```
mon_resultat = 4 + 4 # on affecte à la variable mon_resultat 4 + 4, soit 8
print(mon_resultat) # 8
mon_resultat = mon_resultat - 4 # on 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 conserve 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 l'autre, ce qui s'appelle la **concaténation**.

Exercice – (5 min)

EXERCICE

- Écrire un programme qui, à partir de la saisie d'une longueur, calcule le périmètre et l'aire d'un carré

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 logiques

Après la possibilité de manipuler de façon mathématiques les variables, il faut savoir qu'il est également possible de les comparer via des opérateurs de comparaisons qui sont :

- « > » : Supérieur à
- « < » : Inférieur à
- « >= » : Supérieur ou égal à
- « <= » : Inférieur ou égal à
- « == » : Egal à
- « != » : Différent de

Les opérateurs logiques

Ces opérateurs vont renvoyer un type de variable qui s'appelle le type « booléen ». Ce type ne peut prendre que deux valeurs : **True** ou **False** (Vrai ou Faux). C'est ce type de variables qui est la clé de la plupart des programmes informatiques et qui permet le déplacement de l'utilisateur dans les multiples choix de fonctionnalités d'un logiciel.

Les valeurs de type booléennes peuvent également être manipulées via les tables de vérités et les opérateurs logiques **AND**, **OR** et **XOR** (ET, OU, OU EXCLUSIF), dans le but d'obtenir ce genre de tables :

```
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, afficher true si majeur et false si mineur (sans structure conditionnelle).

Les structures conditionnelles

Lorsque l'on veut permettre à notre programme de prendre des chemins différents en fonction de saisies utilisateurs ou de vérifications des valeurs de variables durant l'exécution, on a recours aux structures conditionnelles. En Python, les structures conditionnelles sont assez simple à appréhender.

Pour réaliser une structure conditionnelle, on se sert des mots clés « **IF** », « **ELSE** » et « **ELIF** » :

- « **IF** » sert à initialiser la structure de contrôle, par exemple pour vérifier qu'une personne est majeur avant de lui permettre d'accéder à une série d'instructions logicielles
- « **ELIF** » permet d'ajouter une autre structure IF dans le cas où la précédente n'aura pas été respectée. On peut enchaîner ainsi autant de structures ELIF que l'on souhaite
- « **ELSE** » est la dernière partie d'une structure de contrôle, et permet d'effectuer une série d'instruction dans le cas où aucune des structures précédente n'a été respectée.

```
mon_age = int(input("Veuillez 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")
```

Exercice – (10 min)

EXERCICE

- Ecrire un programme qui prend en entrée une température t 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

Exercice – (15 min)

EXERCICE

- Écrire un programme qui permet de tester si un profil est valable pour une candidature ou non selon trois critères : l'âge, le salaire demandé et le nombre d'année d'expérience.
- De sorte que les conditions suivantes soient vérifiées
 - L'âge minimum pour le poste est 30 ans
 - Le salaire maximum possible est 40000 euros
 - Le nombre d'années d'expérience min est de 5 ans.
- On affichera différents message pour chaque condition non respectée.

Les structures itératives

Dans le cadre des structures d'itération, il existe en Python deux types de façon de faire des boucles :

- La boucle « **while** » (Tant que...) qui sera exécutée du temps 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**

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

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

for item in range(0, 10):
    print("Je suis l'itération n°:", item + 1)
```

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 alors à l'itération suivante en se replaçant au tout début de la boucle. Pour ce faire, on ignore alors tout ce qui aurait du se dérouler après le mot-clé.
- « **break** » : On sort immédiatement de la boucle sans effectuer les instruction se trouvant après le mot-clé et à l'intérieur de la boucle.
- « **pass** » : Ce mot-clé sert simplement de façon syntaxique à fermer une boucle. Il est également utilisé dans le cadre des fonction et n'est pas obligatoire

```
while True:  
    value = input("Dites STOP pour arreter : ")  
    if value == "STOP":  
        break  
    elif value.upper() == "STOP":  
        print("En UPPERCASE ! ")  
        continue  
    else:  
        pass
```

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 saisi par l'utilisateur.
- Gérer l'affichage en ajoutant des espaces après chaque table.
- Le résultat après exécution est donné par l'image suivant

| 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 en langage python qui permet d'afficher un triangle isocèle formé d'étoile(*) .
 - La hauteur du triangle (c 'est à dire le nombre des lignes) sera fournie en donnée, comme dans l exemple ci dessous.
 - Le nombre des caractères étoiles(*) sur la base sera le double de hauteur du triangle

saisir la hauteur du triangle : 10

Exercice – (15 min)

EXERCICE

- On dispose d'une feuille de papier d'épaisseur 0.1mm.
- Combien de fois doit-on la plier au minimum pour que l'épaisseur dépasse une de 400m
- Écrire un programme en Python pour résoudre ce problème

Exercice – (20 min)

EXERCICE

Réalisez un programme permettant à l'utilisateur d'entrer comme données une population initiale, un taux d'accroissement ainsi qu'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 éventuellement quelque chose.
- 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 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.
- Une particularité des fonctions en Python est que vous n'êtes pas obligé de préciser le type des arguments que vous lui passez, dès lors que les opérations que vous effectuez avec ces arguments 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

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

```
res = carre(2)  
print(res)
```

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

Exercice – (20 min)

EXERCICE

- Un programme principal saisit une chaîne d'ADN valide et une séquence d'ADN valide (valide signifie qu'elles ne sont pas vides et sont formées exclusivement d'une combinaison arbitraire de "a", "t", "g" ou "c").
- Écrire une fonction valide qui renvoie vrai si la saisie est valide, faux sinon.
- Écrire une fonction saisie qui effectue une saisie valide et renvoie la valeur saisie sous forme d'une chaîne de caractères.
- Écrire une fonction proportion qui reçoit deux arguments, la chaîne et la séquence et qui retourne la proportion de séquence dans la chaîne (c'est-à-dire son nombre d'occurrences).
- Le programme principal appelle la fonction saisie pour la chaîne et pour la séquence et affiche le résultat.
- Exemple

chaîne : attgcaatgggtacatg

séquence : ca

Il y a 10.53 % de "ca" dans votre chaîne.

Modules

- Un module est un ensemble de code encapsulés dans un script et qui peut être utilisé par d'autre script.
- 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 help('modules')
- Python nous donne la possibilité de créer nos propres modules.

Modules

- Un module est un script contenant des variables et des fonctions dont certaines seront utilisables par d'autre script.
- Exemple circle.py

```
from math import pi

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

Modules

- L'importation permet à un script d'utiliser le code d'un module
- Syntaxes d'importation
 - 1- import mon_module => importer la totalité du module
 - 2- from mon_module import mon_membre => importer uniquement mon_membre
 - 3- from mon_module import * => importer la totalité des membres du module
- Syntaxe d'accès à un membre d'un module importé
 - mon_module.mon_membre si import avec 1
 - Mon_membre si import avec 2 et 3
- Exemple

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

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

Exercice – (10 min)

EXERCICE

- Ecrire un module avec les fonctions de l'exercice précédent :
- Un programme principal saisit une chaîne d'ADN valide et une séquence d'ADN valide (valide signifie qu'elles ne sont pas vides et sont formées exclusivement d'une combinaison arbitraire de "a", "t", "g" ou "c").
- Écrire une fonction valide qui renvoie vrai si la saisie est valide, faux sinon.
- Écrire une fonction saisie qui effectue une saisie valide et renvoie la valeur saisie sous forme d'une chaîne de caractères.
- Écrire une fonction proportion qui reçoit deux arguments, la chaîne et la séquence et qui retourne la proportion de séquence dans la chaîne (c'est-à-dire son nombre d'occurrences).
- Le programme principal appelle la fonction saisie pour la chaîne et pour la séquence et affiche le résultat.
- Exemple
 - chaîne : attgcaatgggtacatg
 - séquence : ca
 - Il y a 10.53 % de "ca" dans votre chaîne.

Manipuler les fichiers

En Python, il est tout à fait possible de manipuler les fichiers via l'utilisation de la fonction `open()`. Cette fonction va prendre en premier paramètre un chemin de fichier et en second paramètre un mode d'ouverture.

Ce mode d'ouverture peut être:

- r : Lecture
- w : Ecriture
- a : Ajout
- b : Ouverture en mode binaire
- x : Ouverture en mode exclusif
- t : Ouverture sous format texte (par défaut)
- + : Lecture et Ecriture

Il est également important de penser à fermer notre fichier en fin de son utilisation sous peine d'avoir de problèmes d'accès au niveau de notre ordinateur. Pour ce faire, il convient d'utiliser la méthode `.close()` sur notre variable résultat de la méthode `open()`

Ecrire et Lire dans un fichier

Une fois notre fichier ouvert, il est tout à fait possible de le manipuler pour y lire ou y ajouter des informations. Pour ce faire, il existe en Python plusieurs méthodes disponibles sur un fichier telles que :

- `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
- `readlines()` : Pour obtenir une liste des lignes du fichier. Cette méthode prendra en compte les caractères spéciaux tels que le caractère de retour à la ligne
- `write()` : Permet d'ajouter une ligne au fichier
- `writelines()` : Permet d'ajouter une série de lignes au fichier

Exercice – (10 min)

EXERCICE

- Ecrire un programme permettant à un utilisateur de sauvegarder dans un fichier un texte secret (le fichier devra être créé en cas d'absence de ce dernier)
- L'utilisateur pourra ensuite soit voir le secret, soit modifier le secret, soit quitter le programme (ce qui sauvegardera le secret actuellement dans le programme dans le fichier texte)

Modules natifs

- Comme vu avant Python vient avec un ensemble de module de base
- Quelque exemple 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 (ici ";")
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")
ecrivainCSV = csv.writer(fichier,delimiter=";")
ecrivainCSV.writerow(["Nom","Prénom","Téléphone"]) # On écrit la ligne d'en-tête avec le titre des colonnes
ecrivainCSV.writerow(["Dubois","Marie","0198546372"])
ecrivainCSV.writerow(["Duval","Julien \\" , "0399741052"])
ecrivainCSV.writerow(["Jacquet","Bernard","0200749685"])
ecrivainCSV.writerow(["Martin","Julie;Clara","0399731590"])
fichier.close()
```

Exercice – (10 min)

EXERCICE

- Ecrire un script qui demande les informations d'un produit: titre prix et stock et les ajoutes dans un fichier produits.csv.

Packages

- Un package, par essence, est comme un répertoire contenant des sous-packages et des modules. Bien que nous puissions créer nos propres packages, nous pouvons également en utiliser un du Python Package Index (PyPI) à utiliser pour nos projets.
- Pour importer un package, nous utilisons la même syntaxe d'import de module
- Un paquet 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 une `AttributeError`.

Packages Vs module

- Un module est un fichier contenant du code Python. Un package, cependant, est comme un répertoire qui contient des sous-packages et des modules.
- Un paquet doit contenir le fichier `__init__.py`. Cela ne s'applique pas aux modules.
- Pour tout importer depuis un module, nous utilisons le joker *. Mais cela ne fonctionne pas avec les packages.

Qu'est-ce qu'un conteneur ?

Un conteneur est un objet permettant de stocker d'autres objets. A l'exemple d'une liste, il est ainsi possible de stocker plusieurs variables au sein de la même variable de type conteneur. Les conteneurs sont, à l'instar des autres variables du Python, dynamiques (c'est-à-dire qu'ils peuvent contenir plusieurs types de données, comme un Integer, une String voire même des classes telles que Chien. Un conteneur peut également contenir un autre conteneur (du même type ou d'un type différent)

Il est généralement possible au sein d'un conteneur d'accéder aux valeurs par utilisation d'un index ou d'une clé, tel que :

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

mon_dict = {'key1': 123, 'key2': '456', 'key3': [7, 8, 9]}
print(mon_dict) # {'key1': 123, 'key2': '456', 'key3': [7, 8, 9]}

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

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

mon_dict = {'key1': 123, 'key2': '456', 'key3': [7, 8, 9]}
print(mon_dict['key2']) # 456
```

Les Listes

Une liste est un type de conteneur très souvent utilisé, car il permet facilement d'assembler, de trier, de retirer des valeurs d'un conteneur via l'utilisation de ses méthodes propres. Une peut être déclarée vide ou contenant déjà des valeurs de la sorte :

Les méthodes les plus utilisées des listes sont les méthodes **.sort()**, **.append()**, **.extend()**, **.pop()** et **.remove()**,

comme dans l'exemple ci-après :

- **sort()** trie les éléments dans la liste
- **append()** permet d'ajouter un élément à la fin de la liste
- **extend()** permet d'ajouter une liste à la fin de la liste
- **pop()** permet de retirer un élément de la liste à l'index donné
- **remove()** permet de retirer un élément de la liste

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

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

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 – 15 min

EXERCICE

- En vous servant d'une fonction qui aura pour rôle de retourner l'ensemble des nombres premiers entre 1 et une valeur maximale comprise sous la forme de liste
- Vous réaliserez un programme permettant à l'utilisateur d'entrer une valeur et de se voir offrir la liste des nombres premiers alimentée par la fonction précédente. L'affichage de ces nombres devra se faire de sorte à respecter l'exemple ci-dessous :

```
Quel est la valeur du nombre maximal ? 20
1
2
3
5
7
11
13
17
19
```

Exercice – 20 min

EXERCICE

- Via l'utilisation d'une variable de type conteneur (liste), vous devrez réaliser un logiciel permettant à l'utilisateur d'entrer une série de notes (dont le nombre possible à entrer sera soit défini au début du lancement du logiciel, soit 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 meilleure des notes, la moins bonne ainsi que la moyenne de l'ensemble.

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

Les Tuples

- Permet de regrouper des données connexes
- Les données sont non-modifiables et repérées par leur indices
- Syntaxe de définition:
- nom_tuple = () ou nom_tuple = tuple()
- Exemple : mon_tuple= (1,2,3) ou mon_tuple=tuple(('a', 'b'))
- Accès aux éléments comme une liste
- Parcourir comme une liste
- Vérifier la présence d'un élément comme une liste
- Les opérations sur un tuple:
- Len => retourne le nombre d'élément d'un tuple
- Del => supprime complètement le tuple
- Count => retourne le nombre d'occurrence d'un élément dans la tuple
- Index => retourne l'index de la première occurrence

Exercice – 15 min

EXERCICE

- En vous servant de l'unpacking et du packing de tuples, vous réaliserez une fonction permettant de retourner à la fois le résultat des 4 opération arithmétiques de base sur deux variables de type nombre (addition, soustraction, multiplication et division) :

```
Veuillez entrer un premier nombre : 12
Veuillez entrer un second nombre : 18
L'addition de vos deux nombres donne 30.00
La soustraction de vos deux nombres donne -6.00
La multiplication de vos deux nombres donne 216.00
La division de vos deux nombres donne 0.67
```

Les sets

Un set est un ensemble immutable d'éléments uniques et ordonnés. Lors de l'ajout ou du retrait d'un élément d'un set, le conteneur se voit ainsi automatiquement réordonné pour contenir une succession d'éléments dont on se sert généralement pour l'itération et l'affichage.

Lorsque l'on constitue un set à partir d'une liste, on obtient ainsi une série d'élément non répétés qui ne peuvent plus être modifiés via l'index (les sets ne permettant pas la modification des éléments via cette méthode).

Les sets contiennent des méthodes semblables aux listes mais n'en disposent pas de beaucoup.

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

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

Exercice – 20 min

EXERCICE

- Via l'utilisation d'une variable de type conteneur (set), vous devrez réaliser une application permettant à l'utilisateur de stocker des noms de familles, de les afficher, de les supprimer ou de les éditer. 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 valeurs pour contenir des éléments. Il est ainsi possible d'accéder aux éléments qui le constituent via l'utilisation d'une notation entre crochets demandant la clé voulue.

Pour récupérer les valeurs d'un dictionnaire, on peut se servir de la méthode **.values()**. De même, pour les clés, il y a la méthode **.keys()**. Enfin, lorsque l'on veut avoir une association de clés et de valeurs, on peut se servir de la méthode **.items()** :

Via l'utilisation plus avancée des tuples, qui est le type retourné par la méthode **.items()**, il est possible d'afficher les informations du dictionnaire plus facilement :

```
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
```

```
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'})])
```

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



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

Exercice – (10 min)

EXERCICE

- Via l'utilisation de variables de type conteneur (dictionnaire), vous réaliserez deux ensemble de lettres de l'alphabet (en minuscules puis en majuscules) ainsi que de leur correspondance dans la table ASCII. Chaque dictionnaire devra retourner (après demande d'une clé correspondant au caractère demandé) sa valeur en ASCII.
- Vous fournirez ensuite à l'utilisateur l'ensemble de la table ASCII de ces lettres sous le format suivant.

| ==> ASCII lettres minuscules <== | | |
|----------------------------------|---|-----|
| a | - | 97 |
| b | - | 98 |
| c | - | 99 |
| d | - | 100 |
| e | - | 101 |
| f | - | 102 |
| g | - | 103 |
| h | - | 104 |
| i | - | 105 |

| ==> ASCII lettres minuscules <== | | |
|----------------------------------|---|----|
| A | - | 65 |
| B | - | 66 |
| C | - | 67 |
| D | - | 68 |
| E | - | 69 |
| F | - | 70 |
| G | - | 71 |
| H | - | 72 |
| I | - | 73 |
| J | - | 74 |
| K | - | 75 |

Exercice – (20 min)

EXERCICE

- Via l'utilisation d'une variable de type conteneur (dictionnaire), vous devrez réaliser un logiciel servant à un utilisateur pour entrer et stocker une série d'adresses (celles-ci devront avoir un numéro de voie, un complément de voie, un intitulé de voie, une commune et un code postal). Pour ce faire, vous utiliserez des clés de type string qui représenteront les différentes données stockées dans le dictionnaire.
- Le logiciel devra offrir ces données de manière non persistante et permettre l'ajout, l'édition, la suppression et la visualisation des données par l'utilisateur.

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

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

Mutable / Muable

- La mutabilité est la capacité d'une variable à être modifiée. Il ne faut pas la confondre avec la réassiguation, qui stocke simplement une autre variable du même type (ou d'un type différent) à un autre emplacement mémoire. Les types non-mutables 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-dessous

```
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. Les objets de type **list**, **dict** et **set** sont non-hashables. A contrario, tous les autres types de valeur le sont. Par exemple, il est possible de hasher un **int**, une **string** et une **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
```

On remarque une certaine corrélation entre types mutables et hashables. En effet, il est plus facile d'assurer l'invariabilité du condensat quand l'objet est lui-même immuable. Pour les objets mutables, le hash n'est possible que si la modification n'altère pas l'égalité entre deux objets, c'est-à-dire que deux objets égaux le resteront même si l'un est modifié.

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

Le principe des décorateurs

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'appellent des décorateurs de fonction, qui permettent d'altérer le fonctionnement des fonction ou des méthodes de sorte que l'on peut appeler les version 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()
```

Exercice – 10 min

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
```

Décorateur avec des paramètres

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 fléxibilité.

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

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

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

        func()

    return inner
```

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

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

Décorateurs multiples

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 :

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

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

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

Exercice – (15min)

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
```

Le JSON

Le JSON est un format de fichier très utilisé de nos jours, en particulier lorsque l'on travaille avec des API. En Python, il est possible de travailler avec du JSON via l'utilisation du package json. Ce package possède plusieurs fonction utiles telles que :

- `dumps()` : Permet de créer une chaîne de caractère à partir d'un dictionnaire
- `loads()` : Permet de créer un dictionnaire Python à partir d'une chaîne de caractère JSON
- `dump()` : Permet d'écrire un dictionnaire Python dans un fichier au format JSON
- `load()` : Permet de charger un dictionnaire à partir d'un fichier au format JSON

Exercice – 30 min

EXERCICE

- Dans un programme de type console, vous devrez permettre à un utilisateur de voir, de modifier, de supprimer ou d'éditer une série de nom de chien. Cette série de nom devra être persistante via l'utilisation du module json :

```
Liste des chiens
- Bernie

==== MENU PRINCIPAL ====

1. Voir les chiens
2. Ajouter un chien
0. Quitter le programme
Veuillez choisir une fonctionnalité : 0
```

TP – (60 min)

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 instanciés. On peut imaginer la classe comme le récapitulatif des éléments que vont posséder tous les objets de ce type. Par exemple une voiture aurait des roues, un moteur. Un chien aurait 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, comme ci-dessous :

Pour référencer l'objet instancié lors de la déclaration des champs ou des méthodes d'une classe, il faut utiliser le mot-clé **self**. Dans le cas contraire, on risque d'obtenir des problèmes liés au polymorphisme, ou de manipuler des variables de classe, par exemple.

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

    def aboyer(self):
        print("Woof Woof!")

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

Les attributs

Un attribut de classe n'est ni plus ni moins qu'une variable qui est associée à cette classe. Un attribut se voit en général affecter dans le constructeur et peut être accédé via la notation **objet.attribut** dans le cas où l'on souhaite le modifier ou y accéder pour l'utiliser par exemple lors d'un affichage ou d'un calcul.

```
mon_chien.age = 8
print(f"Mon chien a {mon_chien.age} ans, est de race {mon_chien.espece} et porte le patronyme de {mon_chien.nom}")
print(f"Mon chien est donc né en {datetime.date.today().year - mon_chien.age}")
```

Dans le cas où l'on accède à la valeur d'un attribut, on parle généralement d'un **Getter (Get)**, et dans le cas où l'on réaffecte la valeur d'un attribut, on parle alors d'un **Setter (Set)**.

Un objet instancié 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 dehors du scope en cas de modification éventuelle de ses attributs.

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

mon_chien.nom = "Bernie"
print(mon_chien.nom) # Bernie
change_nom(mon_chien, "Bernard")
print(mon_chien.nom) # Bernard
```

Les attributs implicites

Ces attributs sont créés par défaut lors de la manipulation des classes ou d'instances de cette classe. Ils peuvent être accédés via la syntaxe **Dunder** (double underscore) et contiennent par exemple le nom du module qui contient la classe (**module**), le nom de la classe de l'instance (**class**), la liste des attributs de la classe (**dict**) ou un commentaire lui étant associé (**doc**). A l'attribut implicite **classe** sont liés d'autres attributs implicites permettant d'obtenir cette fois-ci les informations suivantes :

- **doc** : Commentaire associé à la classe
- **dict** : La liste de attributs cette fois-ci statiques car liés à la classe
- **name** : Contient le nom de l'instance
- **bases** : Contient la liste des classes dont celle-ci hérite

```
>class ClasseVide:
>    pass
>
>cl = ClasseVide()
>print(cl.__module__)          # __main__
>print(cl.__class__)          # __main__.ClasseVide()
>print(cl.__dict__)           # {}
>print(cl.__doc__)            # None
>print(cl.__class__.__doc__)  # None
>print(cl.__class__.__dict__) # {'__module__': '__main__', '__doc__': None}
>print(cl.__class__.__name__) # ClasseVide
>print(cl.__class__.__bases__)# ()
```

Les méthodes

Une méthode est tout simplement une fonction qui se voit être cette fois-ci associée à un objet ou à une classe. Pour faire appel à une méthode, contrairement aux fonctions, il faudra ainsi utiliser la notation **Classe.méthode()** ou **objet.méthode()**. Une méthode peut accéder aux valeurs de l'objet auquel elle lui est associée de façon très simple, en passant encore une fois par le mot-clé **self**, qu'elle doit d'ailleurs avoir en tant que premier paramètre :

```
def aboyer(self):
    print("Woof Woof!")
```

Une méthode peut réaliser tout ce qu'une fonction faisait de base, mais est en général utilisée pour éviter d'avoir à passer en paramètre des valeurs de variables qui se voient dans le cas d'un objet être affectées à ses attributs :

```
def afficher(self):
    print(f"Mon chien a {self.age} ans, est de race {self.espece} et porte le patronyme de {self.nom}")
```

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.

Le constructeur

Le constructeur est le point d'entrée d'une classe. Il s'agit en fait d'une méthode dite **Dunder ou Magique** (c'est-à-dire une méthode dont le nom commence et fini par deux caractères **underscore** : `__nom_methode__`)

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

Le constructeur est appelé lorsque l'on souhaite instancier une classe :

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

Une fois la variable de type Chien créée, on peut la manipuler et utiliser ses méthodes :

```
mon_chien.afficher()  
print(f"Mon chien est donc né en {datetime.date.today().year - mon_chien.age}")
```

Les propriétés

Les propriété sont en fait une série de trois méthodes magiques qui sont appelées en cas de récupération, d'affectation ou de suppression d'un attribut. Il est ainsi possible de surcharger ces méthodes magiques de sorte à effectuer des opérations avant l'affectation ou avant la récupération des valeurs. Cela évite ainsi d'avoir à répéter des lignes de codes destinées à être réutilisés et évite également la création de méthodes destinées à contrôler et à modifier les affectations ou les récupérations d'attributs d'objets :

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

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

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

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

Les propriétés

Python fournit également un décorateur @property.

La méthode Setter est un peu complexe. Pour créer une méthode de définition, nous devons d'abord qualifier le nom de la propriété après le symbole @, puis un ".setter".

```
@property  
def name(self):  
    return self._name
```

```
@name.setter  
def name(self, value):  
    self._name = value
```

Exercice – (30 min)

EXERCICE

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

Exercice – (60 min)

EXERCICE

En vous servant d'un module (que vous réaliserez vous-même) permettant la génération d'un mot aléatoire parmi ceux présents dans un fichier JSON que vous alimenterez vous-même (Une série de mot par défaut seront disponibles en cas d'absence de ce fichier)

Réalisez une classe Pendu en vous basant sur les méthodes suivantes :

- Une méthode permettant de générer un masque qui sera affiché à l'utilisateur lors de chaque tour
- Une méthode qui permettra de tester la lettre entrée par l'utilisateur à chaque tour
- Une méthode permettant de vérifier si l'utilisateur a actuellement perdu ou gagné le jeu du pendu (L'utilisateur disposera de 8 tentatives infructueuses avant de se voir perdre au jeu)

Vous ferrez ensuite une série d'instruction permettant à un utilisateur de tester votre code dans le cadre d'une partie.

Les attributs de classe

En plus des attributs liés à un objet, il est possible de faire appel à ce qu'on appelle des attributs de classe.

Contrairement à un attribut d'instance, un attribut de classe n'est pas lié à un objet mais à l'ensemble des objets de ce type, soit à la classe elle-même. On peut par exemple se servir des attributs de classe pour compter facilement les instanciations 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:  
    nombre_chien = 0  
    nom_latin = "Canis lupus familiaris"  
  
    def __init__(self, age, nom, espece):  
        self.age = age  
        self.nom = nom  
        self.espece = espece  
        Chien.nombre_chien += 1
```

```
print(f"Il y a en tout {Chien.nombre_chien} chiens d'instanciés et leur nom latin est {Chien.nom_latin}")  
# Il y a en tout 2 chiens d'instanciés et leur nom latin est Canis lupus familiaris
```

Les méthodes de classe

Une méthode de classe est un peu, à l'instar des attributs de classe, une méthode qui est liée non pas à l'objet mais à la classe elle-même. Une méthode de classe devra donc être appelée via la syntaxe **Classe.méthode()** et ne pourra pas accéder aux attributs d'instance vu qu'elle n'est pas liée à un objet.

Pour créer une méthode de classe, il faut se servir du décorateur `@classmethod` à placer avant le nom de la méthode que l'on veut définir :

```
@classmethod
def afficher_chiens(cls):
    print(f"Il y a en tout {cls.nombre_chien} chiens d'instanciés et leur nom latin est {cls.nom_latin}")
```

On peut ainsi voir que tout comme les méthodes d'instance, on se sert d'un mot-clé conventionné (`cls`) dans le but de permettre l'accès plus aisés aux attributs que l'on pourrait vouloir manipuler.

Les méthodes statiques

Une méthode statique est similaire à une méthode classe sauf qu'elle n'a pas recourt au même décorateur ni au mot-clé cls vu précédemment. Elle sert de la même façon à pouvoir réaliser une succession d'instructions en liant la chose à une classe (cela peut être utile par exemple dans le cadre de la réalisation de modules que l'on souhaite exporter sous la forme d'un package regroupant divers fonctionnalités)

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

```
@staticmethod  
def place_dispos(max):  
    print(f"Il reste {max - Chien.nombre_chien} places dans le chenil")
```

```
Chien.place_dispos(10)  
# Il reste 8 places dans le chenil
```

Exercice – 45 min

EXERCICE

- Le but est de créer une classe qui décrit une citerne d'eau WaterTank.
- Cette classe possède un poids à vide, une capacité totale et un niveau de remplissage.
- La classe proposera également les méthodes suivantes :
 - - Une méthode indiquant le poids total de la citerne.
 - - Une méthode pour remplir la citerne avec un nombre de litre d'eau.
 - - Une méthode pour vider la citerne d'eau d'un nombre de litre d'eau.
- La classe possèdera, également, un attribut pour la totalité des volumes des citernes d'eau.
- 1- Créez une classe pour répondre aux conditions.
- 2- Créez un programme pour tester votre classe

L'héritage

L'héritage est un mécanisme fortement utilisé dans la programmation orienté objet. Il consiste en la capacité d'une classe d'hériter d'une autre dans le but de posséder les méthodes et les attributs de la classe parent. On parle alors de classe fille et de classe mère une fois le concept d'héritage organisé et utilisé.

Pour réaliser un héritage en Python, rien de plus simple, il suffit d'ajouter des parenthèses à côté du nom de la classe que l'on crée 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 :

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

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

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

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

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

L'utilisation de la méthode super()

Lors d'un héritage, il est désormais possible d'accéder aux attributs et aux méthodes de la classe mère. Par exemple, si l'on souhaite avoir accès à la méthode **calc_age(annee)** de la classe mammifère dans le but de se servir du résultat dans la classe enfant, alors on doit utiliser le mot-clé **super()** dans le but d'accéder à la classe parent, puis la syntaxe **super().nom_méthode()** pour en déclencher 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(Enfant, self).__init__(nom, prenom, age)
        self.jouet = jouet
```

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

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

    def calc_age(self, annee: int):
        return datetime.date.today().year - annee

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

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

    def age_chien(self, annee):
        return super().calc_age(annee)
```

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 les getters et setters.
- 4-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.

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 `__str__` qui est la méthode utilisée lorsque l'on souhaite afficher un descriptif de l'objet de façon humainement compréhensible, 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).

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

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

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

```
mon_chien = Chien(4, "Rex", "Berger Allemand")
print(mon_chien) # Rex est de race Berger Allemand et a 4 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 en fonction soit de ses paramètres (qui peuvent être différents et ainsi en résultera un polymorphisme de type « **surcharge** ») ou en fonction de l'héritage (ce qui amènera le plus souvent à un polymorphisme de type « **redéfinition** »)

```
ma_liste: list[Personne] = [Personne("John", "MARTIN", 47), Enfant("Mike", "MARTIN", 7, "Bilboquet")]
for item in ma_liste:
    item.jouer() # L'adulte n'a plus le temps de jouer, L'enfant joue avec : Bilboquet
```

On voit ici que lors du parcours de la liste (qui d'ailleurs est typée pour une liste de Personne mais est capable d'accepter les classes enfants de Personne, soit ici un Enfant), on déclenchera alors les méthodes des classes que l'on est en train de parcourir, et non celle de la classe servant de base à la liste (Personne)

```
class Personne:

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

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

class Enfant(Personne):

    def __init__(self, nom, prenom, age, jouet):
        super(Enfant, self).__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 s'amuser à tester les types des classes avant de déclencher 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 = ["Blabla", 12, 3.14]  
mon_dict = {1: "Hello", 2: "World"}  
ma_string = "Bonjour John MARTIN"  
mon_canard = CoinCoin()  
  
print(len(mon_canard)) # 42  
print(len(ma_liste)) # 3  
print(len(mon_dict)) # 2  
print(len(ma_string)) # 19
```

Visibilité en Python : Name Mangling

En Python, tout comme dans beaucoup de langages servant à réaliser de l’Orienté Objet, on a recourt 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 « **object._nom-classe__nom-attribut** »

Ainsi, on a donc recourt à ces convention 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, idéalement abstract, 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 », la classe « Travailleur » 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.
- 3 – Créer une classe « Scientifique », la classe « Scientifique » hérite de la classe « Travailleur » et étend avec les attributs disciplines (physique, chimie, mathématique, ...) et types du scientifique (théorique, expérimental, informatique...) Une méthode `__str__` pour afficher les données

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 recourt 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 même 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 Chien(Animal, Carnivore):
    """Un chien qui est à la fois un animal et un carnivore"""

c = Chien()
c.se_nourrir()
print(c.point_de_vie) # 105
```

```
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

- Soit les classes suivantes :

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

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

- Modifier les classes Person et Address afin qu'elles puissent bien fonctionner dans une hiérarchie à héritage multiple.
- Créer la classe Contact qui hérite à la fois de Address et Person, cette classe doit implémenter la méthode show
- Créer une classe Notebook qui est un dictionnaire qui associe les noms des personnes à un objet Contact. Il n'a pas besoin d'hériter de quoi que ce soit
 - Cette classe devrait avoir une méthode def show(se)
 - Cette classe doit avoir une méthode def add(self, name, email, street, city)

Tester le code suivant:

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

Résultat

```
==== 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.
- Pour 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 chaîne 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 :
 - 2.1. Définir le constructeur de cette classe avec un seul attribut qui est une liste de vols
 - 2.2. Écrire une méthode Liste_successeurs qui retourne une liste contenant les villes arrivées d'une ville de départ passée comme paramètre
 - 2.3. Écrire une méthode Appartient qui vérifie si une ville appartient au plan du vol que ce soit comme ville d'arrivée ou de départ
 - 2.4. Écrire une méthode Affiche qui affiche tous les vols directs.

TP – (60 min) – Suite

TP

- 3. écrire un programme principal permettant de :
- a. Créer une liste 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.
- b. Créer un objet Vol nommé V à partir de la liste déjà créée
- c. Afficher tous les vols
- d. 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

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.

```
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
m1 = MaClasse()
m2 = MaClasse()
m3 = 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 métaclass appelée **ABCMeta** qui permet 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.

```
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)

EXERCICE

- 1 – Créer une classe Interface héritant de abc.ABC qui permet de vérifier qu'un type implémente un certain nombre de méthodes.
- 2 – Créer une classe Container qui impose l'existence de la méthode `__contains__`.
- 3 – Créer une classe Sized qui hérite impose l'existence de la méthode `__len__`.
- 4 – Créer une classe SizedContainer qui impose l'existence de la méthode `__len__` et `__contains__`.
- 5 – Créer une classe Iterable qui impose l'existence de la méthode `__iter__`.

Qu'est-ce qu'une exception ?

Une exception est un problème qui apparaît lors de l'exécution du programme. Les exemples d'exceptions les plus courant 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 bien entendu prendre en compte les erreurs que pourraient causer les utilisateurs et faire en sorte qu'elles soient non bloquantes. En effet, lorsque l'on programme et que 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, rien de plus simple, il suffit de 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** sert 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 de toutes les exceptions 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 soucis ou non

```
value = input(message)
try:
    value = int(value)
except ValueError:
    print("Veuillez entrer un nombre valide SVP")
    continue
else:
    print("Du premier coup ! Bravo !")
finally:
    print("Fin de la vérification de l'exception ValueError")
```

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

```
Veuillez entrer un login SVP (celui-ci ne doit posséder que des lettres minuscules) : aaa
Veuillez 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 !

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

Veuillez entrer un login SVP (celui-ci ne doit posséder que des lettres minuscules) : aa
Veuillez 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 :

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 :

```
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)
```

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

Les méthodes magiques conversions

Il est possible de réaliser des conversions 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 |
|--------------------------------|---|
| <code>__str__(self)</code> | <code>str</code> |
| <code>__bytes__(self)</code> | <code>bytes</code> |
| <code>__bool__(self)</code> | <code>bool</code> OU expression booléenne |
| <code>__int__(self)</code> | <code>int</code> |
| <code>__float__(self)</code> | <code>float</code> |
| <code>__complex__(self)</code> | <code>complex</code> |
| <code>__dict__(self)</code> | <code>dict</code> |

Les méthodes magiques conversions

Opérateurs unaires et arithmétiques

Si les objets doivent pouvoir être utilisés avec les opérateurs unaires +, - 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 |
|--|--------------------------|
| <code><u>add__(self, o)</u></code> | <code>+</code> |
| <code><u>sub__(self, o)</u></code> | <code>-</code> |
| <code><u>mul__(self, o)</u></code> | <code>*</code> |
| <code><u>matmul__(self, o)</u></code> | <code>@</code> |
| <code><u>truediv__(self, o)</u></code> | <code>/</code> |
| <code><u>floordiv__(self, o)</u></code> | <code>//</code> |
| <code><u>mod__(self, o)</u></code> | <code>%</code> |
| <code><u>divmod__(self, o)</u></code> | <code>divmod()</code> |
| <code><u>pow__(self, o, modulo)</u></code> | <code>** OU pow()</code> |

Les méthodes magiques conversions Opérateurs unaires et arithmétiques

```
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)
```

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

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

```
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 plus 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
```

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 de méthodes spéciales telles que :

| Méthode spéciale | Cas d'utilisation |
|--|--|
| <code>__len__(self)</code> | utilisation de la méthode <code>len()</code> |
| <code>__getitem__(self, key)</code> | <code>o[key]</code> |
| <code>__setitem__(self, key, value)</code> | <code>o[key] = value</code> |
| <code>__delitem__(self, key)</code> | <code>del o[key]</code> |
| <code>__contains__(self, key)</code> | <code>key in o</code> |

Les méthodes magiques conteneurs

```
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)
```

```
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 collections.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 hérite d'une des classes du module collections.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 `a`. 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.

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 chaîne 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 (cette méthode remplace 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.

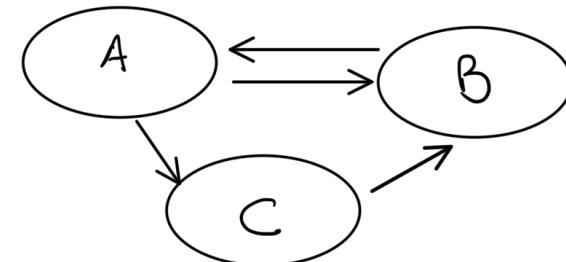
10. Écrire une méthode spéciale `__mul__(self, autre)` qui retourne un nouvel Intervalle multiplication des deux intervalles. Exemple : $[2,5] * [3,4] = [6,20]$

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

TP Global

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.

