Python Initiation Approfondissement

1 Introduction

1.1 Le langage Python

Python est enseigné au collège et au lycée, en particulier en mathématiques, pour l'apprentissage d'un premier langage de programmation, mais aussi pour présenter un domaine fonctionnel. Il est utilisé dans les classes préparatoires ainsi que dans de nombreuses facultés et écoles d'ingénieurs. Python est très utilisé dans le domaine scientifique, en rencherche appliquée, la cryptographie, le big data, ou le cloud par exemple.

Il est versatile et permet de faire des choses aussi différentes que piloter un robot, faire des mathématiques scientifiques (comme MATLAB®) ou encore créer une interface graphique ou un site web. Python a de nombreuses bibliothèques qui sont, comme le langage, gratuites et libres.

Python est un langage de programmation interprété, multiparadigme, de haut niveau, à typage dynamique fort.

- **inteprété** : le code source est tout d'abord compilé en bytecode Python (un fichier *.pyc*) puis interprété par un logiciel par un processus pas à pas : l'intepréteur va transformer le code en langage machine ligne par ligne et l'exécuter en même temps. Cela est à opposer à un langage compilé qui va compiler le code source en code binaire que le système d'exploitation va utiliser pour exécuter les instructions.
  - *Avantages* :
    - le même code source pourra marcher directement sur tout ordinateur. C'est l'interpréteur Python qui se charge de gérer les différences entre les systèmes d'exploitation.
    - la compilation en bytecode Python est légère et rapide.
  - *Inconvénients* :
    - le programme n'est pas directement exécuté en langage machine, et la transformation du bytecode en langage machine a un coût en terme de performance.
- **multiparadigme** : un paradigme est un modèle de programmation qui détermine la formulation des algorithmes, la vue qu'a le développeur de l'éxécution de son programme, et l'organisation du code source. Python implémentent plusieurs paradigmes : le paradigme objet, le paradigme impératif, le paradigme fonctionnel…
- **de haut niveau** : gestion automatisée de toutes les tâches de bas niveau, comme l'allocation mémoire, la libération de la mémoire, la manipulation des registres… Python gère ses ressources, en particulier la mémoire, via un ramasse-miettes.

- **typage dynamique** : un typage statique consiste à déclarer le type des variables en même temps que leur identificateur. Le typage dynamique permet une plus grande souplesse : on peut modifier en Python le type d'une variable au cours de l'exécution de notre programme. Là où les langages statiques doivent implémenter un paradigme générique pour se donner un peu de souplesse, le langage dynamique permet de faire cela naturellement.
- **typage fort** : un typage faible n'accorde de l'importance qu'au contenu tandis que le typage fort accorde également de l'importance au type. En Python, par exemple, la chaîne de caractère "1" et le nombre 1 sont différents.

1.2 Historique

1.2.1 Naissance et évolution

- **1989** : Guido Van Rossum créé sur son temps libre la première version du langage Python, nommé ainsi en l'honneur des Monty Python dont il est fan
- **février 1991** : première version publique 0.9.0
- **1999** : Python est pressenti par un projet lancé en collaboration avec la DARPA (Defense Advanced Research Projects Agency) pour être utilisé comme langage d'enseignement de la programmation. Une équipe est alors dédié au langage.

Année	Version
05/09/2000	version 1.6
16/10/2000	version 2.0
17/04/2001	version 2.1
14/10/2002	version 2.2
29/07/2003	version 2.3
30/03/2005	version 2.4
19/09/2006	version 2.5
01/10/2008	version 2.6
03/07/2010	version 2.7
03/12/2008	version 3.0
27/06/2009	version 3.1
20/02/2011	version 3.2
29/09/2012	version 3.3
16/03/2014	version 3.4
13/06/2015	version 3.5
23/12/2016	version 3.6
27/06/2018	version 3.7
14/10/2019	version 3.8
05/10/2020	version 3.9
04/10/2021	version 3.10
03/10/2022	version 3.11

1.2.2 A propos de Python 2 et Python 3

La version 2.X était le passage pour Python d'un statut de petit langage spécifique à celui d'un langage de référence complet.

L'enjeu de la branche 3.X était de capitaliser, homogénéiser, et stabiliser le langage. C'est pour cela qu'il a été nécéssaire de casser la compatibilité avec les versions antérieures et qu'une nouvelle branche a été crée.

Des outils sont disponibles pour transiter le code 2.X vers 3.X. Initialement, la branche 2.X était prévue pour s'arrêter (support technique et résolution des problèmes de sécurité) en 2015, mais face à la très grande diversité de bibliothèques, d'outils ou logiciels développés en Python, leur énorme diversité, ainsi que la complexité de leur migration et le fait que certains d'entre eux sont des piliers du logiciel libre, la branche 2.X s'est réellement arrété le 01/01/2020.

1.3 Installation

1.3.1 Installation de Python dans un environnement Windows ou Linux

Télécharger Python : https://www.python.org/downloads/. Si une version est déjà installée, la mettre à jour.

Windows / Mac :

- télécharger la dernière version python et exécuter l'installateur
- cocher la case Add Python 3.X to PATH
- install now (ne pas customiser l'installation)

Linux :

- télécharger le code source
- la décompresser
- se placer dans le répertoire obtenu

```
$ ./configure -prefix=/path/to/my/python/directory
$ make
$ sudo make altinstall
```

***note** : Mac/Linux utilisent en interne des versions de Python et ont déjà une version installée.*

1.3.2 Mise en oeuvre de Python : accès au terminal

Dans un terminal, vous pouvez vérifier que l'installation de Python et son ajout au path ont bien été effectués en utilisant la commande :

```
> py
```

qui permet d'avoir la version de Python et de lancer le terminal Python. On peut y écrire des commandes et du code.

```
Python 3.9.4 (tags/v3.9.4:1f2e308, Apr  6 2021, 13:40:21) [...] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world")
Hello world
```

Pour arrêter le terminal Python, vous pouvez écrire `exit()` ou faire `Ctrl+Z` puis `entrée`.

Note : Si la dernière version de Python ne correspond pas à celle de ce document, ce n'est pas un problème. Pour les bases de Python que nous aborderons dans ce cours, ça ne fait pas de différence.

Python est bien installé, mais pourquoi avoir ajouté Python au path ? Lorsqu'on souhaite exécuter un programme depuis un terminal, on peut le faire de deux manières différentes. La première, c'est d'indiquer le chemin vers l'exécutable. La commande suivante :

```
> C:/.../py.exe
```

permet d'exécuter le programme py.exe situé au chemin spécifié. On peut s'abstenir d'écrire systématiquement le chemin complet vers l'exécutable (ce qui peut être fastifieux et lourd à la lecture pour des commandes plus complexes) en utilisant le path. Lorsqu'on fait la commande :

```
> py
```

Windows a une *variable d'environnement*, le path, qui est un ensemble de répertoires dans lequel il va chercher un exécutable py.exe. Si vous avez le message suivant :

```
> py
'py' n'est pas reconnu en tant que commande interne
ou externe, un programme exécutable ou un fichier de commandes.
```

c'est, comme le message l'indique, que *py* n'est pas une commande connue par windows, et qu'il n'a pas été trouvé comme programme exécutable ou fichier de commandes dans les répertoires qui sont contenus dans le path. Vous pouvez modifier les variables d'environnement manuellement depuis les paramètres windows une fois l'installation effectuée : il suffira d'y ajouter le chemin vers le répertoire qui contient l'exécutable Python *py.exe*.

1.3.3 Environnements de développement intégrés

Pour coder, on ne va pas se limiter à utiliser le terminal Python, mais on va utiliser un IDE (integrated development environment), un environnement de développement intégré. Il existe énormément de solutions. L'utilisation d'un IDE permet au développeur de faciliter son travail en proposant une quantité d'outils adaptés : l'autocomplétion, la gestion des erreurs de compilation, l'accès à la documentation, l'utilisation plus aisée de plusieurs fichiers, des codes de couleurs lors de l'écriture du code pour mieux le visualiser, …

Nous allons utiliser VSCode (Visual Studio Code : https://code.visualstudio.com/) qui est un IDE open source qui permet de développer dans une multitude de langages grâce à l'utilisation de nombreuses extensions.

Une fois installé nous allons utiliser 4 extensions VSCode pour développer en Python :

- Python
- Pylint (TODO)
- SQLITE
- French Language Pack (optionnel)

un redémarrage de VSCode sera peut-être nécessaire pour que ces extensions fonctionnent correctement.

1.3.4 Executer du code Python depuis VSCode

Pour écrire du code Python, il suffit de créer un fichier avec l'extension *.py* et d'y mettre le code qu'on souhaite. Pour exécuter du code Python, il faut écrire dans un terminal :

```
> py /path/to/my/Python/file.py
```

Heureusement, VSCode nous permet de nous simplifier la tâche en proposant un bouton `play` attaché à l'éditeur de notre fichier. Voici un exemple de code Python :

```python
print("Hello world") # ceci est un commentaire
# ce code est ignoré
# Ce fichier contient une seule ligne de code : il s'agit d'une instruction.
# Elle contient deux éléments
# une fonction print et un littéral "Hello World".
```

```
Hello world
```

Néanmoins, il faut retenir que le bouton `play` proposé par VSCode ne fait rien d'autre que ***d'aller écrire dans le terminal*** la commande appropriée. Cliquer sur ce bouton va écrire dans la console la commande suivante :

```
& "path/to/python.exe" "path/to/file.py"
```

Important : cliquer sur le bouton `play` va ***écrire quelque chose dans la console***. Il faudra être attentif. Dans le cas où votre programme attend un input utilisateur, **appuyer sur le boutton `play` ne redémarrera pas votre programme**, mais va simuler un input utilisateur contenant la commande Python décrite plus haut. Autrement dit, si vous souhaiter relancer votre programme, **pensez bien à le quitter, puis à l'exécuter à nouveau** avec le bouton `play`. Vous pouvez forcer l'arrêt d'un programme en cours d'exécution en faisant `Ctrl + C` dans la console.

1.3.5 Raccourcis clavier VSCode

Les raccourcis que nous utiliserons le plus :

- **ctrl + /** : commenter / décommenter une ligne ou toutes les lignes séléctionnées (pas celui du pavé numérique, mais celui sur le même bouton que ':')
- **alt + shift + bas** : dupliquer la ligne selectionnée
- **ctrl + clic** *sur un élément* : ramène à sa définition

Vous pouvez consulter également l'ensemble des raccourcis VSCode : https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf

1.3.6 Documentation Python

La documentation Python peut être directement consultée depuis : https://docs.python.org/fr/3/

2 Variables

2.1 Définition et déclaration de variable

Une variable est une zone mémoire qui contient une valeur typée à laquelle on peut accéder dans notre programme via un nom. On peut y accéder pour récupérer la valeur, ou pour la modifier.

Il y a plusieurs types de variables en Python :

- types numériques : les entiers **int**, les nombres flottants **float**, les nombres complexes **complex**
- type textuel : les chaînes de caractères **str**
- type boolean : les booléens **bool**
- types listes : **range**, **tuple**, **list**
- types de mapping : les dictionnaires **dict**
- types d'ensembles : les ensembles **set**, *les frozenset** ,
- *types binaires* : memoryview, bytearray, bytes*

** nous n'en parlerons pas dans ce cours*

Pour déclarer une variable : `NOM = VALEUR`

Il y a quelques règles à respecter dans le nommage des variables :

- le nom d'une variable ne peut contenir que des lettres, chiffres et des underscores (_). Pas d'accent, d'espace ou de tiret (-)
- le nom d'une variable ne peut pas commencer par un chiffre
- le nom d'une variable est sensible aux majuscules. Age, AGE et AgE sont 3 variables différentes.
- On ne peut pas utiliser de mot clé python pour créer nos variables. La liste des mots clés : https://www.w3schools.com/python/python_ref_keywords.asp

---

```
entier = 15  # Python comprend qu'il s'agit d'un entier
flottant = 3.14159265358979323  # Python comprend qu'il s'agit d'un flottant
complexe = 4 + 2j # j pour le nombre imaginaire
texte = "Bonjour"  # Python reconnaît du texte
#texte = 'Bonjour' est exactement la même chose
erreur = True # Mot clés pour les booléens : True et False

print(entier)
print(flottant)
print(texte)
print(erreur)
print(complexe)
```

```
15
3.141592653589793
Bonjour
True
(4+2j)
```

2.2 Interraction avec l'utilisateur

Pour inviter l'utilisateur à saisir quelque chose et mettre le texte dans une variable, on peut utiliser la fonction `input`. Cette fonction prend en paramètre un message qu'elle affiche, puis place un curseur juste après. Tant que l'utilisateur n'a pas appuyé sur entrée, le programme restera figé. Cette fonction renvoie une **str** qui contient ce que l'utilisateur a écrit.

```
saisie = input("Entrez du texte : ")
# print peut etre utilisé pour afficher plusieurs choses
# Elles seront séparées par des espaces lors de l'affichage
print("Vous avez saisi :", saisie)
```

```
Entrez du texte : Coucou !
Vous avez saisi : Coucou !
```

2.3 Bonnes pratiques PEP

Un ensemble de bonnes pratiques est réunie dans le guide stylistique Pyhon appelé la PEP (Python Enhancement Proposals). Entre autres, il y a des indications sur le nommage des variables : https://www.python.org/dev/peps/pep-0008/#naming-conventions

- le nom des variables. Il existe plusieurs manières de nommer ces variables. En voici quelques unes :
  - camelCase : tout en minuscules, sauf les initiales et première lettre en minuscules. ceciEstMaVariable
  - PascalCase : tout en minuscules, sauf les initiales. CeciEstMaVariable
  - snake_case : tout en minuscules, et les mots sont séparés par des _. nom_de_la_variable

  en Python : le nom des variables est en snake_case
- le nom des (pseudo-)constantes en snake_case majuscule: NOM_DE_LA_CONSTANTE
- toujours avoir des noms de variables explicites (on évite les abréviations etc..)

```
mon_nombre = 123430
print(mon_nombre)
# il est possible de rajouter des _ pour améliorer
# la lisibilité des grands nombres
mon_autre_nombre = 456_987
print(mon_autre_nombre)

# on a une certaine liberté dans l'écriture des nombres flottants
nombre = 0.99
print(nombre)
nombre = 00.99
print(nombre)
nombre = .99
print(nombre)
```

```
123430
456987
0.99
0.99
0.99
```

Une pseudo constante fonctionne comme une variable normale, mais n'est pas *censée* être modifiée. Il n'est pas directement possible d'empêcher la modification de la valeur d'une variable en Python. Par conséquent, il est *techniquement possible* de modifier une pseudo constante. Mais le fait qu'elle ait un nom en snake case majuscules est une indication qu'elle n'est pas censée varier. Il s'agit d'une **convention** qui permet d'aider à la lecture et à la compréhension du code.

```python
PSEUDO_CONSTANTE = 123_456
print(PSEUDO_CONSTANTE)
PSEUDO_CONSTANTE = 2 # n'est pas une bonne pratique
# ne pas changer la valeur d'une pseudo constante
print(PSEUDO_CONSTANTE) # c'est techniquement possible
```

```
123456
2
```

2.4 Typage dynamique fort

Python est dynamiquement fortement typé. Les types sont implicites et une variable peut changer de type à n'importe quel moment. Par contre, on ne peut pas faire d'opération entre les types différents.

La fonction `type` permet d'obtenir le type d'une variable. Néanmoins, en Python, tout est objet, et la fonction `type` renvoie un résultat typé, de type *type*. Dans l'exemple suivant, on peut observer une variable dont la valeur change de type pour illustrer le *typage dynamique* de Python.

```python
ma_variable = "Ceci est un texte"
print(type(ma_variable))

ma_variable = 15
print(type(ma_variable))

print(type(type(ma_variable)))
```

```
<class 'str'>
<class 'int'>
<class 'type'>
```

Néanmoins, le typage de Python est un *typage fort*. On ne peut pas manipuler des types différents directement, comme l'illustre l'exemple suivant.

```
test = 15 + "4"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [7], in <cell line: 1>()
----> 1 test = 15 + "4"

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

2.5 Exercices

1) Créer 3 variables avec des noms corrects et les afficher

See *Answer*

2) Demander à l'utilisateur son prénom, puis lui dire bonjour personnellement

See *Answer*

3 Formatage

3.1 Print avec plusieurs arguments

Pour afficher le contenu de variables dans la console, on peut utiliser la fonction `print` avec plusieurs arguments. Le contenu sera alors affiché en séparant les différents éléments par des espaces.

```
age = 64
prenom = "Maude"

print("Age :", age, "Prénom :", prenom)
```

```
Age : 64 Prénom : Maude
```

3.2 Concaténation de str

On peut aussi utiliser la concaténation des chaînes de caractères. `str1 + str2` retourne une chaine de caractères composé du contenu de `str1` suivi du contenu de `str2` (aucun espace n'est ajouté).

Il s'agit d'une concaténation de chaînes de caractères. Si une des variables n'est pas une **str**, il faut donc faire une conversion en **str** en utilisant la fonction `str`.

```
age = 64
prenom = "Maude"

print("Age : " + str(age) + ", prenom : " + prenom)
```

```
Age : 64, prenom : Maude
```

3.3 Format

A partir de Python 3, toutes les **str** ont une méthode qu'on peut utiliser pour mettre en forme une chaîne de caractères. On peut y insérer les valeurs contenues dans certaines variables en utilisant la méthode `format`. Avant Python 3, il existe une syntaxe similaire qui utilise `%`.

```
age = 64
prenom = "Maude"

# A partir de python 3
print("Age : {}, prenom : {}".format(age, prenom))

# Avant python 3
print("Age : %d, prenom : %s" % (age, prenom))
```

```
Age : 64, prenom : Maude
Age : 64, prenom: Maude
```

Il est possible de mettre en forme les données de manière plus précise, comme par exemple d'arrondir les chiffres à une certaine précision après la virgule. La documentation fournit les règles et des exemples : https://docs.python.org/3/library/string.html#formatspec.

```
points = 19
total = 22
taux = points/total

print('{}/{} - Taux de réponses correctes: {:.2%}'.format(points, total, taux))
```

```
19/22 - Taux de réponses correctes: 86.36%
```

3.4 fstring

A partir de Python 3.6, les *f-string* permettent d'afficher des variables de manière plus facilement lisible, en insérant directement dans l'écriture de la *f-string* les variables. On peut également mettre en forme les données à la manière décrite dans la documentation : https://docs.python.org/3/library/string.html#formatspec.

```
age = 64
prenom = "Maude"

# f"..." pour créer une f-string
print(f"Age : {age}, prenom : {prenom}")

points = 19
total = 22
taux = points/total

print(f'{points}/{total} - Taux de réponses correctes: {taux:.2%}')
```

```
Age : 64, prenom : Maude
19/22 - Taux de réponses correctes: 86.36%
```

Il est également possible d'écrire le nom d'une variable et sa valeur en utilisant le symbole `=`. En terminant l'expression Python dans entre les accolades par un `=`, le code Python sera affiché et son résultat sera affiché.

```
age = 64

print(f"{age = }")
```

```
print(f"{5 * 6 = }")

# sans cette méthode, on est obligés de se répéter
# et on peut donc faire une erreur et avoir un affichage incohérent
print(f"age = {age}")
print(f"5 * 6 = {4 * 6}") # oups !
```

```
age = 64
5 * 6 = 30
age = 64
5 * 6 = 24
```

3.5 Caractères spéciaux, échappement de caractères

On a vu qu'on pouvait utiliser les doubles ou simple guillemets (" ou ') pour définir des chaînes de caractères. Comment utiliser ces guillemets dans une chaîne ?

```
chaine = "Ceci est un "test" de mettre des guillemets"
print(chaine)
```

```
  Input In [70]
    chaine = "Ceci est un "test" de mettre des guillemets"
                          ^

SyntaxError: invalid syntax
```

Les guillemets sont vus comme étant la fin de chaîne de caractères par l'interpréteur Python.

```
chaine = "Ceci est un 'test' de mettre des guillemets"
print(chaine)
```

```
Ceci est un 'test' de mettre des guillemets
```

Simple, il suffit de changer de guillemets pour définir les chaînes ! C'est une solution, mais que se passe-t-il si on souhaite utiliser les deux guillemets ? On peut aussi s'en sortir en utilisant le caractère \, qui est un caractère d'échappement qui permet d'afficher des caractères spéciaux. Par exemple, il permet de faire un retour à la ligne $\backslash n$, ou une tabulation $\backslash t$

```
chaine = "Ceci est un \"test\" de mettre des guillemets.\nUne nouvelle␣
 ↪ligne\ttabulation\n\tCoucou !"
```

```
print(chaine)
```

```
Ceci est un "test" de mettre des guillemets.
Une nouvelle ligne      tabulation
        Coucou !
```

3.6 Exercices

1) Demander à l'utilisateur sa rue, son code postal et sa ville puis afficher l'adresse complète

See *Answer*

4 Opérateurs

4.1 Expressions

Une expression est une série d'instructions qui renvoie une valeurs que l'on pourrait mettre dans une variable ou utiliser comme argument d'une fonction.

```
a = 15
# 15 est une expression
b = a # a est une expression
print(f"a = {a}")
print(f"b = {b}")

non_defini = None # valeur spéciale considérée comme "pas de valeur"
print(f"Non défini : {non_defini}")
```

```
a = 15
b = 15
Non défini : None
```

Les opérateurs permettent d'écrire des expressions, et de faire des transformations entre plusieurs expressions.

4.2 Opérateurs arithmétiques

4.2.1 Addition

```
a = 2
b = 3
c = a + b

print(f"a = {a}")
print(f"b = {b}")
print(f"c = {a} + {b} = {c}")

c += 2 # raccourci pour c = c + 2
print(f"après c+= 2, {c = }")
```

```
a = 2
b = 3
c = 2 + 3 = 5
après c+= 2, c = 7
```

On souhaite laisser le choix à l'utilisateur des nombres qu'on va additionner.

Attention : la fonction input renvoie une **str** , et l'opérateur **+** sur les chaînes de caractères est *la concaténation*

```
a = input("a = ")
b = input("b = ")
c = a + b

print(f"c = {a} + {b} = {c}")
```

```
a = 2
b = 3
c = 2 + 3 = 23
```

Pour avoir cette interactivité et bien faire l'addition des nombres proposés par l'utilisateur, il faut **convertir** la **str** en type numérique.

```
a = input("a = ")
b = input("b = ")
c = int(a) + int(b) # on additionne les valeurs converties en int

print(f"c = {a} + {b} = {c}")
```

```
a = 2
b = 3
c = 2 + 3 = 5
```

Mais que se passe-t-il si l'utilisateur n'écrit pas un nombre ?

```
a = input("a = ")
b = input("b = ")
c = int(a) + int(b) # on additionne les valeurs converties en int

print(f"c = {a} + {b} = {c}")
```

```
a = 14
b = pas un nombre
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [18], in <cell line: 3>()
      1 a = input("a = ")
      2 b = input("b = ")
----> 3 c = int(a) + int(b) # on additionne les valeurs converties en int
      5 print(f"c = {a} + {b} = {c}")
```

```
ValueError: invalid literal for int() with base 10: 'pas un nombre'
```

Une erreur se produit, et le programme s'arrête. Nous verrons comment gérer cela dans la partie sur les *exceptions*. Il est néanmoins important d'avoir le réflexe de ne **jamais faire confiance** en l'utilisateur. Dans un cas comme celui-ci, négliger de se poser la question *Mais que se passe-t-il si l'utilisateur n'écrit pas un nombre ?* ferait passer à côté d'un bug potentiel de l'application. Dans d'autres contextes, la communication avec l'utilisateur peut poser des problèmes de **sécurité**.

4.2.2 Soustraction

```
a = 5
b = 2
c = a - b

print(f"a = {a}")
print(f"b = {b}")
print(f"c = {a} - {b} = {c}")

c -= 2 # raccourci pour c = c - 2
print(f"après c-= 2, {c = }")
```

```
a = 5
b = 2
c = 5 - 2 = 3
après c-= 2, c = 1
```

Dans le cas de l'interactivité avec l'utilisateur, il faut toujours bien effectuer une conversion de **str** en type numérique pour avoir le résultat attendu. Néanmoins, comme l'opération - n'existe pas sur les **str**, un oubli produirait cette fois une erreur

---

```
a = input("a = ")
b = input("b = ")
c = a - b

print(f"c = {a} - {b} = {c}")
```

```
a = 15
b = 9
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [20], in <cell line: 3>()
      1 a = input("a = ")
```

```
      2 b = input("b = ")
----> 3 c = a - b
      5 print(f"c = {a} - {b} = {c}")

TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Avec une conversion, tout fonctionne correctement...

```
a = input("a = ")
b = input("b = ")
c = int(a) - int(b) # on soustrait les valeurs converties en str

print(f"c = {a} - {b} = {c}")
```

```
a = 15
b = 9
c = 15 - 9 = 6
```

... sous réserve que l'utilisateur entre bien des nombres et que la conversion peut par conséquent bien s'effectuer.

```
a = input("a = ")
b = input("b = ")
c = int(a) - int(b) # on soustrait les valeurs converties en str

print(f"c = {a} - {b} = {c}")
```

```
a = 8
b = pas un nombre
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [22], in <cell line: 3>()
      1 a = input("a = ")
      2 b = input("b = ")
----> 3 c = int(a) - int(b) # on soustrait les valeurs converties en str
      5 print(f"c = {a} - {b} = {c}")

ValueError: invalid literal for int() with base 10: 'pas un nombre'
```

Ces remarques concernant l'interaction utilisateur sont valables pour tous les prochains opérateurs. Ils ne sont pas disponibles entre chaînes de caractères, il faut faire une conversion de type pour les utiliser avec des entrées utilisateur. Néanmoins, il faut être attentif aux entrées utilisateur et anticiper des comportements différents de ceux demandés.

Cette dernière remarque sur la **confiance** à l'utilisateur se généralise à toutes les interactions que peuvent avoir notre application, que ce soit pour *éviter des bugs*, ou pour des *questions de sécurité*.

4.2.3 Multiplication

```
a = 4
b = 5
c = a * b

print(f"a = {a}")
print(f"b = {b}")
print(f"c = {a} * {b} = {c}")

c *= 3
print(f"après c *= 3, {c = }")
```

```
a = 4
b = 5
c = 4 * 5 = 20
après c *= 3, c = 60
```

4.2.4 Division

```
a = 4
b = 5
c = a / b

print(f"a = {a}")
print(f"b = {b}")
print(f"c = {a} / {b} = {c}")

c /= 2
print(f"après c /= 2, {c = }")
```

```
a = 4
b = 5
c = 4 / 5 = 0.8
après c /= 2, c = 0.4
```

4.2.5 Puissance

```
a = 2
b = 3
c = a ** b

print(f"a = {a}")
print(f"b = {b}")
print(f"c = {a} ** {b} = {c}")

c **= 2
print(f"après c **= 2, {c = }")
```

```
a = 2
b = 3
c = 2 ** 3 = 8
après c **= 2, c = 64
```

4.2.6 Division entière

```
a = 11
b = 2
c = a // b

print(f"a = {a}")
print(f"b = {b}")
print(f"c = {a} // {b} = {c}")

c //= 2
print(f"après c //= 2, {c = }")
```

```
a = 11
b = 2
c = 11 // 2 = 5
après c //= 2, c = 2
```

4.2.7 Modulo

```
a = 14
b = 5
c = a % b

print(f"a = {a}")
print(f"b = {b}")
print(f"c = {a} % {b} = {c}")

c %= 2
print(f"après c %= 2, {c = }")
```

```
a = 14
b = 5
c = 14 % 5 = 4
après c %= 2, c = 0
```

Si vous ne connaissez pas la division entière ou le modulo, il s'agit respectivement du quotient et du reste de la division euclidienne, qui correspond au calcul de division qu'on peut faire à la main. https://fr.wikipedia.org/wiki/Division_euclidienne

4.3 Opérateurs de comparaison

Il est possible d'utiliser les opérateurs de comparaison comme inférieur strictement à `<`, inférieur ou égal à `<=`, supérieur strictement à `>`, supérieur ou égal à `>=`.

```python
a = 7
b = 5

print(f"{a = }, {b = }")

resultat = a > b
print(f"a > b : {resultat}")

resultat = a >= b
print(f"a >= b : {resultat}")

print(f"{a < b = }")

resultat = a <= b
print(f"{a <= b = }")
```

```
a = 7, b = 5
a > b : True
a >= b : True
a < b = False
a <= b = False
```

On peut aussi comparer le contenu de deux variables, pour voir si elles sont égales `==` ou différentes `!=`.

```python
a = 5
b = 5

print(f"{a = }, {b = }")
```

```python
# = : affectation d'une valeur à une variable
# == : opérateur de comparaison
resultat = a == b
print(f"a == b ? {resultat}")

prenom1 = "Anne"
prenom2 = "David"

print(f"{prenom1 = }, {prenom2 = }")

print(f"{prenom1 != prenom2 = }")
```

```
a = 5, b = 5
a == b ? True
prenom1 = 'Anne', prenom2 = 'David'
prenom1 != prenom2 = True
```

4.4 Autres opérateurs

Il existe d'autres opérateurs :

- des opérateurs logiques : or, and not
- des opérateurs logiques par bit : |, &, ~
- is, in, ...

Nous aurons l'occasion de recroiser les opérateurs logiques ainsi que d'autres opérateurs dans la suite du cours. Les opérateurs logiques par bit sont plus rarement utilisés, mais dans certaines situations spécifiques, il est possible que le besoin pour ces opérateurs se fasse ressentir.

Idée des opérateurs par bit : les nombres sont convertis en binaires et les opérations se font bit par bit, en interprétant le 0 comme False et le 1 comme True. Le nombre binaire trouvé est alors reconverti en décimal pour obtenir le résultat.

```python
# 5 | 3 : 5 OU par bit 3
# 5 | 3 -> 101 | 011 -> 111 -> 7
print(f"{5 | 3 = }")
# 5 | 3 : 5 ET par bit 3
# 5 & 3 -> 101 & 011 -> 001 -> 1
print(f"{5 & 3 = }")
```

```
5 | 3 = 7
5 & 3 = 1
```

5 Conditions

5.1 Bloc conditionnel

Un bloc conditionnel est un bloc qui ne s'exécute que si une condition est évaluée à vraie.

Un bloc est une section de code marqué par deux points ":" ainsi qu'une indentation sur la ligne suivante. Tant que le code est indenté au moins au même niveau, on est dans le bloc.

Il est donc nécessaire en Python d'être attentif à l'indentation de son code, puisqu'elle a du sens.

La syntaxe est la suivante :

```
# début du programme

if condition:
    code_execute_si_condition_True()

# suite du programme
```

```
age = int(input("Age ? "))

if age < 18:
    print("Mineur")
    print("Toujours mineur")
print("Tout âge")
```

```
Age ? 14
Mineur
Toujours mineur
Tout âge
```

```
Age ? 24
Tout âge
```

On peut utiliser le mot clé `else` pour créer un bloc qui sera exécuté lorsque la condition est False. Si on souhaite rajouter d'autres conditions, on peut utiliser le mot clé `elif` (à lire comme une contraction de 'else if').

```
age = int(input("Age ? "))

if age < 18:
  print("moins de 18 ans")
elif age <= 25:
  print("jeune adulte")
```

```
else:
  print("adulte")
```

```
Age ? 14
moins de 18 ans
```

```
Age ? 24
jeune adulte
```

```
Age ? 34
adulte
```

On peut utiliser n'importe quelle expression qui renvoie un booléen.

```
age = 24

if age >= 18 and age <= 25:
  print("Vous avez entre 18 et 25 ans")
```

```
Vous avez entre 18 et 25 ans
```

On a une certaine flexibilité en Python pour l'écriture des conditions.

```
if 18 <= age <= 25:
  print("jeune adulte")
```

```
jeune adulte
```

5.2 Bloc vide et mot clé pass

Un bloc vide n'est pas valide en Python.

```
condition = True
if condition:
    # les commentaires sont ignorés ! Le bloc est vide !
# suite du code
print("...")
```

```
  Input In [38]
    print("…")
```

```
    ^
IndentationError: expected an indented block
```

Néanmoins, il est possible de créer un bloc syntaxiquement valide sans rien y faire en utilisant un mot clé spécifique : `pass`.

```python
if True:
  pass

age = 10

if age<= 18:
  print("mon code en développement...")
elif age <= 25:
  pass # pas encore développé, mais avec le pass, pas d'erreur
# de syntaxe
else:
  pass
```

```
mon code en développement…
```

Quelle est l'utilité de faire un bloc vide ? On peut simplement éviter de mettre la condition, le else, ou le elif tout court ! Dans le contexte des blocs conditionnels, il peut être utile d'utiliser des blocs vides lors du développement d'une partie de l'application. Cela permet de construire la logique conditionnelle, et de coder les différents blocs au fur et à mesure sans avoir d'erreur de compilation. Les blocs vides peuvent avoir leur intérêt dans d'autres contextes d'utilisation de blocs.

5.3 Opérateur ternaire

L'opérateur ternaire permet d'écrire de manière condensée la logique suivante

---

```python
age = 15

if age < 18:
  print("mineur")
else:
  print("majeur")
```

```
mineur
```

---

en utilisant la syntaxe

`code_execute_si_True if condition else code_execute_si_False`

Ainsi, le code précédent peut se réécrire en utilisant l'opérateur ternaire.

```
age = 15

print("mineur") if age < 18 else print("majeur")

# on peut aussi assigner une valeur avec l'opérateur ternaire
message = "mineur" if age <18 else "majeur"
print(message)
```

```
mineur
mineur
```

5.4 Exercices

1) Demander à l'utilisateur un nombre. Afficher ensuite si ce nombre est pair ou impair

See *Answer*

2) Demander à l'utilisateur un mot. Afficher si ce mot contient la lettre "e" ou non (utiliser l'opérateur in)

See *Answer*

6 Boucles

6.1 Bloc itératif

Un bloc d'instruction peut être répété un certain nombre de fois. Il y a deux types de bloc itératifs :

- si le nombre d'instructions est connu à l'avance on utilise la boucle **for**, qui répète les instructions à mesure qu'on avance dans le parcours d'une suite d'éléments (une liste, un tuple, un dictionnaire, un set, un string)

```
# Afficher les nombres de 0 à 9
nombres = range(10)  # une collection de 10 entiers de 0 à 9 inclus
for nbr in nombres:  # Le bloc sera exécuté 10 fois
  print(nbr)  # `nbr` contient tour à tour les nombres de 0 à 9
```

```
0
1
2
3
4
5
6
7
8
9
```

- si le nombre d'instructions n'est pas connu à l'avance mais dépend d'une condition, on utilise la boucle **while**, qui répète les instructions tant qu'une certaine condition est vérifiée

---

```
# Demander un nombre compris entre 1 et 10 à
# l'utilisateur, et lui redemander tant que le nombre
# n'est pas bon
nombre = 0
while (not 1 <= nombre <= 10):
  nombre = int(input("Saisir un nombre entre 1 et 10 : "))
print(f"Ok ! Voici votre nombre : {nombre}")
```

```
Saisir un nombre entre 1 et 10 15
Saisir un nombre entre 1 et 10 -8
Saisir un nombre entre 1 et 10 3
Ok ! Voici votre nombre : 3
```

Il est néanmoins possible d'utiliser une boucle **while** pour effectuer un nombre déterminé de fois

un bloc. Ce n'est cependant pas une écriture aussi claire et concise qu'une boucle **for**.

```
compteur = 1
while compteur <= 5:
  print(f"Ligne {compteur} : Je ne dois pas recopier sans comprendre")
  compteur += 1

# il faut être attentif à la condition de la boucle
# while pour éviter de faire des boucles infinies !
# while True:
#   print("Je ne m'arrête pas !")
```

```
Ligne 1 : Je ne dois pas recopier sans comprendre
Ligne 2 : Je ne dois pas recopier sans comprendre
Ligne 3 : Je ne dois pas recopier sans comprendre
Ligne 4 : Je ne dois pas recopier sans comprendre
Ligne 5 : Je ne dois pas recopier sans comprendre
```

Vous pouvez forcer l'arrêt de l'exécution du script Python que vous avez exécuté en faisant **ctrl** + **c** dans la console. Cela peut notamment vous servir si vous faites une boucle infinie par accident.

6.2 Parcourir une chaine de caractères

On peut utiliser les crochets [et] pour parcourir une chaine de caractères, en utilisant la syntaxe *chaine[index]*.

```
prenom = "Erwan"

print(f"Prénom : {prenom}")
print("Charbactère en position 0 : " + prenom[0])
print("Charactère en position 1 : " + prenom[1])
```

```
Prénom : Erwan
Charbactère en position 0 : E
Charactère en position 1 : r
```

On peut donc utiliser une boucle **while** pour parcourir une chaîne de caractères.

```
compteur = 0
while compteur < len(prenom):
  print(f"Charactère en position {compteur} : {prenom[compteur]}")
  compteur+=1
```

```
Charactère en position 0 : E
Charactère en position 1 : r
Charactère en position 2 : w
Charactère en position 3 : a
Charactère en position 4 : n
```

6.3 Parcourir une chaine de caractères avec un for

On peut directement parcourir une chaîne de caractères avec une boucle for.

```
prenom = "Sylvain"
for lettre in prenom:
  print(lettre)
```

```
S
y
l
v
a
i
n
```

6.4 Mots clés continue et break

Le mot clé continue permet de forcer le passage à l'itération suivante le mot clé break permet de forcer l'arrêt de la boucle.

```
prenom = "Sylvain"
for lettre in prenom:
  if (lettre == "y"):
    continue
  if (lettre == "i"):
    break
  print(lettre)
```

```
S
l
v
a
```

6.5 Exercices

1) Afficher tous les nombres de 1 (inclus) à 15 (inclus)

See *Answer*

2) Afficher tous les nombres pairs entre 1 (inclus) et 10 (inclus)

See *Answer*

3) Afficher les 20 premiers termes de la suite de fibonacci. Elle est définie de la manière suivante : Les premiers termes sont 0 et 1, et les termes suivants sont définis comme étant la somme des deux termes précédents. Autrement dit, fibo(0) = 0, fibo(1) = 1, et fibo(n+2) = fibo(n) + fibo(n+1)

See *Answer*

4) Afficher un menu : Menu
   1) Convertir minutes en heures
   2) Quitter Votre choix ? Et implémenter les fonctionnalités. Par exemple, la conversion transformera 90 minutes en 1 h 30 minutes

See *Answer*

7 Exceptions

Il y a 3 types d'erreurs :

- **Les erreurs de compilation** : détéctées par Visual Studio Code
- **Les exceptions** : les erreurs qui stoppent l'exécution du programme https://docs.python.org/3/library/exceptions.html
- **Les erreurs qui n'arrêtent pas l'application** : c'est lorsque le programme fonctionne, mais n'a pas le comportement désiré

```
a = "4"
b = "5"
addition = a + b
print(addition)
```

```
45
```

Pour gérer ces différents types d'erreur, on a plusieurs moyens pour les diagnostiquer et les résoudre.

- **Les erreurs de compilation** : détectées et donc résolues avant que le produit puisse être démarré
- **Les exceptions** : doivent être anticipées par le développeur et on a des moyens de les gérer
- **Les erreurs non stoppantes** : on peut coder des tests unitaires pour vérifier le bon fonctionnement de notre application. Tests unitaires -> formation de 2j chez Dawan : *Python Intermédiaire : Multithreading et Tests*

Idée : pour toutes les méthodes, automatiser un processus qui va (pour l'exemple de l'addition) :

- prendre des valeurs en entrée : testons pour $a = 10$ et $b = 5$
- faire tourner la méthode : *resultat = addition(a,b)*
- vérifier que le résultat correspond au résultat attendu / plus généralement vérifier que la méthode a le bon comportement avec ces arguments en entrée : *AssertEquals(15, resultat)*

Dans cette partie, nous allons voir comment gérer les exceptions. On utilise un bloc try - except.

```
1/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Input In [66], in <cell line: 1>()
----> 1 1/0
```

```
ZeroDivisionError: division by zero
```

```python
try:
  1 / 0
except:
  print("La division par 0 n'est pas possible")
  # Ce code est exécuté si un exception est levée
  # dans le bloc try

print("On continue")
```

```
La division par 0 n'est pas possible
On continue
```

Il existe beaucoup d'exceptions, qui ont des noms différents selon le problème qui se produit. Plus précisément, il existent plusieurs types d'exceptions. Et il sera plus clair lorsque nous aurons vu le concept d'héritage que toutes les exceptions héritent d'un type d'exception de base, nommé exception.

https://docs.python.org/3/library/exceptions.html

On peut spécifier un comportement selon le type d'exception qui a été levée en le précisant.

```python
nombre = input("Saisir un nombre: ")
try:
  nombre = int(nombre)
  resultat = 5 / nombre
  chaine = "Test"
  char = chaine[10]
except ZeroDivisionError:
  print("La division par 0 n'est pas possible")
except ValueError:
  print("Vous devez saisir un nombre")
except Exception as erreur: # on peut récupérer l'erreur pour la manipuler
  # on capture ici toutes les autres exceptions qui n'ont pas été
  # précédemment capturées
  print("Autre erreur : ", erreur)
```

```
Saisir un nombre: pas un nombre
Vous devez saisir un nombre
```

```
Saisir un nombre: 0
La division par 0 n'est pas possible
```

```
Saisir un nombre: 2
Autre erreur :  string index out of range
```

Le mot clé **else** permet d'exécuter un bloc uniquement si aucune exception n'a été levée dans le bloc **try**, et le mot clé **finally** permet d'exécuter un bloc dans tous les cas.

```python
nombre = input("Saisir un nombre: ")
try:
  nombre = int(nombre)
  resultat = 5 / nombre
  chaine = "Test"
  # char = chaine[10]
  # le mot clé raise permet de soulever soit-même une exception
  raise IndexError("Out of range !")
except ZeroDivisionError:
  print("La division par 0 n'est pas possible")
except ValueError:
  print("Vous devez saisir un nombre")
except Exception as erreur:
  print("Autre erreur : ", erreur)
else: # executé uniquement si aucune exception n'a été levée
  print("Super ! tout s'est bien passé")
finally: # executée dans tous les cas
  print("Peu importe ce qu'il arrive, on passe ici !")

print("Suite du programme...")
```

```
Saisir un nombre: 5
Autre erreur :  Out of range !
Peu importe ce qu'il arrive, on passe ici !
Suite du programme…
```

```python
nombre = input("Saisir un nombre: ")
try:
  nombre = int(nombre)
  resultat = 5 / nombre
  chaine = "Test"
  # char = chaine[10]
  # le mot clé raise permet de soulever soit-même une exception
  # raise IndexError("Out of range !")
except ZeroDivisionError:
  print("La division par 0 n'est pas possible")
except ValueError:
  print("Vous devez saisir un nombre")
```

```
except Exception as erreur:
  print("Autre erreur : ", erreur)
else: # executé uniquement si aucune exception n'a été levée
  print("Super ! tout s'est bien passé")
finally: # executée dans tous les cas
  print("Peu importe ce qu'il arrive, on passe ici !")

print("Suite du programme...")
```

```
Saisir un nombre: 12
Super ! tout s'est bien passé
Peu importe ce qu'il arrive, on passe ici !
Suite du programme…
```

Pourquoi faire un bloc finally ? Quelle est la différence avec la suite de programme qui est à la fin du try - except ? Techniquement, aucune. Comme vous avez pu le voir sur les deux exemples précédents, le bloc finally et la suite du programme s'effectuent dans tous les cas. Le bloc finally a pourtant une utilité dans la gestion de son code.

Par exemple, lorsqu'on va se connecter à une base de données, ou manipuler des fichiers, nous allons ouvrir des flux de connexion, de lecture ou d'écriture. Ce sont ces flux qui nous permettront de faire des opérations, mais ils correspondent à des ressources mémoire allouées qu'il va falloir libérer lorsque nous auront terminé. Il faudra se déconnecter de la base de données, ou fermer les fichiers.

```
try:
  pass
  # ouverture de connexion à une bdd
  # traitement
  # ....
except:
  pass
  # gestion d'erreur
finally:
  pass
  # fermeture de connexion à la bdd

# ...
# ...
# ...
```

En écrivant le code de fermeture du flux dans le bloc finally, cela nous permet de coupler le code d'ouverture et de fermeture de la connexion. Cela permet d'éviter des problèmes futurs de modification de code. Au fur du temps, il est possible qu'on souhaite rajouter du code, le modifier, ou le déplacer et on pourrait par exemple supprimer la fermeture du flux par accident. Ecrit ainsi,

nous avons une structure logique qui se charge de tout le traitement lié au flux, de son ouverture à sa fermeture.

Beaucoup de bonnes pratiques de programmation existent puisqu'elles permettent de répondre à ce type de problématique. Voici quelques problématiques clés.

- Faire en sorte que le code soit lisible et compréhensible par quelqu'un d'autre. Que ce soit quelqu'un d'autre dans notre équipe, mais aussi soit-même dans quelques semaines lorsque nous n'aurons plus la tête dans le code que nous sommes en train de créer. Le futur nous ne sera pas aussi familier avec tous les détails du code que ce que nous somme présentemment
- Minimiser les risque d'erreurs du développeur en instaurant des reflexes, des conventions qui permettent d'éviter certains problèmes sans trop à avoir à y réflechir
- Faciliter la modification et l'extension future du code
- Centraliser les comportements communs. Si la même logique est présente à différents endroits dans notre code, nous perdons du temps à mettre en place ces différents endroits, et il devient très fastidieux de modifier le code. C'est en lien avec le point précédent : une modification du comportement commun signifie une modification de tous les endroits où le code a été répété. Non seulement cela peut être long si il y en a beaucoup, mais on peut facilement oublier de modifier un de ces endroits et avoir de fâcheuses répercussions.

7.1 Exercices

1) Ecrire un programme qui demande à l'utilisateur un nombre entre 1 et 10 et qui affiche le double. Si l'utilisateur n'écrit pas un nombre, lui redemander.

See *Answer*

2) Ecrire un programme qui choisit un nombre aléatoirement entre 0 et 100 et qui demande à l'utilisateur de deviner ce nombre. Après chaque proposition, le programme indique si la valeur est plus grande ou plus petite que la proposition. Le programme gèrera le cas où l'utilisateur n'entre pas un nombre. Lorsque l'utilisateur a trouvé, lui donner son nombre de tentatives. Si l'utilisateur dépasse un certain nombre de coups, il a perdu.

Indication : pour choisir un nombre aléatoire, utiliser :

```
import random # importe le module qui contient toutes
              # les fonctions permettant de gérer
              # l'aléatoire
nombre = random.randint(0,100)  # génère un nombre
                                # aléatoire entre 0 et
                                # 100 selon une loi uniforme
```

See *Answer*

8 Fonctions

Une fonction est une brique de code simple, facile à identifier.

On a déjà utilisé plusieurs fonctions : *print*, *input* sont des fonctions.

- elles portent un nom simple qui indique bien à quoi elles servent
- elles prennent des paramètres qui permettent de varier la manière dont on les utilise
- on n'a pas besoin de savoir comment elles sont écrites, juste ce qu'elles vont faire

Sans fonction, le code devient vite long, et il est difficile de s'y repérer si une certaine foncionnalité est utilisée plusieurs fois, on va être obligé de répéter du code plusieurs fois. On peut à la place factoriser le code et utiliser une fonction.

Pour faciliter la lecture, une bonne pratique est de faire des fonctions relativement courtes c'est-à-dire ~20 lignes, pas beaucoup plus.

```
def ma_fonction(): # signature de la fonction
    print("Je suis la fonction ma_fonction.")  # corps de la
                                               # fonction

# ici, on n'a uniquement déclaré une fonction. On a un ensemble
# d'instructions prêt à être utilisé. A cet endroit du code,
# on n'a pas encore exécuté la fonction, rien ne s'affichera :
# on a seulement chargé en mémoire une fonction qu'on a mis
# dans la variable ma_fonction

# tout est objet en Python. function1 est un objet
# qui est une fonction
print(ma_fonction)
print(type(ma_fonction))

# pour utiliser la fonction, on utilise son
# nom suivi entre parenthèses des différents
# paramètres qu'on souhaite utiliser
# (ici ma_fonction est sans paramètres)
ma_fonction()
```

```
<function ma_fonction at 0x000002DFF2362B80>
<class 'function'>
Je suis la fonction ma_fonction.
```

8.1 Fonctions avec paramètres

Voici un exemple de fonction avec paramètres.

```python
def punition(message, nombre_repetition):
  for i in range(nombre_repetition):
    print(f"Ligne {i+1} : {message}")

message = "Je ne dois pas me répéter"
punition(message, 5)

print("_____")

message = "Je dois me répéter"
punition(message,1)
```

```
Ligne 1 : Je ne dois pas me répéter
Ligne 2 : Je ne dois pas me répéter
Ligne 3 : Je ne dois pas me répéter
Ligne 4 : Je ne dois pas me répéter
Ligne 5 : Je ne dois pas me répéter
_____
Ligne 1 : Je dois me répéter
```

8.2 Fonctions renvoyant une valeur

Une fonction peut renvoyer une valeur (penser à input qui renvoie ce que l'utilisateur a écrit) pour coder cela, on utilise le mot clé **return** VALEUR.

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

print(f"Le carré de 6 est {carre(6)}")

# une fonction qui ne renvoie rien (en apparence)
# renvoie en réalité None
print(ma_fonction())
```

```
Le carré de 6 est 36
Je suis la fonction ma_fonction.
None
```

Voyons par exemple une fonction qui permet de demander à l'utilisateur un nombre et qui renvoie à coup sur un nombre.

```python
def demander_saisie_nombre():
  while True:
    user_input = input("Merci d'entrer un nombre : ")
```

```
    # on peut utiliser un paramètre pour la chaîne
    # de caractères utilisée pour demander le nombre
    # plutôt que le coder en dur. Cela rendra notre fonction
    # plus flexible. C'est une bonne question à se poser et
    # un bon reflexe à avoir lorsqu'on code nos fonctions.
    try:
      result = int(user_input)
      return result
    except ValueError:
      print("Seul les caractères [0-9] sont autorisés")

nombre = demander_saisie_nombre()
print(f"Nombre choisi : {nombre}")
```

```
Merci d'entrer un nombre : pas un nombre
Seul les caractères [0-9] sont autorisés
Merci d'entrer un nombre : 15
Nombre choisi : 15
```

```
def demander_saisie_nombre(saisie):
  while True:
    user_input = input(saisie)
    try:
      result = int(user_input)
      return result
    except ValueError:
      print("Seul les caractères [0-9] sont autorisés")

# on peut utiliser cette fonction dans des contextes différents
# où le message affiché sera plus adapté à la situation

nombre = demander_saisie_nombre("Merci d'entrer un nombre :")
print(f"Nombre choisi : {nombre}")

print("_____")

nombre = demander_saisie_nombre("Quel nombre voulez-vous choisir ? ")
print(f"Nombre choisi : {nombre}")
```

```
Merci d'entrer un nombre :5
Nombre choisi : 5
-----
Quel nombre voulez-vous choisir ? 6
Nombre choisi : 6
```

8.3 Valeurs par défaut

On peut utiliser des valeurs par défaut en les spécifiant dans la signature de la méthode.

Il y a un ordre imposé par l'interpréteur Python pour les arguments : les arguments par défaut doivent être déclarés après les arguments non optionnels.

```
def f(x, alpha=15, beta=16):
  print(x, alpha, beta)

f(4)
f(1,2)
```

```
4 15 16
1 2 16
```

```
def f(x, alpha=15, beta): # alpha est un argument optionnel
  # et doit être placé après les arguments obligatoires
  print(x, alpha, beta)

f(4)
f(1,2)
```

```
  Input In [97]
    def f(x, alpha=15, beta): # alpha est un argument optionnel
                              ^
SyntaxError: non-default argument follows default argument
```

Les arguments passés de cette manière doivent respecter l'ordre dans lequel ils ont été déclarés dans la fonction. Tous les arguments peuvent être aussi passés par leur nom, et dans ce cas on n'a pas besoin de respecter la position :

```
def f(x, alpha=15, beta=16):
  print(x, alpha, beta)

f(beta=23, x=50) # x peut être passé par son nom,
# sans respecter la position
```

```
50 15 23
```

Avec les outils que nous avons pu voir depuis le début de ce cours, on peut écrire une fonction qui permet de demander à l'utilisateur un nombre compris entre deux nombres et de s'assurer qu'il entre bien un nombre correct.

```python
def demander_saisie_nombre_borne(invite = "Merci d'entrer un nombre",minimum =␣
↪1, maximum = 10):
  invite = invite + f" entre {minimum} et {maximum} : "
  while True:
    nombre_input = demander_saisie_nombre(invite)
    # on peut utiliser des fonctions dans le corps de notre fonction
    if minimum <= nombre_input <= maximum:
      return nombre_input

nombre = demander_saisie_nombre_borne()
print(f"Nombre choisi {nombre}")

print("_____")

nombre = demander_saisie_nombre_borne("Entrer un nombre", 0, 100)
print(f"Nombre choisi {nombre}")
```

```
Merci d'entrer un nombre entre 1 et 10 : test
Seul les caractères [0-9] sont autorisés
Merci d'entrer un nombre entre 1 et 10 : 15
Merci d'entrer un nombre entre 1 et 10 : 4
Nombre choisi 4
_____
Entrer un nombre entre 0 et 100 : test
Seul les caractères [0-9] sont autorisés
Entrer un nombre entre 0 et 100 : -9
Entrer un nombre entre 0 et 100 : 3
Nombre choisi 3
```

Cet exemple permet de mettre en avant un autre intérêt des fonctions. Il est important, comme indiqué en début de partie, qu'une fonction soit

- un bout de code simple. Une fonction a la responsabilité de répondre à une seule problématique. Ainsi, lorsqu'on est face à un problème, on peut le décomposer en une série de problème plus simples, qui constituent les différentes étapes de la résolution, et chaque étape correspond à une fonction. Sans avoir fait de fonction, la problématique de demander à l'utilisateur un nombre compris entre deux bornes mélange plusieurs problématiques : le fait de s'assurer que l'utilisateur écrit bien un nombre, et que ce nombre est bien valide. La solution proposée met en avant deux fonctions qui s'assurent chacune de répondre à une seule de ces problématiques. Il est d'autant plus important d'utiliser différentes fonctions à mesure que le problème traité devient complexe
- facile à utiliser. Sans avoir besoin de comprendre comment elle fonctionne, on a juste besoin de savoir ce qu'elle fait. Dans l'exemple précédent, il n'est plus utile de se demander comment résoudre la problématique de demander à l'utilisateur un nombre et de s'assurer qu'il écrive bien un nombre (Est-ce qu'on doit faire une boucle ? Un try except ? Comment mettre tout ça en place ?). On a une fonction qui permet de résoudre ce problème, et on l'utilise sans avoir besoin de savoir les détails de son implémentation

8.4 Fonctions retournant plusieurs valeurs

Une fonction peut renvoyer plusieurs valeurs, en séparant les différentes valeurs par des virgules.

```
def demander_saisie_deux_nombres(invite):
  nombre1 = demander_saisie_nombre(invite+" (premier nombre) ")
  nombre2 = demander_saisie_nombre(invite+" (deuxième nombre) ")
  return nombre1, nombre2

x = demander_saisie_deux_nombres("Entrez des nombres pour tester le type de␣
 ↪retour")
print(x)
print(type(x))
```

```
Entrez des nombres pour tester le type de retour (premier nombre) 5
Entrez des nombres pour tester le type de retour (deuxième nombre) 9
(5, 9)
<class 'tuple'>
```

Une fonction qui renvoie plusieurs valeurs renvoie en réalité un tuple. Nous n'avons pas encore vu ce type, et nous le verrons plus en détails plus tard. Pour l'instant, vous pouvez retenir qu'on peut récupérer les différentes valeurs de la manière suivante :

```
nombre1, nombre2 = demander_saisie_deux_nombres("Entrez deux nombres")
print(f"Premier nombre {nombre1}")
print(f"Deuxième nombre {nombre2}")
```

```
Entrez deux nombres (premier nombre) 11
Entrez deux nombres (deuxième nombre) 222
Premier nombre 11
Deuxième nombre 222
```

8.5 Type hints

Comme vous avez pu le constater, on n'a pas spécifié le type des différents paramètres de nos fonctions. En Python, il n'est pas possible de forcer les paramètres à être d'un certain type, on peut utiliser n'importe quelle variable. Cela peut poser des problèmes.

```
def addition(x,y):
  return x + y

print(f"{addition(4,5) = }")
print(f"{addition('4','5') = }") # on profite ici de la
```

```python
# possibilité d'utiliser " ou ' pour définir des chaînes
# de caractères

def soustraction(x,y):
  return x - y

print(f"{soustraction(4,5) = }")
print(f"{soustraction('4','5') = }") # oups ! Pas d'erreur
# de compilation, mais une exception lors de l'exécution
```

```
addition(4,5) = 9
addition('4','5') = '45'
soustraction(4,5) = -1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [105], in <cell line: 13>()
     10   return x - y
     12 print(f"{soustraction(4,5) = }")
---> 13 print(f"{soustraction('4','5') = }")

Input In [105], in soustraction(x, y)
      9 def soustraction(x,y):
---> 10   return x - y

TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Il faut donc être attentif au type des variables qu'on utilise lors de l'appel d'une fonction pour éviter des erreurs de ce type. Une autre conséquence de ce fait en Python :

```python
# upper est une méthode sur les strings qui retourne la
# chaine en majuscules
# l'IDE nous propose l'autocomplétion, nous donne accès à
# la documentation et le code est coloré correctement
chaine = "truc"
print(chaine.upper())

# dans ce cas, il n'y a pas d'autocomplétion, ni de code
# couleur, et on a pas accès à la doc de la méthode
# upper()
# en effet la variable chaine peut être de n'importe
# quel type, pas forcement une chaine de caractère
# et il n'y a pas de raison que l'IDE nous propose
# de l'autocomplétion ou de la documentation associée
```

```python
# aux chaines de caractères
def exemple(chaine): # chaine est un paramètre, ce n'est
  # pas la variable déclarée plus haut, il s'agit de deux
  # choses différentes, il n'y a pas de conflit
  print(chaine.upper())

exemple(chaine)
exemple("bidule")
```

```
TRUC
TRUC
BIDULE
```

Depuis *Python 3.5*, le langage propose un moyen d'indiquer aux développeurs et à l'IDE le type attendu de variables et d'arguments de fonctions.

Attention cependant : les indications en question n'ont aucune incidence sur l'exécution du code, c'est-à-dire que l'interpréteur les ignore, comme les commentaires. Par contre, les développeurs et les environnements de développement, eux, peuvent se servir de ces indications pour proposer de l'autocomplétion dans certains cas.

```python
def exemple_type_hints(chaine: str) -> None:
  print(chaine.upper)

from typing import Optional # le module typing
# fourni des outils pour faire du type hint
variable: Optional[int] = None
# indique que la variable contiendra `None` ou un entier
```

8.6 Fonctions prenant une collection d'arguments

8.6.1 args

On a croisé la fonction print qui permet d'afficher à la suite plusieurs arguments.

```python
print("Coucou")
print("Coucou","toi")
print("Coucou","ceci","est","un","test","de","la","fonction","print")
```

```
Coucou
Coucou toi
Coucou ceci est un test de la fonction print
```

Comment peut-on coder une fonction qui prend autant d'argument que l'on veut lors de l'appel ? Il existe un type d'argument, qui n'apparaît qu'une seule fois maximum, et toujours après les arguments obligatoires. Il est précédé d'une étoile.

```
#
def f_star(number, *words):
  print(number, words) # words est toujours une tuple
f_star(8)
f_star(15, "Mot 1", "Mot 2")
f_star(-6, "Mot 1", "Mot 2", "Mot 3", "Mot 4", "Mot 5", "Mot 6")
```

```
8 ()
15 ('Mot 1', 'Mot 2')
-6 ('Mot 1', 'Mot 2', 'Mot 3', 'Mot 4', 'Mot 5', 'Mot 6')
```

Nous verrons les tuples plus tard, mais pour l'instant on peut observer qu'on récupère bien toutes les valeurs données en arguments, qu'il n'y a pas un nombre définit à l'avance d'arguments possibles.

8.6.2 kwargs

Un dernier type d'argument, apparaissant aussi une seule fois maximum, et en dernier dans les argument, permet de passer autant d'argument que l'on souhaite par nom.

```
def f_keywords(other=15, **kwargs):
  print(other, kwargs) # toujours un dictionnaire
f_keywords(plop="Hello", foo=19)
# vont dans le dictionnaire `kwargs`.
```

```
15 {'plop': 'Hello', 'foo': 19}
```

Nous verrons les dictionnaires plus tard, mais pour l'instant on peut observer qu'on récupère bien toutes les valeurs données en arguments, qu'elle sont associées aux noms qu'on a spécifié, et qu'il n'y a pas un nombre définit à l'avance d'arguments possibles.

8.7 Exercices

1) Ecrire une fonction qui renvoie le résultat de la multiplication de deux nombres

See *Answer*

2) Ecrire une fonction qui affiche la table de multiplication d'un nombre ou le premier et le dernier multiples sont paramétrés

See *Answer*

3) Ecrire une fonction qui permet de demander à l'utilisateur un nombre et qui renvoie à coup sur un nombre.

See *Answer*

4)

- Coder convertir_heures_en_minutes(heures,minutes) qui renvoie la conversion d'une heure en minutes. Par exemple convertir_heures_en_minutes(1,30) renvoie 90
- Coder convertir_minutes_en_heures(minutes) qui renvoie la conversion de minutes en heures. Par exemple convertir_minutes_en_heures(90) renvoie 1, 30
- reprendre les fonctions demander_saisie_nombre
- coder demander_saisie_nombre_positif(invite) qui renvoie un nombre positif entré par l'utilisateur
- on souhaite afficher un menu

  Menu

  1) Convertir minutes en heures
  2) Convertir heures en minutes
  3) Quitter Votre choix ? Implémenter les fonctionnalités.

See *Answer*

5) Recoder le jeu où l'on devine un nombre choisi aléatoirement par le programme. Cette fois ci, le programme va demander à l'utilisateur quelles bornes utiliser et lorsqu'il aura deviné un nombre incorrect, il lui proposera les choix cohérents. Exemple : je choisi 1 et 100 comme bornes. Le programme me demande de saisir un nombre entre 1 et 100. Je choisi 50. Le programme me répond : le nombre est plus grand, choisi un nombre entre 51 et 100. Lorsque j'ai trouvé, le programme s'arrête. On va cette fois ci utiliser des fonctions.

- reprendre les fonctions demander_saisie_nombre et demander_saisie_nombre_borne
- coder decider_bornes() qui renvoie les bornes utilisées pour le jeu
- coder jouer_un_coup(nombre,minimum,maximum) qui demande à l'utilisateur un entier entre minimum et maximum, et qui renvoie :
  - un boolean à vrai uniquement si l'utilisateur a gagné (il a choisi le paramètre nombre)
  - le minimum pour le coup suivant
  - le maximum pour le coup suivant
- coder jouer_une_partie(nombre,minimum,maximum) qui fait une partie du jeu où l'entier à deviner est le paramètre nombre, et où les bornes sont minimum et maximum et qui renvoie le nombre de tentatives du joueur
- coder une fonction jouer() qui fait tout le jeu : demander à l'utilisateur les bornes du jeu, puis choisi un nombre aléatoirement, puis demande à l'utilisateur de deviner un nombre, en lui disant si le nombre est plus grand ou plus petit, et en lui proposant au fur et à mesure des choix cohérents avec ces indications. Lorsque le joueur trouve le nombre, la fonction le félicite, lui donne son nombre de tentatives et s'arrête.

See *Answer*

9 Modules

Dans les exercices, on a réutilisé des fonctions dans des fichiers différents, et on les a copier/coller. On peut éviter ça en utilisant des modules. C'est ce qu'on a utilisé dans un *exercice* quand on a importé les fonctions qui gèrent l'aléatoire avec *import random*.

9.1 Importer des modules existants

On peut reprendre cet exemple et utiliser des méthodes du module *random*, livré avec Python.

```
import random
nombre = random.randint(1, 10) # génère un nombre
# aléatoire entre 0 et 10 selon une loi uniforme
print(nombre)
```

```
8
```

On a du préfixer l'appel de la fonction *randint* par le nom du module. Cela peut paraître lourd à écrire mais la raison est simple et importante : on utilise l'espace de nom *random* pour éviter les conflits de nommage entre ce que le module fournit et ce que nous avons écrit dans notre code. Nous n'avons pas besoin de nous poser la question suivante : *est-ce qu'une de nos variable s'appelle randint ?* pour éviter les conflits de nommage, et encore pire *quelles sont les noms de toutes les variables que le module random propose à disposition, qu'on puisse éviter de nommer nos variables de la même manière ?*

Cependant, cette méthode peut vous paraître un peu longue à écrire, et un lecteur perspicace pourrait demander *et si j'ai appelé une variable random dans mon fichier ?* On peut modifier le nom de l'espace de nom associé au module de la manière suivante

---

```
import random as r
nombre = r.randint(1, 10)
print(nombre)
```

```
10
```

On peut aussi importer des éléments spécifiques d'un module pour s'abstenir d'utiliser l'espace de nom comme préfixe systématiquement. On peut également les renommer si on le souhaite. En utilisant cette méthode, on a accès dans notre code uniquement à ce qu'on a explicitement importé depuis le module.

```
from random import randrange
from random import uniform as r_uni

```

```
nombre = randrange(0, 101, 5) # nombre aléatoire parmis
# un ensemble de valeurs qui suit les mêmes règles que
# de syntaxe qu'un range, suivant une loi uniforme
print(nombre)

nombre = r_uni(0, 1)
print(nombre)
```

```
100
0.9850293516583172
```

Enfin, on peut également importer tous les éléments d'un module

---

```
from random import *

nombre = randrange(0, 101, 5)
print(nombre)

nombre = r_uni(0, 1)
print(nombre)
```

```
15
0.8469373236780978
```

Néanmoins, on s'expose à tous les risques de conflits de nommage, et cette écriture est largement **déconseillée**.

9.2 Créer ses propres modules

Un module est une collection de fonctions, de classes, de constantes ou autres variables.

Un package est un répertoire qui contient des modules pour transformer un répertoire en un package Python, il suffit qu'il contienne un fichier ___init___.py (_ _ init _ _ .py sans les espaces : double underscore) ce fichier sert à configurer le package et on peut le laisser vide. Le nom du package doit être tout en miniscule sans espace ni underscore

```
|_ app.py
 |_ mypackage
     |_ mymodule.py
     |_  __init__.py (vide)
```

Les modules peuvent êtres des bibliothèques de fonctions, ou peuvent être destinés à être le point d'entrée d'une application… ou les deux.

Dans le fichier mymodule, je peux faire du code Python comme nous l'avons vu, et toutes les variables seront exposées lors de l'import. Avec ce contenu dans *mymodule.py* :

```
def fonction_dans_module():
  print("Ceci est un test, je suis dans mymodule")

fonction_dans_module()
```

```
Ceci est un test, je suis dans mymodule
```

On peut faire le code suivant dans le fichier *app.py*

```
import mypackage.mymodule as mymod

print("Module importé")

mymod.fonction_dans_module()
```

```
Ceci est un test, je suis dans mymodule
Module importé
Ceci est un test, je suis dans mymodule
```

Pourquoi est-il affiché "Ceci est un test, je suis dans mymodule" avant "Module importé" ? On n'a pas fait appel à la fonction fonction_dans_module ! Lorsqu'un module est importé, il est aussi exécuté.

On peut vouloir différencier le comportement selon si il est exécuté en tant que point d'entrée ou si il est exécute lors de son import. On peut s'en sortir puisqu'un module a un nom contenu dans la variable ___name___. Cette variable vaut le nom qu'on lui a donné lorsqu'il est importé, mais dans le cas où c'est le point d'entrée, cette variable vaut ___main___. En modifiant le code de *mymodule.py* :

```
def fonction_dans_module():
  print("Ceci est un test, je suis dans mymodule")

fonction_dans_module()
print(__name__)
```

```
Ceci est un test, je suis dans mymodule
__main__
```

Le code de app.py restant le même

```python
import mypackage.mymodule as mymod

print("Module importé")

mymod.fonction_dans_module()
```

```
Ceci est un test, je suis dans mymodule
mypackage.mymodule
Module importé
Ceci est un test, je suis dans mymodule
```

Pour distinguer le comportement du module selon si il est exécuté en tant que point d'entrée de l'application ou qu'on l'importe, on peut alors utiliser une condition. Avec ce nouveau *module.py*

```python
def fonction_dans_module():
  print("Ceci est un test, je suis dans mymodule")

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

```
Ceci est un test, je suis dans mymodule
```

Le code de app.py restant le même

```python
import mypackage.mymodule as mymod

print("Module importé")

mymod.fonction_dans_module()
```

```
Module importé
Ceci est un test, je suis dans mymodule
```

9.3 Modules externes

9.3.1 Installation et imports de modules externes

Pour installer des modules externes, on va utiliser ce qu'on appelle un environnement virtuel qui permet d'isoler le projet actuel et de n'installer que les dépendances du projet.

- pas de risque de conflits avec d'autres projets qui utiliseraient des dépendances dans d'autres versions
- on ne pollue pas les autres projets sur la machine avec des dépendances inutiles

Pour créer un environnement virtuel : dans une console, au niveau du dossier projet

```
> py -m venv nom-venv
```

- py : execute py.exe
- -m : execute un script
- venv : nom du script qui permet de créer un environnement virtuel
- nom-venv : le nom de l'environnement virtuel à créer. Il peut être ce qu'on veut, souvent on l'appelle venv. Ainsi on fait souvent la commande :

```
> py -m venv venv
```

En cas d'erreur : activer les scripts. Dans un powershell en mode administrateur :

```
> Set-ExecutionPolicy RemoteSigned
> O
```

Une fois l'environnement virtuel installé, il faut ensuite l'activer avec la commande

```
./venv/Scripts/Activate.ps1
```

On voit qu'un environnement virtuel est actif au fait que le terminal commence par *(venv)* où venv est le nom de l'environnement virtuel choisi.

Pour désactiver l'environnement virtuel :

```
deactivate
```

Une fois l'environnement virtuel activé, on peut ensuite utiliser pip pour installer des dépendances externes. https://pypi.org/

pip va se charger d'installer la dépendance, et de gérer les interdépendances. Cela peut se produire si le module que vous voulez installer dépend lui aussi d'autres modules, qu'il faudra alors installer : pip s'occupe d'installer tout le nécessaire pour que le module fonctionne. On utilise la syntaxe

```
> pip install nom-dependance
```

La commande peut également être trouvée sur https://pypi.org/ , où on peut même choisir une version en particulier d'une dépendance.

On peut par exemple installer Pylint avec

```
> pip install pylint
```

Cela permet la vérification du code, ainsi que les recommandations PEP (Python Enhancement Proposal) https://www.python.org/dev/peps/ Depuis VSCode, pour l'activer :

- **ctrl+shift+p** : selectionner le linter -> pylint
- **ctrl+shift+p** : enable linter -> enable

9.3.2 Partage de projet ayant des dépendances externes

La commande

```
> pip list
```

permet de voir toutes les dépendances du projet. Lorsqu'on partage un projet, on ne partage pas le dossier venv. Ce dossier peut avoir une grande quantité de fichiers, et ce n'est pas la source de notre travail. Il est plus rapide de se partager les fichiers concernant notre projet, et de laisser chaque personne avec qui on le partage installer les dépendances.

On partage donc une liste de dépendances avec les noms des dépendances à télécharger et leur versions. On récupère le code, la liste, et on télécharge les dépendances.

Faire ces opérations manuellement serait long et fastidieux. Heureusement, on peut utiliser des commandes pour le faire.

Pour générer un fichier *requirements.txt* avec toutes les dépendances :

```
> pip freeze --local > requirements.txt
```

Pour installer toutes les dépendances listées dans un fichier *requirements.txt* :

```
> pip install -r requirements.txt
```

9.3.3 Path Python

Lorsqu'on fait un import, Python va chercher le package en interne, parmi les packages installés, et aussi dans dossier qui est contenu dans le path Python. Ce dossier est par défaut le dossier du point d'entrée de l'application. Cela veut dire que si vous essayer d'exécuter un fichier Python qui utilise une dépendance qui ne respecte pas la hierarchie des fichiers, où une dépendance fait toujours partie de packages dans le même dossier que le point d'entrée de l'application, la dépendance ne sera pas trouvée.

Avec ce genre d'architecture :

```
|_ monprojet
    |_ mondossier
        |_ app.py
    |_ myotherpackage
        |_ \_\_init\_\_.py
        |_ myothermodule.py
```

Un import du type *import myotherpackage.myothermodule* dans le fichier *app.py* ne fonctionnera pas. Python cherchera la dépendance dans les packages livrés avec Python, dans les packages installés, et dans le répertoire *modossier*, qui sera celui contenu dans le Path Python.

```
import myotherpackage.myothermodule
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In [2], line 1
----> 1 import myotherpackage.myothermodule

ModuleNotFoundError: No module named 'myotherpackage'
```

Il faut alors rajouter le dossier *monprojet* dans le path Python pour qu'il puisse trouver la dépendance.

```
if __name__ == "__main__":
  import sys
  import os
  # __file__ est une variable qui contient le chemin vers le fichier
  # python actuel, ici il s'agira de c:\...\monprojet\mondossier\app.py

  # chemin vers mondossier
  SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
  # chemin vers monprojet
  MAIN_DIR = os.path.dirname(SCRIPT_DIR)
  sys.path.append(MAIN_DIR)

import myotherpackage.myothermodule
```

L'erreur disparaît ainsi et on peut bien importer notre module. Pour récupérer le répertoire à ajouter au Path, nous sommes partis du chemin vers le dossier actuel pour remonter dans la hierarchie des dossiers. Nous avons une méthode relative au fichier actuel : si ce fichier est amené à être déplacé, le code devra être modifié en conséquence.

On peut également faire un chemin relatif au dossier du projet actuel en utilisant la méthode

```
os.getcwd()
```

GET Current Work Directory

```
if __name__ == "__main__":
  import sys
  import os
  sys.path.append(os.getcwd())

import myotherpackage.myothermodule
```

On peut se demander pourquoi est-ce que ce le dossier associé au projet dans son intégralité n'est pas ajouté au path Python par défaut, et une proposition a été faite pour modifier ce comportement de Python https://mail.python.org/pipermail/python-3000/2007-April/006793.html . Guido van Rossum, le créateur de Python, considère que les situations où le besoin de monter plus haut dans la hierarchie des fichiers sont des anti-patterns, qu'elles correspondent à des mauvaises pratiques de programmation. C'est fastidieux pour une bonne raison : si vous avez besoin de le faire, c'est que votre projet n'est pas bien structuré.

10 Listes

10.1 Définition

Les listes permettent de regrouper un ensemble de données cohérents

Liste de prenoms, liste de notes, liste de courses

```
# liste vide
notes = []
print(notes)

notes = [2,6,7,9]
print(notes)
```

Pour définir une liste, il faut mettre les éléments entre crochets [ ] et les séparer par des virgules. Dans la console, la liste apparaît elle aussi sous cette forme. Cette observation est en fait valable pour tous les types de base de Python, et cela nous permet de voir simplement dans la console si une collection d'éléments est une liste, mais aussi un *tuple*, un *dictionnaire*, ...

Il n'y a aucune restriction sur le type des éléments d'une liste. La cohérence fait plutôt référence à une question d'organisation du code. Dans une situation qui semble nécessiter d'utiliser une liste de données non cohérentes, de nature différente, c'est probablement qu'une autre structure est plus adaptée pour le problème (*dictionnaires*, *classes* par exemple).

```
liste = [2,6,7,9, True, "Salut !", [0, 2]]
print(liste)
```

```
[2, 6, 7, 9, True, 'Salut !', [0, 2]]
```

10.2 Atteindre un élément d'une liste

```
notes = [2,6,7,9]

# Les indices commencent toujours à 0
print(f"Première note: { notes[0] }")
print(f"Deuxième note: { notes[1] }")
```

```
Première note: 2
Deuxième note: 6
```

```
notes = [2,6,7,9]
notes[4]  # il y a 4 notes mais la dernière est à
          # l'index 3 -> IndexError: list index
          # out of range
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Input In [89], in <cell line: 2>()
      1 notes = [2,6,7,9]
----> 2 notes[4]

IndexError: list index out of range
```

```python
notes = [2,6,7,9]

taille_notes = len(notes)
print(f"Il y a {taille_notes} notes")

print(f"Dernière note: {notes[taille_notes - 1]}")
# Python autorise les index négatifs, on part de la
# fin de la liste
print(f"Dernière note: {notes[-1]}")
print(f"L'avant dernière note: { notes[-2] }")

notes = [2,6,7,9]
# notes[-5] <- il n'y a pas 5 notes : erreur
```

```
Il y a 4 notes
Dernière note: 9
Dernière note: 9
L'avant dernière note: 7
```

```python
notes = [2,6,7,9]
# on peut modifier une liste
notes[0] = 8
print(notes)
```

```
[8, 6, 7, 9]
```

10.3 Parcourir une liste

10.3.1 Avec un while

```python
notes = [2,6,7,9]
index = 0 # ou parfois juste i
taille_notes = len(notes)
while index < taille_notes:
  note = notes[index]
  print(f"Note {index} : {note}")
  index += 1
```

```
Note 0 : 2
```

```
Note 1 : 6
Note 2 : 7
Note 3 : 9
```

10.3.2 Avec un for

```
notes = [2,6,7,9]
for note in notes:
  print(f"Note : {note}")
```

```
Note : 2
Note : 6
Note : 7
Note : 9
```

10.3.3 Avec un for et l'indice

```
notes = [2,6,7,9]

for i in range(len(notes)):
  print(f"Note {i}: {notes[i]}")
```

```
Note 0: 2
Note 1: 6
Note 2: 7
Note 3: 9
```

Il existe une autre solution qui rend le code encore plus lisible. On peut utiliser la fonction *enumerate*. Nous reviendrons sur son fonctionnement lorsque nous verrons les *tuples*. Pour l'instant, vous pouvez vous contenter de retenir qu'on peut l'utiliser de la manière suivante

---

```
notes = [2,6,7,9]

for index, note in enumerate(notes):
  print(f"Note {index}: {note}")
```

```
Note 0: 2
Note 1: 6
Note 2: 7
Note 3: 9
```

10.4 Manipuler une liste

```
prenoms = ["Valentin", "David", "Anne", "Alicia"]
print(prenoms)

# Ajoute "Akim" à la fin de la liste
prenoms.append("Akim")
print(prenoms)
```

```
['Valentin', 'David', 'Anne', 'Alicia']
['Valentin', 'David', 'Anne', 'Alicia', 'Akim']
```

10.4.1 Mutabilité

Une liste est un objet mutable, c'est à dire que son espace mémoire st modifiable et que l'on peut travailler en son sein. Par conséquent, ses méthodes renvoient None et modifient la liste.

```
prenoms = ["Valentin", "David", "Anne", "Alicia"]
prenoms = prenoms.append("Chloé") # prenoms devient None
print(prenoms)
```

```
None
```

```
prenoms = ["Valentin", "David", "Anne", "Alicia"]
prenoms.append("Chloé") # la liste est modifiée
print(prenoms)
```

```
['Valentin', 'David', 'Anne', 'Alicia', 'Chloé']
```

C'est à opposer au comportement des chaînes de caractères, qui sont des objets non mutables

---

```
chaine = "test"
chaine.upper() # chaine n'est pas modifiée
print(chaine)
```

```
test
```

```
chaine = "test"
chaine = chaine.upper() # upper renvoie une copie
# en majuscule qui est réaffectée à la variable chaine.
print(chaine)
```

```
TEST
```

10.5 Autres opérations

```
prenoms = ["Valentin", "David", "Anne", "Alicia"]
print(prenoms)

# on peut également ajouter un élément à un indice précis
print("insert\tajouter un élément à un indice précis")
prenoms.insert(1,"Marjorie")
print(prenoms)

# supprimer le dernier élément de la liste
print("pop\tsupprimer le dernier élément de la liste")
prenoms.pop()
print(prenoms)

# supprimer un élément à un indice précis
print("pop\tsupprimer un élément à un indice précis")
prenoms.pop(1) # peut soulever une exception
# si l'indice n'est pas valide
print(prenoms)

# inverser la liste par position
print("reverse\tinverser la liste par position")
prenoms.reverse()
print(prenoms)

# trier la liste par ordre alphabétique
print("sort\ttrier la liste par ordre alphabétique")
prenoms.sort()
print(prenoms)

# tester si une valeur appartient à la liste
# pour éviter les erreurs
print("in\ttester si une valeur appartient à la liste")
print("\tpour éviter les erreurs")
if "John" in prenoms:
    prenoms.remove("John")
else:
    print("John n'est pas dans la liste")


if "Valentin" in prenoms:
    index = prenoms.index("Valentin")
    prenoms.pop(index)
else:
    print("Valentin n'est pas dans la liste")
```

```
['Valentin', 'David', 'Anne', 'Alicia']
```

```
insert  ajouter un élément à un indice précis
['Valentin', 'Marjorie', 'David', 'Anne', 'Alicia']
pop     supprimer le dernier élément de la liste
['Valentin', 'Marjorie', 'David', 'Anne']
pop     supprimer un élément à un indice précis
['Valentin', 'David', 'Anne']
reverse inverser la liste par position
['Anne', 'David', 'Valentin']
sort    trier la liste par ordre alphabétique
['Anne', 'David', 'Valentin']
in      tester si une valeur appartient à la liste
        pour éviter les erreurs
John n'est pas dans la liste
```

10.6 Slicing

Le slicing est une syntaxe qui permet de récupérer certains éléments d'une liste, en utilisant une syntaxe proche de celle des *range*.

```python
prenoms = ["Océane", "Chloé", "Marc", "Marjorie"]

liste = prenoms[1:3]  # Les elements de l'index 1
                      # à l'index 3 non inclus
print(liste)

liste = prenoms[0:3]  # Les elements de l'index 0
                      # à l'index 3 non inclus.
print(liste)

liste = prenoms[:3]   # Les elements du début à
                      # l'index 3 non inclus.
print(liste)

liste = prenoms[:4]   # Les elements du début à
                      # l'index 4 non inclus.
print(liste)

liste = prenoms[:]  # Les elements du début
                    # jusqu'à la fin
print(liste)

liste = prenoms[0:4:2]  # Les elements du début à
                        # l'index 4 non inclusavec un
                        # pas (step) 2.
print(liste)
```

```
liste = prenoms[::2]  # Les elements du début
                      # jusqu'à la fin avec un pas
                      # (step) 2
print(liste)

liste = prenoms[::-1]   # Les elements du début
                        # jusqu'à la fin avec un pas
                        # (step) 2
print(liste)
```

```
['Chloé', 'Marc']
['Océane', 'Chloé', 'Marc']
['Océane', 'Chloé', 'Marc']
['Océane', 'Chloé', 'Marc', 'Marjorie']
['Océane', 'Chloé', 'Marc', 'Marjorie']
['Océane', 'Marc']
['Océane', 'Marc']
['Marjorie', 'Marc', 'Chloé', 'Océane']
```

10.7 Listes en compréhension

On peut croiser différents termes : les listes en intension (avec un « s »), « comprehension lists » ou les listes en compréhension.

En Python, on itère beaucoup, c'est à dire qu'on applique très souvent un traitement à tous les éléments d'une séquence, un par un. Et pour ça il y a la boucle for

---

```
sequence = ["a", "b", "c"]
for element in sequence:
  print(element)
```

```
a
b
c
```

Et très souvent, on fait une nouvelle liste avec les éléments de la première liste, mais modifiés

---

```
sequence = ["a", "b", "c"]
new_sequence = []
for element in sequence:
  new_sequence.append(element.upper())

print(new_sequence)
```

```
['A', 'B', 'C']
```

Cette opération – prendre une séquence, modifier les éléments un par un, et faire une autre liste avec – est très commune. Et Python possède une manière élégante de le faire plus vite

---

```
sequence = ["a", "b", "c"]
new_sequence = [element.upper() for element in sequence]
print(new_sequence)
```

```
['A', 'B', 'C']
```

On peut assigner le résultat d'une liste en intension directement à la variable originale

---

```
sequence = ["a", "b", "c"]
sequence = [element.upper() for element in sequence]
print(sequence)
```

```
['A', 'B', 'C']
```

C'est un moyen très propre de transformer toute une liste.

Python est un langage dynamiquement typé, donc on peut transformer le type des éléments de la liste.

```
sequence = [1, 2, 3]
print([str(nombre) for nombre in sequence])
```

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

On peut aussi faire des opérations un peu plus complexes

```
sequence = [1, 2, 3]
print(['a' * nombre for nombre in sequence])
```

```
['a', 'aa', 'aaa']
```

La syntaxe *[expression for element in sequence]* autorise n'importe quelle expression, du coup on peut créer des listes très élaborées, en utilisant tous les opérateurs mathématiques, logiques, etc, et toutes les fonctions que l'on veut.

Une autre opération courante consiste à filtrer la liste plutôt que de la transformer

```
nombres = range(10)
nombres_pairs = []
for nombre in nombres:
  if nombre % 2 == 0: # garder uniquement les nombres pairs
    nombres_pairs.append(nombre)
print(nombres_pairs)
```

```
[0, 2, 4, 6, 8]
```

Python a également une syntaxe plus courte pour cela. Il suffit de rajouter la condition à la fin

```
nombres = range(10)
print([nombre for nombre in nombres if nombre % 2 == 0])
```

```
[0, 2, 4, 6, 8]
```

Toutes les expressions habituellement utilisables pour tester une condition sont également disponibles.

Rien ne vous empêche de filtrer ET de transformer la liste en même temps.

```
nombres = range(10)
print([nombre ** 2 for nombre in nombres if nombre % 2 == 0])
```

```
[0, 4, 16, 36, 64]
```

10.8 Le hasard

Le module random nous permet de choisir un élément ou plusieurs éléments au hasard dans une liste, de la mélanger

```
from random import choice, sample, shuffle
cartes = [x for x in range(1,11)]

print(cartes)

print("choice")
print(choice(cartes))

print("sample")
print(sample(cartes,5))
```

```
print("shuffle")
shuffle(cartes)
print(cartes)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
choice
4
sample
[6, 4, 9, 3, 1]
shuffle
[4, 9, 5, 8, 2, 6, 10, 7, 1, 3]
```

10.9 Exercices

1) Faire un programme qui permet d'ajouter ou retirer des produits d'une liste de courses. Lorsque le programme demarre, l'utilisateur a le choix entre:

   1- Afficher la liste de courses.

   2- Ajouter un produit a la liste de courses (chaine de caracteres ex: pomme)

   3- Retirer un produit de la liste de course

   4- Supprimer toute la liste de courses

   5- Quitter le programme

Tant que l'utilisateur ne saisit pas "quitter", on continue la boucle

Si l'utilisateur tape 1 : on affiche la liste des produits

Si l'utilisateur tape 2 : on lui demande le nom du produit à ajouter et on l'ajoute à la liste de courses

Si l'utilisateur tape 3 : on lui demande le nom du produit à retirer et on le retire de la liste de courses

Si l'utilisateur tape 4 : on vide la liste

Si l'utilisateur tape 5: on affiche "Au revoir" et on quitte le programme

See *Answer*

2) Réduire ces algorithmes à l'aide, des listes en compréhension

a)

```
nombres = [1, 21, 45, 12, 32, 65, 1002, 109, 83]
nombres_pairs = []
for i in nombres:
    if i % 2 == 0:
        nombres_pairs.append(i)
print(nombres_pairs)
```

b)

```
nombres = range(-10, 10)
nombres_positifs = []
for i in nombres:
    if i >= 0:
        nombres_positifs.append(i)
print(nombres_positifs)
```

c)

```
nombres = range(5)
nombres_doubles = []
for i in nombres:
    nombres_doubles.append(i * 2)
print(nombres_doubles)
```

d)

```
nombres = range(10)
nombres_inverses = []
for i in nombres:
    if i % 2 == 0:
        nombres_inverses.append(i)
    else:
        nombres_inverses.append(-i)
print(nombres_inverses)
```

See *Answer*

3) Ecrire une fonction qui transformer une chaine de caracteres en accronyme Salut les geeks -> SLG

See *Answer*

11 Tuples

11.1 Définition

Les tuples sont des collections qui sont très proches et qui se manipulent presque comme des listes. Contrairement aux listes, les tuples ne sont pas modifiables.

Pour définir une liste, il faut mettre les éléments entre parenthèses ( ) et les séparer par des virgules. Dans la console, le tuple apparaît elle aussi sous cette forme.

```
prenoms = ("Vincent", "David", "Alaïs", "Valentin")
print(prenoms)
```

```
('Vincent', 'David', 'Alaïs', 'Valentin')
```

Une grande partie de ce qu'on a vu sur les manipulations de listes s'applique également aux tuples.

```
prenoms = ("Valentin", "David", "Alaïs", "Valentin")

print(prenoms[0])
print(prenoms[2])

print("_____")

for prenom in prenoms:
    print(prenom)

print("_____")

index = prenoms.index("David")
print(f"Index : {index}")

nombre = prenoms.count("Valentin")
print(f'Il y a {nombre} fois le prénom "Valentin" dans la liste')
```

```
Valentin
Alaïs
-----
Valentin
David
Alaïs
Valentin
-----
```

```
Index : 1
Il y a 2 fois le prénom "Valentin" dans la liste
```

Les tuples ne sont pas des sequences modifiables.

```python
prenoms = ("Valentin", "David", "Alaïs", "Valentin")
prenoms.append("Salut") # <- cette fonction n'existe
# pas sur les tuples
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [132], in <cell line: 2>()
      1 prenoms = ("Valentin", "David", "Alaïs", "Valentin")
----> 2 prenoms.append("Salut")

AttributeError: 'tuple' object has no attribute 'append'
```

```python
prenoms = ("Valentin", "David", "Alaïs", "Valentin")
prenoms[0] = "Maude" # TypeError: 'tuple' object does
#  not support item assignment
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [133], in <cell line: 2>()
      1 prenoms = ("Valentin", "David", "Alaïs", "Valentin")
----> 2 prenoms[0] = "Maude"

TypeError: 'tuple' object does not support item assignment
```

11.2 Tuples à un élément

Déclarer un tuple à un seul élément nécessite une syntaxe un peu particulière

---

```python
nombres = (1)
print(f"{nombres} est de type {type(nombres)} ")
```

```
1 est de type <class 'int'>
```

En effet, les parenthèses sont ici vues comme des parenthèses de priorité d'opération, et elles sont alors ignorées. C'est donc l'entier 1 qui est affecté à la variable *nombres*.

Pour déclarer un tuple à un élément, il faut utiliser la syntaxe suivante

---

```
nombres = (1,)
print(f"{nombres} est de type {type(nombres)} ")
```

```
(1,) est de type <class 'tuple'>
```

Dans la console, l'affichage est encore identique à la manière dont on déclare le tuple dans le code.

11.3 Déballage ou unpacking de tuple

11.3.1 Principe

Le déballage (ou unpacking) de tuple est une technique qui permet d'écrire en une seule ligne des instructions de ce type

---

```
prenoms = ("Valentin", "David", "Alaïs")
first_name_1 = prenoms[0]
first_name_2 = prenoms[1]
first_name_3 = prenoms[2]
```

On peut plutôt écrire

```
prenoms = ("Valentin", "David", "Alaïs")
first_name_1, first_name_2, first_name_3 = prenoms

print(f"{prenoms} est de type {type(prenoms)} ")
print(f"{first_name_1} est de type {type(first_name_1)} ")
```

```
('Valentin', 'David', 'Alaïs') est de type <class 'tuple'>
Valentin est de type <class 'str'>
```

On peut utiliser le déballage de tuple pour déclarer plusieurs variables directement en passant par un tuple de manière implicite

---

```
prenom, prenom2 = "toto", "tata"
print(f"{prenom = }")
print(f"{prenom2 = }")
```

```
prenom = 'toto'
prenom2 = 'tata'
```

Si on ne souhaite récupérer que certaines valeurs du tuple, on ne peut pas les négliger

```
prenoms = ("Valentin", "David", "Alaïs", "Valentin", "Jean")
first_name_1, first_name_1, first_name_3 = prenoms
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [156], in <cell line: 2>()
      1 prenoms = ("Valentin", "David", "Alaïs", "Valentin", "Jean")
----> 2 first_name_1, first_name_1, first_name_3 = prenoms

ValueError: too many values to unpack (expected 3)
```

On peut néanmoins y arriver en utilisant une variable préfixée d'une étoile, qui sera une liste qui contiendra toutes les valeurs restantes, qu'on peut convertir en tuple si on le souhaite.

```
prenoms = ("Vincent", "David", "Valentin", "Valentin", "Valentin")
prenom, prenom2, prenom3, *reste = prenoms

print(reste)
print(type(reste))

print("_____")

reste = tuple(reste)
print(reste)
print(type(reste))
```

```
['Valentin', 'Valentin']
<class 'list'>

_____
('Valentin', 'Valentin')
<class 'tuple'>
```

11.3.2 Retour sur des points utilisant le déballage de tuples

Fonctions renvoyant plusieurs valeurs Lorsqu'on a vu comment une fonction peut renvoyer plusieurs valeurs

```python
def demander_saisie_deux_nombres(invite):
  nombre1 = demander_saisie_nombre(invite+" (premier nombre) ")
  nombre2 = demander_saisie_nombre(invite+" (deuxième nombre) ")
  return nombre1, nombre2

x = demander_saisie_deux_nombres("Entrez des nombres pour tester le type de␣
 ↪retour")
print(x)
print(type(x))

nombre1, nombre2 = demander_saisie_deux_nombres("Entrez deux nombres")
print(f"Premier nombre {nombre1}")
print(f"Deuxième nombre {nombre2}")
```

```
Entrez des nombres pour tester le type de retour (premier nombre) 1
Entrez des nombres pour tester le type de retour (deuxième nombre) 22
(1, 22)
<class 'tuple'>
```

Nous avions en réalité fait renvoyer à la fonction un tuple qui contient le différentes valeurs. On pouvait utiliser le déballage de tuples pour récupérer les différentes valeurs retournées par la fonction.

Utilisation de la fonction enumerate pour boucler sur une liste en conservant l'indice
Nous pouvons désormais comprendre ce que la fonction *enumerate* fait d'après sa documentation

---

```python
notes = [2,6,7,9]

for index, note in enumerate(notes):
  # enumerate renvoie une liste de tuples :
  # [ (0, notes[0]), (1, notes[1]), ...]
  # index,nbr vaudra successivement (0, notes[0]) puis (1, notes[1])
  # avec le unpacking de tuples, cela signifie que :
  # index vaudra successivement 0 puis 1 ...
  # nbr vaudra successivement notes[0] puis notes[1] ...
  print(f"Note {index}: {note}")
```

```
Note 0: 2
Note 1: 6
Note 2: 7
Note 3: 9
```

Nous faisons du déballage de tuple dans la déclaration de la variable d'incrémentation de la boucle for.

12 Dictionnaires

12.1 Définition

Un dictionnaire est une sequence non-ordonée qui fonctionne par association cle/valeur

- un dictionnaire (physique) : c'est un ensemble de paires clé/valeur : mot/définition.
- dans un dictionnaire (physique), un mot n'apparaît qu'une seule fois. Les définitions peuvent se répéter.

Un dictionnaire (désormais au sens de la collection en Python) est un ensemble de paires clé/valeur où les clés sont uniques, et pas les valeurs.

Pour définir un dictionnaire, on utilise des accolades. Les différentes paires clé/valeur sont séparées par des virgules, et une paire clé/valeur est définie par des :.

`dictionnaire = {cle:valeur, cle2:valeur}`

On peut accéder à la valeur associée à une clé en utilisant les crochets [ ].

```
dictionnaire = {
  "Python":"language de programmation",
  "Dawan":"organisme de formation",
}

print(dictionnaire["Python"])
print(dictionnaire["Dawan"])
```

```
language de programmation
organisme de formation
```

On peut revenir à la ligne dans la définition des dictionnaires, mettre des espaces, de l'indentation : tout cela sera ignoré par l'interpréteur par la suite. Le retour à la ligne et l'indentation permet d'améliorer la lisibilité. Notez également la différence entre ces deux manières de définir le même dictionnaire :

```
dictionnaire = {
  "Python":"language de programmation",
  "Dawan":"organisme de formation"
}

dictionnaire = {
  "Python":"language de programmation",
  "Dawan":"organisme de formation",
```

```
}
```

Pourquoi mettre cette dernière virgule, n'est-elle pas inutile ? Pourtant, c'est bien la **deuxième écriture** qui est conseillée. La raison étant que si à l'avenir un développeur souhaite rajouter une paire clé/valeur, il n'aura pas à modifier une ligne de code déjà existante mais juste à rajouter une ligne avant l'accolade fermante. Ainsi, une paire clé/valeur à rajouter se traduit par une seule ligne à modifier : on a pas besoin d'aller toucher une ligne qui ne concerne pas ce qu'on souhaite ajouter, vu qu'on ne veut *a priori* pas la modifier. C'est une petite optimisation, mais les bonnes pratiques sont une série d'optimisations, parfois plus importantes que celle là, qui permettent de se simplifier la vie pour l'avenir. Prenez le réflexe de le faire, ça ne demande pas beaucoup d'effort ! L'ensemble de ces reflexes s'accumulent, et permettent en définitive d'améliorer la qualité de votre code, de faciliter sa lecture, la possiblité future de modification, et fera de vous un meilleur développeur.

Un dictionnaire rassemble un ensemble de caractéristiques

```
utilisateur = {
    'nom': "Doe",
    "prenom": "John",
    "age": 15,
}

print(utilisateur['nom'])
```

```
Doe
```

---

avec une liste, on aurait écrit

```
utilisateur = ["Doe", "John", 15]
print(utilisateur[0])
```

```
Doe
```

Ce qui est moins clair, les données de la liste ne sont pas homogènes (pourquoi le nom serait en position 0 ?). Les dictionnaires ont tout de même quelques inconvénients supplémentaires. Le premier, c'est qu'on a pas une structure commune d'un dictionnaire à l'autre.

```
utilisateur = {
    'nom': "Doe",
    "prenom": "John",
    "age": 15,
}
```

```
print(utilisateur['nom'])

autre_utilisateur = {
    'first name': "Curie",
    "last name": "Marie",
    "age": 34,
}

print(autre_utilisateur['first name'])
```

```
Doe
Curie
```

Le deuxième, c'est que cette utilisation dépend de la casse pour les chaînes de caractères, ce qui peut être dérangeant et provoquer des erreurs.

```
dictionnaire = {
  "Python":"language de programmation",
  "Dawan":"organisme de formation",
}

print(dictionnaire["python"])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Input In [177], in <cell line: 6>()
      1 dictionnaire = {
      2   "Python":"language de programmation",
      3   "Dawan":"organisme de formation",
      4 }
----> 6 print(dictionnaire["python"])

KeyError: 'python'
```

Le plus gros problème c'est qu'on ajoute pas réellement de structure, de sémantique à notre code. On pourra le faire avec des *classes*.

On peut éviter d'avoir des problèmes lorsqu'on essaye de récupérer des valeurs associées à une clé dans un dictionnaire en utilisant la méthode *get*, qui renvoie *None* lorsque la clé n'est pas présente dans le dictionnaire.

```
dictionnaire = {
  "Python":"language de programmation",
  "Dawan":"organisme de formation",
}

definition = dictionnaire.get("ornithorynque")
# if definition != None:
if definition:
    print(f"Definition: {definition}")
else:
    print("Ce mot n'existe pas ! ")
```

```
Ce mot n'existe pas !
```

On peut affecter ou modifier des valeurs d'un dictionnaire

```
dictionnaire = {
  "Python":"language de programmation",
  "Dawan":"organisme de formation",
}

# Si la clé n'existe pas, la paire clé/valeur
# sera créée
dictionnaire['Truc'] = "Bidule"
print(dictionnaire)
```

```
{'Python': 'language de programmation', 'Dawan': 'organisme de formation',
'Truc': 'Bidule'}
```

```
dictionnaire = {
  "Python":"language de programmation",
  "Dawan":"organisme de formation",
}

# Si la clé existe, la valeur est modifiée
dictionnaire['Python'] = "Serpent des forêts tropicales d'Afrique et d'Asie"
print(dictionnaire)
```

```
{'Python': "Serpent des forêts tropicales d'Afrique et d'Asie", 'Dawan':
'organisme de formation'}
```

12.2 Types possibles dans un dictionnaire

Les valeurs d'un dictionnaire peuvent être de n'importe qu'elle type. On peut composer les dictionnaires : une valeur peut être elle même un dictionnaire. Les clés peuvent être de n'importe quel type **non mutable** (*str*, *int* ou encore *tuple* par exemple)

```python
utilisateur = {
    'nom': "Doe",
    "prenom": "John",
    "age": 15,
    "telephones": ["0612345678", "0987654321"],
    "adresse": {
        'rue': '1 rue bidon',
        'code_postal': 75015,
    },
}

for numero in utilisateur.get("telephones"):
    print(numero)

print("_____")

rue = utilisateur.get("adresse").get('rue')
print(f"Rue : { rue } ")
```

```
0612345678
0987654321

-----
Rue : 1 rue bidon
```

En interne, l'implémentation utilise un système de hachage pour retrouver rapidement des objets. Or, en Python, tous les objets ne sont pas hachables. Ceux qui ne sont pas hachables ne peuvent pas être ajoutés à un ensemble.

Une fonction de hash est une fonction qui permet de transformer un objet en un hash, un genre de signature qui représente l'objet. https://fr.wikipedia.org/wiki/Fonction_de_hachage De cette manière, en interne, lorsqu'on cherche à récupérer une valeur associée à une clé, elle va être identifiée par son hash et non par sa valeur elle même. C'est plus optimal puisque vérifier par exemple si deux chaînes de cartères sont identiques revient à regarder, caractère par caractère, si les chaînes ont le même contenu. En définitive, les caractères sont stockés en binaire, ce sont des nombres. La machine est donc ramenée à faire des comparaisons entre plusieurs nombres, qui correspondent à chacun des caractères de la chaîne. Quand la chaîne est très longue, le calcul prend un certain temps. Comparer le hash revient à ne comparer qu'un seul nombre : le hash de la clé qu'on cherche avec celle existant dans le dictionnaire avec laquelle on la compare.

Ce mécanisme ne fonctionne que pour les objets mutables. La fonction *hash* permet de récupérer le hash d'un objet, et renvoie une erreur lorsque l'objet n'est pas hachable.

```python
print(hash("Coucou"))
print(hash([1, 2, 3]))
```

-3403310727347999534

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [197], in <cell line: 2>()
      1 print(hash("Coucou"))
----> 2 print(hash([1, 2, 3]))

TypeError: unhashable type: 'list'
```

12.3 Parcourir un dictionnaire

```
dic = {"a":1, "b":2}
print(dic)

for item in dic:
  print(item) # affiche uniquement les clés
```

```
{'a': 1, 'b': 2}
a
b
```

```
dic = {"a":1, "b":2}
print(dic)

for value in dic.values():
  print(value) # affiche uniquement les valeurs
```

```
{'a': 1, 'b': 2}
1
2
```

```
dic = {"a":1, "b":2}
print(dic)
for x in dic.items():
  # x vaut successivement ('a',1) puis ('b',2)
  # autrement dit .items() permet de boucler sur des tuples
  # (clé,valeur)
  print(x)
```

```
{'a': 1, 'b': 2}
('a', 1)
('b', 2)
```

```
dic = {"a":1, "b":2}
print(dic)
for key, value in dic.items():
```

```
  # on peut utiliser plusieurs variables
  # et le déballage de tuples
  print(key, value)
```

```
{'a': 1, 'b': 2}
a 1
b 2
```

12.4 Exercices

1) Transformer la liste de courses. La liste de courses reste une liste mais on y mettra des dictionnaires. Un dictionnaire contient: un nom de produit (str) et une quantité (int)

See *Answer*

2) Black jack simplifié. Créer un dictionnaire où les clés sont les cartes et les valeurs sont leur valeur en nombre de points. Faire piocher une carte au joueur, lui proposer de continuer ou s'arrêter. Lorsqu'il s'arrête, lui donner son score. Si il dépasse les 21, il a perdu et a un score de 0.

See *Answer*

13 Ensembles

14 Définition

Un ensemble est une collection non ordonnée d'éléments sans doublons. De plus, cette collection supporte les opérations sur les ensembles comme l'union, l'intersection, la différence, la différence symétrique.

Pour définir un dictionnaire, on utilise des accolades. On sépare les différents éléments par des virgules.

```
panier = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(panier) # il n'y a pas de doublons
```

```
{'orange', 'banana', 'pear', 'apple'}
```

Attention : pour créer un ensemble vide, il faut utiliser set() et non {} (qui crée un dictionaire vide)

```
x = {}
print(type(x))

x = set()
print(type(x))
```

```
<class 'dict'>
<class 'set'>
```

14.1 Types possibles dans un ensemble

On ne peut mettre dans un ensemble que des éléments non mutables. Les ensembles, comme les *dictionnaires*, utilisent un interne un système de hachage pour optimiser la comparaison des éléments (et éviter les doublons)

```
panier = {'apple', 'orange', 'apple', [1,2,3], 'orange', 'banana'}
print(panier)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [201], in <cell line: 1>()
----> 1 panier = {'apple', 'orange', 'apple', [1,2,3], 'orange', 'banana'}
      2 print(panier)
```

```
TypeError: unhashable type: 'list'
```

14.2 Opérations sur les ensembles

14.2.1 Union

```python
a = set('abracadabra') # conversion de la chaîne de caractères
# en ensemble
b = set('alacazam')

print(f"a = {a}")
print(f"b = {b}")

# lettres dans a ou dans b ou dans les deux
print(a | b)
print(a.union(b))
```

```
a = {'b', 'a', 'r', 'd', 'c'}
b = {'m', 'a', 'l', 'z', 'c'}
{'b', 'a', 'm', 'r', 'l', 'd', 'z', 'c'}
{'b', 'a', 'm', 'r', 'l', 'd', 'z', 'c'}
```

Il est bien précisé dans un ensemble, dans l'autre, ou dans les deux.

En effet, la langue française est ambigüe dans son utilisation du mot ou. Considérez ces deux situations :

- **ou inclusif** : en fin de repas avec des amis, vous leur demandez *vous voulez du fromage ou du dessert ?* Evidément en tant que bon hôte vous accepterez de donner les deux à un bon vivant faisant preuve de gourmandise. Vous utilisez un *ou inclusif* signifiant qu'au moins une des deux possibilités est vraie
- **ou exclusif** : si vous dites à un enfant *tu ranges ta chambre ou tu es puni !* vous n'utilisez pas le même *ou*. Si l'enfant range sa chambre et que vous le punissez quand même, il risque de ne pas être très content. Vous utilisez un *ou exclusif* signifiant qu'une seule des deux possibilités est vraie

Le language humain comprend une dose importante d'implicite, qui ne nous dérange pas puisqu'on comprend le sens de la phrase sans avoir eu besoin qu'on explique les concepts de *ou inclusif*, de *ou exclusif* ou d'autres concepts sous-jacents. Néanmoins, quand on est plus formel, ou qu'on utilise un language de programmation en codant, ces précisions et ces nuances sont importantes puisqu'on se doit d'être complètement explicite.

14.2.2 Intersection

```
# lettres dans a et dans b
print(a & b)
print(a.intersection(b))
```

```
{'a', 'c'}
{'a', 'c'}
```

14.2.3 Différence

```
# lettres dans a mais pas dans b
print(a - b)
print(a.difference(b))
```

```
{'d', 'r', 'b'}
{'d', 'r', 'b'}
```

14.2.4 Différence symétrique

```
# lettres dans a ou dans b mais pas dans les deux
print(a ^ b)
print(a.symmetric_difference(b))
```

```
{'d', 'b', 'm', 'z', 'r', 'l'}
{'d', 'b', 'm', 'z', 'r', 'l'}
```

14.3 Ensembles en compréhension

Les ensembles en compréhension sont supportés. La syntaxe est la même que pour les listes, mis à part qu'on utilise les accolades plutôt que les crochets.

```
a = {x for x in 'abracadabra' if x not in 'abc'}
print(a)
```

```
{'r', 'd'}
```

14.4 Exercices

Ecrire deux ensembles de prénoms Afficher les prénoms communs

See *Answer*

15 Strings

15.1 Méthodes disponibles sur les str

Utilisation des méthodes proposées pour les chaînes de caractères

---

```
texte = "    Ô rage, Ô désespoir, Ô vieillesse ennemie    "

# Méthodes pour la casse
print(texte.upper())  # met en majuscules
print(texte.lower())  # met en minuscules

print("espoir" in texte)  # tester qu'une châine fait
                          # partie de `texte`
print(texte.endswith("ennemie"))  # tester que la chaîne
print(texte.endswith("mie    "))  # se termine par une
                                  # autre

print(texte.find("espoir")) # trouve l'indice de la première
print(texte.find("zzzz"))   # occurence de la sous-chaîne
print(texte.strip())  # enlève les espaces au début et à
                      # la fin, utile

print(texte.count("e"))
```

```
    Ô RAGE, Ô DÉSESPOIR, Ô VIEILLESSE ENNEMIE
    ô rage, ô désespoir, ô vieillesse ennemie
True
False
True
17
-1
Ô rage, Ô désespoir, Ô vieillesse ennemie
8
```

La zone mémoire d'une chaîne de caractères n'est jamais modifiée. Une chaine de caractères est un objet non mutable. Après transformation, la variable pointe vers une zone mémoire différente. La chaine originale ne sera jamais affectée par la fonction.

```
texte = "essai"
texte.replace('e', 'au')
# la str n'est pas modifiée
print(texte)
```

```
texte = "essai"
texte = texte.replace('e', 'au')
# la méthode renvoie une nouvelle str
print(texte)
```

```
essai
aussai
```

On peut briser une chaîne selon une chaîne de caractère avec la méthode *split*, et faire l'opération inverse avec la méthode *join*.

```
texte = "mon.fichier.png"
split_string = texte.split(".")
print(split_string)

result = "////".join(split_string)
print(result)
```

```
['mon', 'fichier', 'png']
mon////fichier////png
```

On ne sera pas exhaustif sur les méthodes sur les *str*, comme on ne l'a pas été sur les listes, les tuples, ou les dictionnaires. On a vu les plus fréquemment utilisées, et avec le temps on peut s'en rappeler. Mais on ne peut pas tout connaître, on ne peut pas attendre d'un développeur qu'il connaisse en connaisse l'intégralité non plus : il ne faut pas hésiter à regarder les méthodes disponibles sur l'objet qu'on souhaite manipuler en regardant ce que l'auto-complétion de l'IDE nous propose. Les noms sont explicites, et la documentation est claire pour comprendre l'utilité de l'étendue de ces méthodes existantes. Il est aussi important et utile de connaître certaines méthodes par coeur que de savoir utiliser ces autres outils pour chercher celles qu'on ne connaît pas.

15.2 Encodage

En Python, en définitive, tout est stocké en binaire sur la machine. Cela signifie que tout est un nombre dans la mémoire. Ce nombre est traité différement selon le type de la variable associé, mais il n'en reste pas moins que ce sont des nombres. Chaque caractère correspond donc à un nombre, correspondant à ce qui est réellement stocké sur la machine. Cette correspondance entre caractère et nombre s'appelle l'encodage : chaque caractère est positionné dans une table et dispose d'une position dans cette table, nommé ordinal. Python utilise l'encodage unicode. https://en.wikipedia.org/wiki/List_of_Unicode_characters Par exemple 'A' a pour ordinal 65 et l'ordinal 97 correspond à la lettre 'a'. Python fournit la méthod *ord* pour récupérer l'ordinal d'un caractère, et la méthode *chr* pour récupérer le caractère correspondant à un ordinal.

```
print(ord("A"))
print(chr(97))
print([chr(x) for x in range(191,264)])
```

```
65
a
['¿', 'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç', 'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î',
'Ï', 'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×', 'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ',
'ß', 'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç', 'è', 'é', 'ê', 'ë', 'ì', 'í', 'î',
'ï', 'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷', 'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ',
'ÿ', 'Ā', 'ā', 'Ă', 'ă', 'Ą', 'ą', 'Ć', 'ć']
```

C'est cet ordinal qui est utilisé pour l'ordre alphabétique. La comparaison des chaînes de caractères utilise l'ordre lexicographique : chaque caractère est comparé successivement. On compare les premiers et si ils sont différents, on s'arrête puisqu'on sait l'ordre. Si ils sont identiques, on passe au caractère suivant. Mais comment sont comparés les caractères ? Ce sont les ordinaux qui sont utilisés. Les majuscules apparaissant avant toutes les minuscules dans la table, le code suivant et son résultat ne devrait pas vous surprendre

---

```
liste = ["arbre", "Python", "chat", "Banane", "programmation"]
print(liste)

liste.sort()
print(liste)
```

```
['arbre', 'Python', 'chat', 'Banane', 'programmation']
['Banane', 'Python', 'arbre', 'chat', 'programmation']
```

15.3 Module string

Le module *string* permet d'accéder à certaines constantes qui peuvent être bien utiles.

```
import string   # ne pas appeler le fichier string.py
                # il risque d'y avoir des conflits de
                # nommage !
print(string.ascii_letters)
print(string.ascii_lowercase)
print(string.ascii_uppercase)
print(string.digits)
print(string.punctuation)

saisie = input("Entrez un caractère : ")
if saisie in string.punctuation:
  print("C'est de la ponctuation")
```

```
else:
  print("Ce n'est pas de la ponctuaction")
```

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
Entrez un caractère : ;
C'est de la ponctuation
```

Attention de ne pas appeler votre fichier avec le même nom qu'un module que vous souhaitez importer. Vous risquez d'avoir une erreur du type :

```
AttributeError: partially initialized module 'string' has no attribute '...'
(most likely due to a circular import)
```

puisque l'import, dans ce cas là de *string*, importe le fichier lui même, qui doit à nouveau importer *string* et s'importe à nouveau lui-même, … C'est un import circulaire. Heureusement, le message d'erreur est assez utile pour comprendre ce qu'il se passe.

15.4 Exercices

1) créer une fonction find_char(chaine, lettre) qui affiche "trouvé" ou "aucun résultat" selon si une lettre apparait dans une chaine de caractères

See *Answer*

2) Créer une fonction annagramme(mot1: str, mot2: str) qui permet de vérifier si deux mots sont des annagrammes

See *Answer*

3) Créer une fonction palindrome(mot, space_ignored = False) qui permet de vérifier si une chaine de caractères est un palindrome.

Le paramètre space_ignored permet d'inclure ou non les espaces

palindrome('kayak') = True

palindrome('Esope reste ici et se repose') = False

palindrome('Esope reste ici et se repose', True) = True

palindrome('chien') = False

See *Answer*

4) Créer une fonction rot(chaine, number) qui encode une chaine de caractères selon le code de César en utilisant le paramètre number.

Code de césar : https://fr.wikipedia.org/wiki/Chiffrement_par_d%C3%A9calage

Attention aux majuscules/minuscules !

Exemple :

rot(“ABCD”,3) = “DEFG”

rot(“Message secret”,13) = “Zrffntr frperg”

Pour décoder un message, il suffit d’utiliser rot(message_encode,26-ancien_number) Et donc :

rot(“DEFG”,23) = “ABCD”

rot(“Zrffntr frperg”,13) = “Message secret”

See *Answer*

5) Créer une fonction decimal_to_binary(decimal) qui converti un nombre décimal en un nombre binaire (renvoie une chaine de caractère) Créer une fonction binary_to_decimal(chaine: str) qui converti un nombre binaire (sous forme de chaine de caractères) en un nombre décimal

See *Answer*

16 Itérables

Fonctions de base de python utilisables sur les itérables (listes, tuples, dictionnaires, set,... n'importe quel objet sur lequel on peut faire une boucle *for*)

```
iterable1 = [0, 1, 0, 1, 0, 1]
iterable2 = [0, 0, 0, 0, 0, 0]
iterable3 = [1, 1, 1, 1, 1, 1]
iterable4 = [0, 1, 2, 3, 4, 5]
iterable5 = ["a", "b", "c", "d", "e", "f"]
# Vérifier que toutes les valeurs sont "vraies"
print("\tall")
print(all(iterable1))
print(all(iterable3))
# Vérifier qu'au moins une valeur est vraie
print("\tany")
print(any(iterable1))
print(any(iterable2))
print(any(iterable4))
# Vérifier la longueur de l'itérable
print("\tlen")
print(len(iterable1))

# Calculer la somme des éléments
print("\tsum")
print(sum(iterable1))
print(sum(iterable4))

# sum(itérable de string)
print("".join(iterable5))

# Trouver la valeur minimale/maximale des éléments
print("\tmin/max")
print(min(iterable4))
print(max(iterable4))
print(min(iterable5))
print(max(iterable5))
```

```
        all
False
True
        any
True
False
True
        len
6
```

```
        sum
3
15
abcdef
        min/max
0
5
a
f
```

17 Dates

Pour créer une date, il faut utiliser le package datetime. On va utiliser trois classes : date, datetime et timedelta, pour gérer les dates et les durées.

- date : dates sans heures
- datetime : dates avec heures (minutes, secondes, microsecondes)
- timedelta : durées, intervalles de temps

17.1 Date

```
from datetime import date

# Créer deux dates
today = date.today()
before = date(2018, 7, 15)  # 15 juillet 2018

print(today)
print(before)
```

```
2022-09-07
2018-07-15
```

On peut afficher la date du jour, mais aussi la formater pour l'afficher comme on le souhaite.

```
before = date(2018, 7, 5)

print(f"Aujourd'hui nous sommes le {before}")
print(f"Aujourd'hui nous sommes le {before.day}/{before.month}/{before.year}")
```

```
Aujourd'hui nous sommes le 2018-07-05
Aujourd'hui nous sommes le 5/7/2018
```

On peut afficher une date en utilisant un formattage plus formalisé. cette méthode nous permet d'éviter de se poser des questions que l'exemple précédent peut soulever : on peut souhaiter afficher les jours et les mois toujours avec deux chiffres, quitte à commencer par un zéro. Au lieu de mettre en place systématiquement cette logique, on peut utiliser des codes de format fournis par Python https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes

On utilise ces codes dans le formattage des fStrings ou avec la méthode *strftime* (STRing Format TIME)

```
before = date(2018, 7, 5)

print(f"Aujourd'hui nous sommes le (formaté) {before:%d/%m/%Y}")
# test = before:%d%m%Y ne fonctionne pas : ce formatage
# est propre au fString
print(before.strftime("%d/%m/%Y"))
```

```
Aujourd'hui nous sommes le (formaté) 05/07/2018
05/07/2018
```

17.2 Datetime

L'utilisation de cette classe est très similaire à son utilisation pour *date*.

```
from datetime import datetime

now = datetime.now()
then = datetime(1966, 12, 19, hour=15)

print(f"{then.year} {then.month} {then.day}")
print(then.date())
print(then.strftime("%d/%m/%Y, %Hh%M"))
```

```
1966 12 19
1966-12-19
19/12/1966, 15h00
```

On peut faire l'opération inverse en utilisant la méthode *strptime* (STRing Parse TIME)

```
# on peut faire le contraire en convertissant une chaine
# de caractère en datetime en spécifiant le format
chaine = "23/12/1989 15h00"
print(datetime.strptime(chaine, '%d/%m/%Y %Hh%M'))
```

```
1989-12-23 15:00:00
```

On peut utiliser la classe *timezone* pour gérer les fuseaux horaires.

```
from datetime import timezone, timedelta

now = datetime.now(timezone.utc)
```

```
before = datetime(2018, 7, 15, 20, 0, tzinfo=timezone.utc)  # 15 juillet 2018 à␣
↪20h
# Afficher la date du jour, et aussi la formater

print(f"Maintenant nous sommes le {now}")
print(f"Maintenant nous sommes le (formaté) {now:%d/%m/%Y %H:%M:%S (%Z)}")

now_other_time_zone = now.astimezone(timezone(timedelta(hours=5)))
print(f"Maintenant nous sommes le (formaté) {now_other_time_zone:%d/%m/%Y %H:%M:
↪%S (%Z)}")
print(f"Fuseau horaire : {now_other_time_zone.tzname()}")
```

```
Maintenant nous sommes le 2022-09-07 08:40:01.344391+00:00
Maintenant nous sommes le (formaté) 07/09/2022 08:40:01 (UTC)
Maintenant nous sommes le (formaté) 07/09/2022 13:40:01 (UTC+05:00)
Fuseau horaire : UTC+05:00
```

Pour en savoir plus sur les fuseaux horaires, ou les heures d'été, d'autres modules existent https://towardsdev.com/giant-mess-dealing-with-timezones-and-daylight-saving-time-in-python-7222d37658cf

---

17.3 Timedelta

La classe *timedelta* permet de gérer des intervalles de temps. On peut les manipuler de manière intuitive en les construisant comme des différence de dates, et en pouvant ajouter un intervalle de temps à une date.

```
now = datetime.now()
then = datetime(1966, 12, 19, hour=15)
interval = timedelta(days=100)
print(interval)
print(now - then) # renvoie un intervalle de temps

print("_____")

now: datetime = datetime.now()
interval: timedelta = timedelta(hours=6)
in_six_hours: datetime = now + interval
print(now)
print(in_six_hours)
```

```
100 days, 0:00:00
20350 days, 19:42:20.753786
-----
```

```
2022-09-07 10:42:20.753786
2022-09-07 16:42:20.753786
```

18 Conversions

On peut convertir un objet dans un type en utilisant la fonction dont le nom est le nom du type. On a vu qu'on pouvait convertir un objet en *str* en utilisant la fonction *str*, mais c'est valable pour tous les types.

```
# Convertir une chaîne en valeur entière
print(int("26"))
print(float("36.5"))
# Convertir une valeur en chaîne
print(str(36.5))
# Convertir une valeur complexe en chaîne
print(str([1, 2, 3, 4, 5]))
```

```
26
36.5
36.5
[1, 2, 3, 4, 5]
```

On a des fonctions qui permettent de faire d'autres conversions, comme *hex* qui convertir un entier en hexadécimal, ou *round* qui arrondi un nombre avec une certaine précision. *L'arrondi se fait à l'inférieur dans le cas où on est à …5*

```
# Convertir un entier en valeur hexadécimale
print(hex(571))

# Conversions mathématiques
print(round(2.8))
print(round(2.5))
print(round(2.45678, 1))
print(round(2.45678, 2))
```

```
0x23b
3
2
2.5
2.46
```

Les conversions booléennes sont particulières et valent le coup d'être retenues.

```
# Les valeurs dites fausses
print("Conversions booléennes qui renvoient faux")
```

```
print(bool(None))
print(bool(0))
print(bool(""))
print(bool(False))
print(bool([]))
# Les valeurs dites vraies
print("Conversions booléennes qui renvoient vrai")
print(bool(1))
print(bool(-1))
print(bool("Bonjour"))
print(bool(True))
print(bool([1, 3, 5, 7, 9]))
```

Les conversions booléennes qui renvoient *False* sont None, 0, une chaîne ou une collection vide. Les autres valeurs sont converties en *True*. Lorsqu'on fait une instruction conditionnelle avec un *if*, une conversion booléenne est faite implicitement. C'est pour cette raison qu'on peut écrire ce genre de choses

---

```
chaine = input("Ecrire quelque chose : ")
# au lieu de
# if chaine != ""
if chaine:
  print("Ok !")
else:
  print("Chaine vide !")
```

```
Ecrire quelque chose : test
Ok !
```

```
chaine = input("Ecrire quelque chose : ")
if chaine:
  print("Ok !")
else:
  print("Chaine vide !")
```

```
Ecrire quelque chose :
Chaine vide !
```

Enfin, on a une série de fonctions

- bool()
- int()
- float()

- str()
- list()
- tuple()
- set()
- dict()

Sans argument, ces fonctions renvoient False, 0, une chaine ou une collection vide : les valeurs par défaut sont les valeurs dont la conversion booléenne renvoie *False*. Avec argument, ces fonctions convertissent la valeur passée en paramètre dans le type en question si c'est possible.

Note : en réalité, il ne s'agit pas de fonctions. Nous verrons lorsque nous aborderons les *classes* qu'il s'agit en réalité des *constructeurs* des classes *bool*, *int*, *float*, ...

19 Stdlib

Voici deux exemples de librairies standard.

- **math** et **statistics** : fournient des outils de base de mathématiques et de statistiques. Ces modules sont fournis avec Python
- **request** : le module de base pour faire des requêtes http. C'est un module externe qu'il faut installer (avec pip)

```python
import math
import statistics

angle = math.pi / 2.0
sample = [1, 5, 30, 80.7, 2100, 2101]

print(f"Le cosinus de {angle} est {math.cos(angle):.1f}")
print(f"\tet le sinus est {math.sin(angle)}")
print(f"L'écart type de {sample} est {statistics.stdev(sample):.2f},")
print(f"\tla médiane est {statistics.median(sample):.2f}")
print(f"\tet la moyenne est {statistics.mean(sample):.2f}")

print("_____")

import requests

r = requests.get('https://github.com')
print(r.content[-50:]) # on n'affiche pas toute la page,
# mais on reconnait bien du contenu html aux balises
```

```
Le cosinus de 1.5707963267948966 est 0.0
        et le sinus est 1.0
L'écart type de [1, 5, 30, 80.7, 2100, 2101] est 1070.00,
        la médiane est 55.35
        et la moyenne est 719.62

_____
b'copy>\n  </div>\n</template>\n\n\n\n\n  </body>\n</html>\n\n'
```

20 Logging

Les logs sont utiles pour conserver un journal de l'utilisation de notre programme. Ils sont l'équivalent d'une boîte noire d'un avion. En cas d'erreur, ils permettent d'aider le débugage en apportant des informations sur ce qui a amené la situation problématique. Ils sont également utiles lors du développement, pour récupérer des informations sur l'état du programme et évaluer la progression de notre code.

Jusqu'à maintenant, on a utilisé la fonction *print* pour afficher des messages dans la console, et pour afficher l'état de certaines variables pour illustrer le comportement de notre code, pour des raisons pédagogiques. Dans un vrai programme, on ne souhaite pas révéler ce genre d'informations à l'utilisateur. C'est vrai non seulement pour des raisons d'expérience utilisateur et d'ergonomie, mais aussi pour des raisons de **sécurité**. Plus on donne d'information à un utilisateur malveillant, moins il aura de mal à poser des problèmes de sécurité. Pourtant, certaines informations sont bien utiles pendant le développement, elles peuvent nous aider à développer plus facilement nos fonctionnalités. Si on se limite à utiliser la fonction *print*, il faudra systématiquement avant la mise en production effacer les différents *print* de notre programme qui dévoilent des informations sensibles. Dans le cas d'un grand programme, ça peut être très long, et il faudra faire la distinction entre les appels de la fonction *print* qui sont pour l'utilisateur et ceux qui sont pour le développeur. Non seulement cette tâche serait longue et répétitive, mais la moindre erreur pourrait nuire à notre programme : qu'il s'agisse de ne pas afficher à l'utilisateur un message qu'il aurait du avoir, ou d'un message qui n'aurait pas du être exposé à l'utilisateur.

En contrepartie, nous allons voir que les logs ont une configuration centralisée, et que les logs se font sur plusieurs niveau. On pourra, en modifiant une ligne dans la configuration, désactiver l'affichage des logs de développement. Les *print* seront réservés à l'affichage dans la console pour l'utilisateur et non pour le développeur.

On peut logger plusieurs types d'informations

- informations de "debug" : c'est pour le développement
- informations générales non problématiques : "info" : pour indiquer une action
- informations de types erreur :
  - "warning" : problèmes mineurs
  - "error" : pour indiquer un problème
  - "critical" : pour indiquer une erreur critique (arrêt de l'application)

```
import logging
import sys

logging.debug("Ceci est mon message - debug")
logging.info("Ceci est mon message - info")
logging.warning("Ceci est mon message - warning")
logging.error("Ceci est mon message - error")
logging.critical("Ceci est mon message - critical")
```

```
WARNING:root:Ceci est mon message - warning
ERROR:root:Ceci est mon message - error
CRITICAL:root:Ceci est mon message - critical
```

Par défaut, le logger n'affiche que les message de niveau *warning* ou supérieur. On peut néanmoins modifier la configuration du logger.

```python
import logging
import sys

logging.basicConfig(
  level=logging.INFO, # niveau minimal pris en compte
  format = "%(asctime)s - [%(levelname)s]: %(filename)s: %(message)s",
  # format du log
  datefmt="%d/%m/%Y - %H:%M", # format de la date dans le log
)

logging.debug("Ceci est mon message - debug")
logging.info("Ceci est mon message - info")
logging.warning("Ceci est mon message - warning")
logging.error("Ceci est mon message - error")
logging.critical("Ceci est mon message - critical")

def addition(nombre1,nombre2):
  logging.info("Appel de la fonction addition commencé")
  return nombre1 + nombre2

resultat = addition(1,2)
print(resultat)
```

```
07/09/2022 - 11:48 - [INFO]: 276448718.py: Ceci est mon message - info
07/09/2022 - 11:48 - [WARNING]: 276448718.py: Ceci est mon message - warning
07/09/2022 - 11:48 - [ERROR]: 276448718.py: Ceci est mon message - error
07/09/2022 - 11:48 - [CRITICAL]: 276448718.py: Ceci est mon message - critical
07/09/2022 - 11:48 - [INFO]: 276448718.py: Appel de la fonction addition
commencé

3
```

Attention : pour configurer le logger, on ne peut le faire qu'avant de l'avoir utilisé. Si on essaye de modifier sa configuration après, rien ne sera pris en compte. De plus, la configuration ne peut être faite qu'une seule fois.

On peut voir un loggeur comme une machine qui permet d'écrire des logs et la classe logging comme une usine qui fabrique des machines. A partir du moment où on utilise le logger, ou qu'on essaye

de le paramétrer, il créé une machine et n'utilisera que celle là : le logger **root**.

On peut, dans cette configuration, changer le comportement du logger root pour qu'il écrive les logs dans un fichier. On peut également modifier le format du log.

```python
import logging
import sys

logging.basicConfig(
  level=logging.INFO, # niveau minimal pris en compte
  format = "%(asctime)s - [%(levelname)s]: %(message)s",
  # format du log
  # https://docs.python.org/3/library/logging.html#logrecord-attributes
  datefmt="%d/%m/%Y - %H:%M", # format de la date dans le log
  filename="app.log",
  filemode="w", # mode d'écriture
  # a pour append (par défaut) : écrit à la suite du fichier
  # w pour write : écrit un nouveau fichier à chaque fois
  encoding="utf-8"
)

logging.debug("Ceci est mon message - debug")
logging.info("Ceci est mon message - info")
logging.warning("Ceci est mon message - warning")
logging.error("Ceci est mon message - error")
logging.critical("Ceci est mon message - critical")

def addition(nombre1,nombre2):
  logging.info("Appel de la fonction addition commencé")
  return nombre1 + nombre2

resultat = addition(1,2)
print(resultat)
```

```
3
```

Contenu du fichier *app.log*

```
07/09/2022 - 11:54 - [INFO]: Ceci est mon message - info
07/09/2022 - 11:54 - [WARNING]: Ceci est mon message - warning
07/09/2022 - 11:54 - [ERROR]: Ceci est mon message - error
07/09/2022 - 11:54 - [CRITICAL]: Ceci est mon message - critical
07/09/2022 - 11:54 - [INFO]: Appel de la fonction addition commencé
```

On peut tout de même créer d'autres loggeurs. Lorsqu'on les créé, leur configuration par défaut

sera celle du logger **root**.

```
mon_loggeur = logging.getLogger("logger_console")

# modification du niveau de mon_loggeur
mon_loggeur.setLevel(logging.DEBUG)
# le loggeur utilise un Handler pour gérer les messages
# on ajoute un StreamHandler qui écrit le message dans un flux
# le flux en question étant sys.stdout, le flux d'écriture vers
# la console
mon_loggeur.addHandler(logging.StreamHandler(sys.stdout))

mon_loggeur.debug("Ceci est mon message de mon loggeur - debug")
mon_loggeur.info("Ceci est mon message de mon loggeur - info")
mon_loggeur.warning("Ceci est mon message de mon loggeur - warning")
mon_loggeur.error("Ceci est mon message de mon loggeur - error")
mon_loggeur.critical("Ceci est mon message de mon loggeur - critical")
```

```
Ceci est mon message de mon loggeur - debug
Ceci est mon message de mon loggeur - info
Ceci est mon message de mon loggeur - warning
Ceci est mon message de mon loggeur - error
Ceci est mon message de mon loggeur - critical
```

Contenu du fichier app.log

```
07/09/2022 - 11:54 - [INFO]: Ceci est mon message - info
07/09/2022 - 11:54 - [WARNING]: Ceci est mon message - warning
07/09/2022 - 11:54 - [ERROR]: Ceci est mon message - error
07/09/2022 - 11:54 - [CRITICAL]: Ceci est mon message - critical
07/09/2022 - 11:54 - [INFO]: Appel de la fonction addition commencé
07/09/2022 - 11:55 - [DEBUG]: Ceci est mon message de mon loggeur - debug
07/09/2022 - 11:55 - [INFO]: Ceci est mon message de mon loggeur - info
07/09/2022 - 11:55 - [WARNING]: Ceci est mon message de mon loggeur - warning
07/09/2022 - 11:55 - [ERROR]: Ceci est mon message de mon loggeur - error
07/09/2022 - 11:55 - [CRITICAL]: Ceci est mon message de mon loggeur - critical
```

21 Documentation

On a vu que la documentation était très utile : nous ne pouvons pas tout connaître sur l'ensemble de tout ce qui est fourni par Python, et dans le cas d'un travail en équipe, on ne peut pas tout connaître dans le détail sur l'ensemble de tout ce qui a été développé par les autres. La documentation permet d'avoir des instructions claires sur ce qui est disponible, comment utiliser les différents outils, leur rôle et le problème auquel ils répondent.

Lorsqu'on développe nos propres outils, il est bon de créer également de la documentation pour son code pour que d'autres développeur (y compris soi-même quelques semaines plus tard) puissent comprendre comment les utiliser.

Pour récupérer la documentation par le code d'un module, d'une classe, d'une fonction ou d'un objet, on utilise *__doc__*

```
import math

print(math.__doc__)
print("_____")
print(math.cos.__doc__)
```

```
This module provides access to the mathematical functions
defined by the C standard.

_____
Return the cosine of x (measured in radians).
```

On peut créer notre propre documentation en utilisant trois “, juste après la définition de ce qu'on souhaite documenter. Cela signifie qu'une documentation en début de fichier documente le module, et juste après la signature d'une fonction documente la fonction par exemple.

```
"""Ce module est une bibliothèque de fonctions très utiles.

Ce module contient des fonctions très utiles.
Ce module s'utilise dans plusieurs cas de figure etc..."""

def ma_fonction():
  """Une fonction à titre d'exemple

On peut documenter nos fonctions comme ceci"""
  print("Je suis dans ma_fonction")


print(__doc__)
```

```
ma_fonction()

print(ma_fonction.__doc__)
```

```
Ce module est une bibliothèque de fonctions très utiles.

Ce module contient des fonctions très utiles.
Ce module s'utilise dans plusieurs cas de figure etc…
Je suis dans ma_fonction
Une fonction à titre d'exemple

On peut documenter nos fonctions comme ceci
```

On peut documenter notre code de cette manière : modules, packages, fonctions, méthodes, classes…

On peut également générer la documentation depuis un terminal.

- Pour générer la documentation dans la console :

```
> py -m pydoc nomdufichier
```

- Pour générer la documentation dans un fichier html :

```
> py -m pydoc -w nomdufichier
```

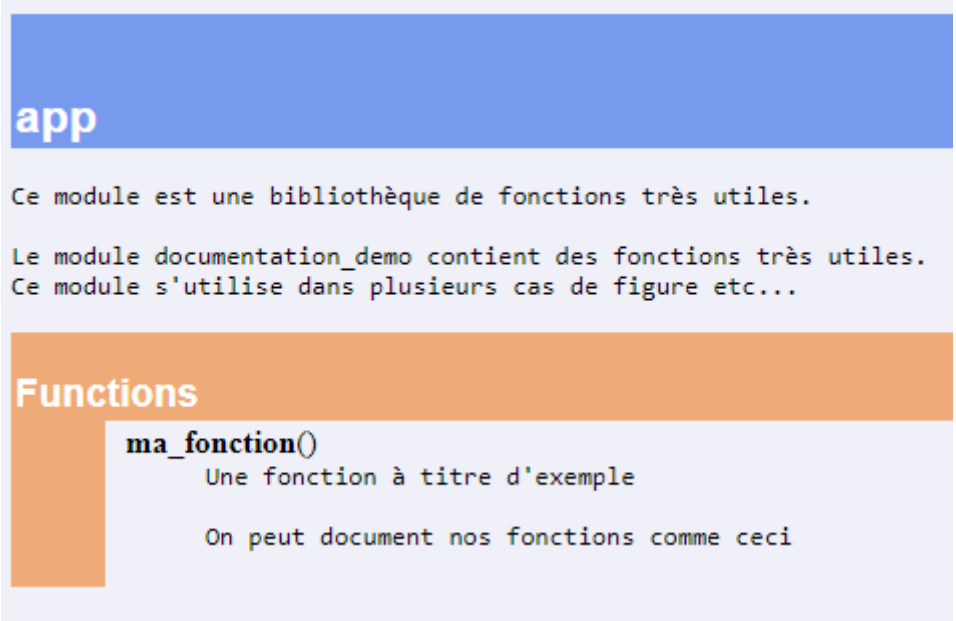
Documentation dans la console de l'exemple précédent :

```
Ce module est une bibliothèque de fonctions très utiles.

Ce module contient des fonctions très utiles.
Ce module s'utilise dans plusieurs cas de figure etc...
Je suis dans ma_fonction
Une fonction à titre d'exemple

On peut documenter nos fonctions comme ceci
Help on module app:

NAME
    app - Ce module est une bibliothèque de fonctions très utiles.

DESCRIPTION
    Ce module contient des fonctions très utiles.
    Ce module s'utilise dans plusieurs cas de figure etc...

FUNCTIONS
    ma_fonction()
        Une fonction à titre d'exemple
```

```
        On peut documenter nos fonctions comme ceci

FILE
    c:\...\app.py
```

Le module est importé par le script, donc il est exécuté, ce qui explique les première lignes du résultat. L'affichage de la documentation par le script *pydoc* commence à la ligne *Help on module app:*

Documentation html de l'exemple précédent :



22 Classes

Une classe est un type d'objet. Un objet a 3 particularités, 3 concepts qu'il va falloir définir dans une classe. La classe n'est qu'un descriptif de l'objet. Lorsqu'on veut parler d'un objet à proprement parler, on parle d'instance d'une classe.

Une analogie :

- La classe : le plan de construction d'un batiment
- Une instance de cette classe : l'objet à proprement parler, un batiment construit suivant le plan

Les 3 concepts à définir dans une classe pour définir un type d'objet :

1) les propriétés/les champs/les attributs qui permettent de décrire l'objet, de donner son état interne. Ces 3 termes existent parce qu'ils ont des sens légèrement différents, mais nous ne rentrerons pas dans de tels détails. Si vous voulez aller plus loin, il faudra se renseigner sur ces différences. Nous utiliserons ces termes comme si ils étaient synonmyes.
2) les méthodes : qui permettent de faire une action avec un objet. Modifier son état interne, ou utiliser son état pour faire un traitement….
3) le constructeur : la méthode appelée pour construire les objets.

En Python, tout est objet !

22.1 Constructeur, attributs

```
# PascalCase: la 1ère lettre de chaque mot est en majuscule
# ex: MaSuperClasse
class Personnage:
  # Python utilise la fonction __init__ pour initialiser les objets
  # la fonction __init__ a un paramètre obligatoire
  # qu'on appelle souvent self (c'est this dans d'autres languages)
  # qui fait référence à l'objet qui est en train d'être
  # instancié
  # On peut utiliser ce qu'on connait sur les fonctions
  # pour rajouter des paramètres, leur donner des valeurs par défaut...
  def __init__(self, nom_param = "John", arme = "Epée", pdv = 100):
    # nom est un attribut de l'instance qu'on créé alors que
    # nom_param est un paramètre de la fonction. Ce sont deux choses
    # différentes.
    self.nom = nom_param
    # Il n'y a donc pas de conflit si on leur donne le même nom :
    # On a d'un côté les attributs arme et pdv de l'instance, et
    # de l'autre les paramètres arme et pdv qui sont des variables
    self.arme = arme
    self.pdv = pdv
```

```
#Personnage.__init__(p1)
p1 = Personnage()
print(p1)
print("Nom du personnage: " + p1.nom )
print("Arme du personnage: " + p1.arme )
print(f"Points de Vie du personnage: {p1.pdv}" )
```

```
<__main__.Personnage object at 0x000001E0EEB8CB50>
Nom du personnage: John
Arme du personnage: Epée
Points de Vie du personnage: 100
```

On peut observer que du point de vue de l'utilisation de la classe personnage, le code ressemble beaucoup à des choses qu'on a utilisé depuis le début du cours. On accède à des propriétés ou des méthodes d'objets depuis le départ sans le savoir. Là, c'est nous-même qui définissons leur comportement via le code de la classe.

Une observation, c'est que l'affichage de nos personnages est assez spéciale. En effet, nous n'avons rien codé pour cela, et par défaut les objets peuvent être convertis en chaîne de caractères et affichés dans la console, mais il apparait

```
<classe at adresse_memoire>
```

Nous verrons plus tard comment modifier ce comportement. On peut instancier d'autres personnages, en spécifiant certaines valeurs dans le constructeur plutôt que de laisser les valeurs par défaut comme lorsqu'on appelle une fonction.

```
# personnage 2
p2 = Personnage()
p2.nom = "Maude"
print("Nom du personnage: " + p2.nom )
print("Arme du personnage: " + p2.arme )
print(f"Points de Vie du personnage: {p2.pdv}" )

print("_____")

#Personnage.__init__(p1, "John", "Epée", 100)
p1 = Personnage("John", "Epée")
print(p1)
print("Nom du personnage: " + p1.nom )
print("Arme du personnage: " + p1.arme )
print(f"Points de Vie du personnage: {p1.pdv}" )

print("_____")
```

```
#Personnage.__init__(p2, "Maude", "Epée lourde", 150)
p2 = Personnage("Maude", "Epée lourde", 150)
print("Nom du personnage: " + p2.nom )
print("Arme du personnage: " + p2.arme )
print(f"Points de Vie du personnage: {p2.pdv}" )
```

```
Nom du personnage: Maude
Arme du personnage: Epée
Points de Vie du personnage: 100

-----
<__main__.Personnage object at 0x000001E0EEA269A0>
Nom du personnage: John
Arme du personnage: Epée
Points de Vie du personnage: 100

-----
Nom du personnage: Maude
Arme du personnage: Epée lourde
Points de Vie du personnage: 150
```

22.2 Attributs et méthodes (d'instances, statiques, de classe)

Pour l'instant, nous avons déclaré des attributs d'instance : chaque objet a sa propre valeur, indépendantes des autres.

Il est également possible de déclarer des attributs statiques (ou de classe, c'est la même chose pour les attributs) qui sont partagés par toutes les instances. Si on souhaite conserver le nombre total de personnage instanciés, nous ne pourrons pas utiliser un attribut d'instance. En effet, un personnage n'est pas notifié lorsqu'un autre personnage est instancié. La classe, par contre, l'est puisqu'on passe par le constructeur.

Nous allons voir également des méthodes d'instance, statiques ou de classe.

```
class Personnage:

  total_personnage = 0 # attribut statique. Sa valeur
  # est initialisée lorsque la classe est chargée en mémoire

  def __init__(self, param_nom = "John", arme = "Epée", pdv = 100):
    self.nom = param_nom
    self.arme = arme
    self.pdv = pdv
    Personnage.total_personnage += 1

  # les méthodes ou les attributs qui commencent et terminent par
  # du _ _ sont appelés soit spéciales soit magiques
  # Lors de la suppression d'un objet, juste avant de libérer
  # l'espace mémoire, le ramasse miette appelle la méthode
```

```
  # __del__()
  def __del__(self):
    Personnage.total_personnage -= 1

  # méthode d'instance : méthode qui effectue un traitement avec
  # une instance de la classe, méthode qui est propre à chaque
  # instance
  # premier argument obligatoire self, qui correspond à l'instance
  # qui utilise la méthode
  def combattre(self, degat = 10):
    self.pdv -= degat
    print(f"Combat terminé, il reste {self.pdv} pdv au personnage {self.nom}")

  # méthode de classe : méthode qui est partagée par toutes les instances
  # dont le comportement ne dépend pas d'une instance en particulier
  # on va utiliser un décorateur :
  # une spécification à la ligne précédent la signature de la méthode
  # les méthodes de classe ont un argument obligatoire, qui correspond
  # à la classe
  @classmethod
  def afficher_total(cls): # cls correspondra à Personnage
    print(f"Il y a {cls.total_personnage} personnages")

  # méthode statique : même principe, mais elles ne nécessitent
  # pas de faire appel à la classe. Autrement dit, l'argument
  # cls est inutile.
  # on a aucun argument obligatoire, et on ne peut pas faire référence
  # à la classe
  @staticmethod
  def dire_bonjour():
    print("Bonjour !")

print(Personnage.total_personnage)

p1 = Personnage() # John
print(Personnage.total_personnage)

p2 = Personnage("Marie","Lance",150)
print(Personnage.total_personnage)

p1 = Personnage("Cyril","Dague",50)
print(Personnage.total_personnage)

# Le personnage John n'est plus référencé est accessible depuis notre
# programme : plus aucune variable n'y fait référence
# Le ramasse-miettes passe régulièrement libérer l'espace mémoire
# occupé par des objets qui ne sont plus référencés par des
```

```
# variables
# John va être supprimé. Il ne reste que Marie et Cyril, et
# donc 2 personnages. Il faut bien décrémenter le compteur
# dans la méthode __del__ appelée par le ramasse-miettes
# juste avant la suppression de l'objet

print("_____")

p1.combattre()

Personnage.afficher_total()
Personnage.dire_bonjour()
```

```
0
1
2
2
_____
Combat terminé, il reste 40 pdv au personnage Cyril
Il y a 2 personnages
Bonjour !
```

22.3 Méthodes spéciales importantes

Il existe plusieurs fonctions spéciales qu'on peut définir dans une classe qui permettent d'avoir certains comportements pour nos objets. Les plus importantes sont :

- la méthode ___str___ permet de définir la conversion en chaîne de caractères d'une instance. Elle ne prend que le paramètre self et doit renvoyer la chaîne de caractère correspondant à la conversion
- la méthode ___eq___ qui permet de définir la comparaison de deux instances

On peut voir l'ensemble de ces méthodes spéciales dans la documentation Python https://docs.python.org/3/reference/datamodel.html nous allons en voir quelques unes.

```
class Personnage:
  total_personnage = 0

  def __init__(self, param_nom = "John", arme = "Epée", pdv = 100):
    self.nom = param_nom
    self.arme = arme
    self.pdv = pdv
    Personnage.total_personnage += 1

  def __del__(self):
    Personnage.total_personnage -= 1
```

```
  # lorsqu'on fait print(instance) ou lorsqu'on souhaite convertir
  # un objet en chaîne de caractère en faisant str(instance),
  # on fait appel à la méthode __str__ de l'instance
  # Par défaut, elle affiche le type et l'adresse mémoire de l'objet
  # on peut la redéfinir pour modifier son comportement
  # cette méthode renvoie le résultat de la conversion
  def __str__(self):
    return f"Le personnage {self.nom} est armé de {self.arme} et a {self.pdv}␣
↪pdv"

  # par défaut, la comparaison de deux objets compare les
  # adresses mémoire et utilise la fonction __eq__
  # elle prend deux arguments (les objets à comparer) et renvoie
  # un booléen correspondant au résultat attendu de ==
  def __eq__(p1,p2):
    return p1.nom == p2.nom

  # lorsqu'on utilise par exemple des méthodes de tri, ou des méthodes
  # qui font de la comparaison, implicitement on fait
  # appel à l'une des méthodes suivantes :
  # __lt__ : lower than <
  # __gt__ : geater than >
  # __leq__ : lower or equal <=
  # __geq__ : greater or equal >=
  # __neq__ : not equal !=
  def __lt__(p1,p2):
    return p1.nom < p2.nom # on compare les noms

  def combattre(self, degat = 10):
    self.pdv -= degat
    print(f"Combat terminé, il reste {self.pdv} pdv au personnage {self.nom}")

  @classmethod
  def afficher_total(cls):
    print(f"Il y a {cls.total_personnage} personnages")

  @staticmethod
  def dire_bonjour():
    print("Bonjour !")


p1 = Personnage()

print(p1)

p2 = Personnage()
```

```
print(p2)

print(p1 == p2)

print("_____")

liste = [
  Personnage(),
  Personnage("Marie","Lance",150),
  Personnage("Cyril","Dague",50)
  ]

if p1 in liste: # on cherche à savoir si le personnage dans la variable
  # p1 appartient à la liste au sens de ==
  # est-ce qu'il existe un élément e de la liste tel que p1 == e ?
  print(f"{p1} est dans la liste")
else:
  print(f"{p1} n'est pas dans la liste")

print("_____")

liste.sort() # vu notre définition de __lt__,
# la liste doit être triée par ordre alphabétique des noms
for p in liste:
  print(p)
```

```
Le personnage John est armé de Epée et a 100 pdv
Le personnage John est armé de Epée et a 100 pdv
True
-----
Le personnage John est armé de Epée et a 100 pdv est dans la liste
-----
Le personnage Cyril est armé de Dague et a 50 pdv
Le personnage John est armé de Epée et a 100 pdv
Le personnage Marie est armé de Lance et a 150 pdv
```

22.4 Autres méthodes spéciales

En Python, on manipule tout le temps des objets, même si on ne le voit pas forcément. On fait toujours appel à un attribut, ou à une méthode d'une instance d'une classe, que ce soit explicitement lorsqu'on fait par exemple

`chaine.upper()`

lorsqu'on souhaite créer une nouvelle chaîne de caractères en majuscules, mais aussi comme on a pu le voir dans des méthodes comme

`liste.sort()`

est implicitement exécuté une méthode de tri sur les éléments, la méthode ___eq___. Un autre

exemple pour renforcer ce point, c'est que des opérations comme l'addition sont également des appels cachés à une méthode d'une instance d'une classe. On peut créer une classe *Period* qui contient une durée, et on pourra additionner une instance de la classe *Period* avec un nombre, pour ajouter un certain nombre de minutes à la durée. *Ce serait un premier petit pas vers la création d'une classe ressemblant à la classe timedelta du module datetime qu'on a vu dans la partie sur les dates*

```python
class Period:
  def __init__(self, heures, minutes):
    self.heures = heures
    self.minutes = minutes

  def __str__(self):
    return f"{self.heures}h{self.minutes}"

  # La methode __add__ prend 1 parametre et renvoie une
  # nouvelle valeur
  def __add__(self, valeur):
    new_periode = Period(0, 0)
    total_minutes = self.minutes + valeur

    # le nombre de minutes peut dépasser les 60
    # auquel cas il faudra ajouter un certain nombre aux heures
    new_periode.heures = self.heures + (total_minutes // 60)
    new_periode.minutes = (total_minutes % 60)
    return new_periode


p1 = Period(2, 45)
print(p1)

p2 = p1 + 30
print(p2)
```

```
2h45
3h15
```

22.5 Pourquoi faire de la programmation orientée objet (POO) ?

Nous allons voir un des intérêts de la programmation orientée objet (POO, *OOP en anglais*)

Partons d'un problème simple : on souhaite pouvoir calculer la surface de rectangles. Voyons d'abord un exemple de la résolution de ce problème qui n'utilise pas de classe, nous pourrons le critiquer et voir les problèmes de ce genre de solution, et nous verrons la résolution du problème en utilisant des classes. Sur un problème simple comme celui là, les problèmes peuvent vous paraître légers, mais il faut bien avoir en tête que ces remarques s'étendent à n'importe quel type de problème, bien plus complexe, et que dans une situation réelle, avec un ensemble de problème à

résoudre, dans un projet de taille conséquente, les remarques qui vont suivre seront d'autant plus valides.

Sans utiliser les classes :

```
def surface(longueur, largeur):
  return longueur * largeur

longueur_1 = 20
largeur_1 = 10

longueur_2 = 25
largeur_2 = 15

s1 = surface(longueur_1, largeur_1)
s2 = surface(longueur_2, largeur_2)

print(f"{s1 = }")
print(f"{s2 = }")
```

```
s1 = 200
s2 = 375
```

Il y a plusieurs problèmes :

- la fonction surface peut être appelée avec n'importe quel nombre, pas forcément des longueurs et des largeurs de rectangles

---

```
nbr_personnages = 10
print(surface(nbr_personnages,nbr_personnages)) # aucun sens, mais possible
print(surface(longueur_1,largeur_2)) # de même, aucun sens, mais possible
```

```
100
300
```

- les variables sont totalement décorrélées. La seule raison qui fait qu'on a compris qu'il s'agissait du calcul de la surface de deux rectangles différents c'est grâce au _1, et _2 dans le nom des variables et au retours à la ligne pour mieux les visualiser
- le code concernant les traitements appliqués aux rectangles n'est pas centralisé : en cas de modification de traitement des rectangles, on doit retenir l'emplacement de toutes les fonctions qui traitent des rectangles, et les modifier toutes
- rien ne nous empêche de mettre des valeurs complètement absurdes

---

```
longueur_3 = -7
largeur_3 = 10
s3 = surface(longueur_3,largeur_3)
print(f"{s3 = }")
```

```
s3 = -70
```

Voyons maintenant l'équivalent en objet :

```
class Rectangle:
  def __init__(self, longueur, largeur):
    self.longueur = longueur
    self.largeur = largeur
  def surface(self):
    return self.longueur * self.largeur
  def __str__(self):
    return f"Rectangle de longueur {self.longueur} et de largeur {self.largeur}"

rec1 = Rectangle(20,10)
rec2 = Rectangle(25,15)

print(rec1)
print(rec1.surface())

print(rec2)
print(rec2.surface())
```

```
Rectangle de longueur 20 et de largeur 10
200
Rectangle de longueur 25 et de largeur 15
375
```

- la fonction surface n'existe que sur les rectangles. Seule une instance de rectangle peut faire appel à la méthode surface
- les longueurs et largeurs sont des attributs d'un rectangle. On ne peut pas se mélanger : on a une instance de rectangle, qui a son attribut longueur et son attribut largeur. La programmation orientée objet permet d'apporter de la structure à notre code, de lui donner un sens. Dans ce code, on ne manipule plus deux entiers, on manipule **des rectangles**.
- le code concernant les traitements appliqués aux rectangles est centralisé : c'est le code de la classe Rectangle. Si on souhaite le modifier, ou ajouter du comportement aux rectangles, on peut le faire en modifiant la classe.
- pour l'instant, rien ne nous empêche encore de mettre des valeurs complètement absurdes

```
rec3 = Rectangle(-10,6)
print(rec3)
print(rec3.surface())
```

```
Rectangle de longueur -10 et de largeur 6
-60
```

Ce dernier problème est temporaire. Nous allons voir le principe d'*encapsulation* qui permettra de résoudre ce dernier problème.

22.6 Exercices

Créer un nouveau package dans le projet myclasses.

Y mettre un fichier product.py qui contient une classe Product qui est décrit par

- un id, *cet id est en anticipation des `bases de données`. Une base de données utilise un identifiant unique pour chacune des données, un code numérique. Pour faciliter notre travail, on souhaite que soient identifiés comme identiques des objets dans notre code aux mêmes conditions que dans la base de données. Comparer des produits dans notre code sera alors comme en base de données : une comparaison d'id.*
- une description,
- un prix,

qui a des méthodes pour calculer la *TVA d'un produit* et le *prix TTC* (prix tva = 0.2 * prix, et prix ttc = prix + prix tva), un *compteur du nombre total de produits instanciés*, une méthode qui permet de *print correctement un produit*, et une méthode qui *compare des produits* (deux produits sont les mêmes ssi ils ont le même id)

See *Answer*

23 POO

23.1 Encapsulation

Pour rajouter du contrôle sur les attributs d'une classe, la bonne pratique est d'utiliser le principe d'encapsulation. Le principe est de protéger les attributs et de ne pouvoir n'y accéder (en lecture : pour récupérer la valeur / en écriture : pour modifier la valeur) qu'en passant par des accesseurs.

On commence par "protéger" un attribut en commençant son nom par ___ (__ _ sans espace) : on le rend pseudo-privé. Python modifie le nom réel de l'atribut, il y a obfuscation de nom à l'extérieur de la classe. Essayer d'accéder à l'attribut à l'exérieur de la classe en écrivant instance.___attribut renvoie une erreur.

On créé ensuite des accesseurs qui vont avoir la responsabilité d'accéder aux variables pseudo-privées

- accesseurs en lecture : GETTERS
- accesseurs en écriture : SETTERS

enfin, on met en place le mécanisme de protection en créant une propriété à l'aide de *property*.

```
class Rectangle:
  def __init__(self, longueur, largeur):
    self.__longueur = longueur
    self.__largeur = largeur

  # les GETTERS ont un argument obligatoire (self) et un seul
  # et renvoie la valeur attendu
  # nom de la méthode : comme on veut
  def __get_largeur(self):
    print("GET LARGEUR")
    return self.__largeur
  # les SETTERS ont deux arguments obligatoires : self, et un
  # argument qui correspond à la nouvelle valeur qu'on essaye
  # d'assigner à l'attribut
  # nom de la méthode : comme on veut
  def __set_largeur(self,valeur):
    print("SET LARGEUR")
    if (valeur <0):
      raise ValueError("La hauteur doit être positive !")
    self.__largeur = valeur

  def __get_longueur(self):
    print("GET LONGUEUR")
    return self.__longueur
  def __set_longueur(self,valeur):
    print("SET LONGUEUR")
```

```
    if (valeur <0):
      raise ValueError("La longueur doit être positive !")
    self.__longueur = valeur

  # on met en place le mécanisme de protection qui utilise
  # les accesseurs qu'on a définit
  longueur_prop = property(__get_longueur,__set_longueur)
  largeur = property(__get_largeur,__set_largeur)

  # L'encapsulation permet de regrouper dans une classe
  # des propriétes ou des comportements de manière à
  # centraliser le code, et à donner la responsabilité
  # à la classe de gérer son comportement
  def surface(self):
    # depuis la classe, on peut accéder aux attributs privés
    # selon le contexte, vous pouvez utiliser soit l'attribut
    # soit la propriété. Ca dépend si vous souhaitez que le contrôle
    # s'applique dans les méthodes
    return self.__longueur * self.__largeur
    # on pourrait donc tout à fait écrire à la place
    # return self.longueur_prop * self.largeur

rec1 = Rectangle(20,10)
rec2 = Rectangle(25,15)


print(rec1)
print(rec1.surface())

print(rec2)
print(rec2.surface())

rec3 = Rectangle(-10,6)
print(rec3)
print(rec3.surface())

rec1.longueur_prop = -6
```

```
<__main__.Rectangle object at 0x00000205D0AA1040>
200
<__main__.Rectangle object at 0x00000205D0AA1E50>
375
<__main__.Rectangle object at 0x00000205D08F8790>
-60
SET LONGUEUR
```

```
ValueError                                Traceback (most recent call last)
Cell In [3], line 63
     60 print(rec3)
     61 print(rec3.surface())
---> 63 rec1.longueur_prop = -6

Cell In [3], line 28, in Rectangle.__set_longueur(self, valeur)
     26 print("SET LONGUEUR")
     27 if (valeur <0):
---> 28   raise ValueError("La longueur doit être positive !")
     29 self.__longueur = valeur

ValueError: La longueur doit être positive !
```

On a bien une exception levée lorsqu'on essaye de modifier un attribut, mais on a toujours un problème quand on passe par le constructeur ! Tel qu'on l'a écrit, le constructeur modifie directement les variables protégées sans passer par les accesseurs. On doit donc légèrement modifier le code.

```python
class Rectangle:
  def __init__(self, longueur, largeur):
    self.__set_longueur(longueur)
    self.__set_largeur(largeur)

  def __get_longueur(self):
    print("GET LONGUEUR")
    return self.__longueur

  def __set_longueur(self, value):
    print("SET LONGUEUR")
    if value < 0:
      raise ValueError("La longueur doit être positive ou nulle")
    self.__longueur = value

  def __get_largeur(self):
    print("GET LARGEUR")
    return self.__largeur

  def __set_largeur(self,value):
    print("SET LARGEUR")
    if value < 0:
      raise ValueError("La largeur doit être positive ou nulle")
    self.__largeur = value

  longueur = property(__get_longueur,__set_longueur)
```

```
  largeur = property(__get_largeur,__set_largeur)

  def surface(self):
    return self.__longueur * self.__largeur
  def __str__(self):
    return f"Rectangle de longueur {self.__longueur} et de largeur {self.
 ↪__largeur}"

rec1 = Rectangle(20,10)
rec2 = Rectangle(25,15)


print(rec1)
print(rec1.surface())

print(rec2)
print(rec2.surface())

rec3 = Rectangle(-10,6)
print(rec3)
print(rec3.surface())
```

```
SET LONGUEUR
SET LARGEUR
SET LONGUEUR
SET LARGEUR
Rectangle de longueur 20 et de largeur 10
200
Rectangle de longueur 25 et de largeur 15
375
SET LONGUEUR
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In [4], line 44
     41 print(rec2)
     42 print(rec2.surface())
---> 44 rec3 = Rectangle(-10,6)
     45 print(rec3)
     46 print(rec3.surface())

Cell In [4], line 3, in Rectangle.__init__(self, longueur, largeur)
      2 def __init__(self, longueur, largeur):
----> 3   self.__set_longueur(longueur)
      4   self.__set_largeur(largeur)

Cell In [4], line 13, in Rectangle.__set_longueur(self, value)
```

```
     11 print("SET LONGUEUR")
     12 if value < 0:
---> 13   raise ValueError("La longueur doit être positive ou nulle")
     14 self.__longueur = value

ValueError: La longueur doit être positive ou nulle
```

On a bien une exception levée si on essaye d'instancier un rectangle avec une longueur ou une largeur négative !

Comme annoncé, l'attribut ___longueur n'est pas accessible depuis l'extérieur de la classe

---

```
print(rec1.__longueur)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [5], line 1
----> 1 print(rec1.__longueur)

AttributeError: 'Rectangle' object has no attribute '__longueur'
```

On peut utiliser ___dict___ qui renvoie un dictionnaire contenant l'ensemble des attributs disponibles sur un objet

---

```
print(rec1.__dict__)
```

```
{'_Rectangle__longueur': 20, '_Rectangle__largeur': 10}
```

On observe que Python n'a fait qu'obfusquer le nom des attributs, en réalité on peut toujours y accéder. Néanmoins l'IDE reconnait qu'il s'agit d'attributs pseudo-privés et il ne nous aide pas : pas d'auto-complétion, pas de code couleur, on ne dirait pas que ça va marcher.

```
print(rec1._Rectangle__longueur) # à ne pas faire
```

```
20
```

On parle d'attribut pseudo-privé parce qu'il est possible d'y accéder de cette manière mais il faut le vouloir. De la même manière qu'une pseudo-constante peut en pratique être modifié, un attribut

pseudo-privé peut être accédé directement comme ci-dessus mais on est pas censé le faire.

On passe par la propriété, qui va elle même faire appel aux accesseurs selon qu'on souhaite y accéder en lecture ou en écriture, et ce sont eux qui vont manipuler l'attribut pseudo-privé.

```
print(rec1.longueur)
```

```
GET LONGUEUR
20
```

```
rec1.longueur = -10
```

```
SET LONGUEUR
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In [9], line 1
----> 1 rec1.longueur = -10

Cell In [4], line 13, in Rectangle.__set_longueur(self, value)
     11 print("SET LONGUEUR")
     12 if value < 0:
---> 13   raise ValueError("La longueur doit être positive ou nulle")
     14 self.__longueur = value

ValueError: La longueur doit être positive ou nulle
```

La bonne pratique lors d'une création de classe est d'avoir les automatismes suivants :

- créer la classe, le constructeur, avec les attributs qu'on souhaite
- créer des getters, des setters et utiliser des propriétés : encapsuler tous les attributs.
- redéfinir les méthodes spéciales ___str___ et ___eq___
- mettre ensuite en place les méthodes de la classe qu'on souhaite

23.1.1 Exercices

Réécrire la classe Product en respectant les bonnes pratiques d'encapsulation des champs dans des accesseurs en lecture et en écriture. Vous pouvez renomer l'ancien fichier product.py en product_naive.py pour garder une trace du premier jet.

See *Answer*

23.2 Héritage

Lorsque plusieurs objets partagent certaines propriétés ou certains comportements, on peut les factoriser dans une classe.

```
class Chat:# on dit que Chat hérite de la classe Animal
  # et que la classe Animal est la classe mère de Chat
  def __init__(self, nom, age):
    self.nom = nom
    self.age = age
  def identite(self):
    print(f"Je suis un chat et je m'appelle {self.nom}")
  def cri(self):
    print("Miaou")
  def __str__(self):
    return f"Nom : {self.nom} - Age : {self.age}"

class Chien:
  def __init__(self, nom, age):
    self.nom = nom
    self.age = age
  def identite(self):
    print(f"Je suis un chien et je m'appelle {self.nom}")
  def cri(self):
    print("Ouaf")
  def __str__(self):
    return f"Nom : {self.nom} - Age : {self.age}"
```

---

on peut factoriser les comportements communs dans une classe Animal

---

```
class Animal:
  def __init__(self, nom, age):
    self.nom = nom
    self.age = age
  def identite(self):
    print(f"Je suis un animal et je m'appelle {self.nom}")
  def __str__(self):
    return f"Nom : {self.nom} - Age : {self.age}"

class Chat(Animal): # on dit que Chat hérite de la classe Animal
  # et que la classe Animal est la classe mère de Chat
  compteur = 0
  def __init__(self, nom, age, couleur):
    super().__init__(nom, age)
    # super() fait référence à la classe mère
```

```
    # ici il fait référence à la classe Animal
    # on peux voir super() comme étant un équivalent
    # de "self vu comme une instance de la classe mère"
    # ici self vu comme un Animal
    self.couleur = couleur
    Chat.compteur += 1
  def __del__(self):
    Chat.compteur -= 1
  def identite(self):
    print(f"Je suis un chat et je m'appelle {self.nom}")
  def cri(self):
    print("Miaou")

class Chien(Animal):
  def identite(self):
    print(f"Je suis un chien et je m'appelle {self.nom}")
  def cri(self):
    print("Ouaf")

c = Chien("Médor",5)
print(c)
c.identite()

cat = Chat("Mistigris",3,"gris")
print(cat)
cat.cri()
```

```
Nom : Médor - Age : 5
Je suis un chien et je m'appelle Médor
Nom : Mistigris - Age : 3
Miaou
```

Quand on écrit

`class MaClasse:`

c'est comme si on écrivait :

`class MaClasse(object):`

autrement dit toutes les classes héritent d'une classe object qui comprend entre autres les comportements par défaut des méthodes ___str___ et ___eq___.

Nous avions vu dans la partie sur les *exceptions* que toutes les classes d'exceptions héritent d'une classe : la classe Exception.

Une bonne pratique est d'implémenter nos propres classes d'exception pour chaque type d'erreur dans notre projet. Ainsi, lorsqu'on fait un try-except, on peut avoir un comportement spécifique

pour chaque erreur. On a non seulement un meilleur contrôle sur les comportements en cas d'erreur, mais on peut aussi mieux débugger dans le cas où une erreur apparaît.

Le **polymorphisme** est l'idée qu'un objet peut avoir plusieurs formes. Dans l'exemple précédent, on peut traiter un chat comme un chat ou comme un animal.

En Python, on peut hériter de plusieurs classes. Néanmoins, cela peut poser des problèmes complexes (Diamond Problem par exemple) https://he-arc.github.io/livre-python/super/index.html

Idée : si on a une class A(B,C) avec B et C qui implémentent la même méthode… laquelle choisir ? A quoi super() fait-il référence ? On pourrait dire *facile, on prend en priorité l'implémentation de la classe qui est écrite en première dans la liste, donc ici B !* mais que se passe-t-il si B hérite de C ?

Ces problèmes ont été résolus par Python en utilisant un algorithme déterminant la priorité, même dans le cas où on a une structure d'héritage complexe. La plupart des langages ont simplement décidé de se passer de l'héritage multiple.

23.2.1 Exercices

Créer deux classes d'exception personnalisées correspondant à la tentative de mettre une valeur négative pour le prix ou l'id d'un produit et adapter le code de la classe Product en conséquence

indications : - rajouter un champ "message" - redéfinir la fonction **str** pour afficher ce message - pour faire appel au constructeur de la classe mère, utiliser super().___init___(params)

See *Answer*

23.3 Interfaces

Une interface est un contrat qui présente des signatures de méthodes. Autrement dit, si une classe implémente une interface, elle se doit d'implémenter toutes les méthodes définies dans l'interface.

Ca permet de voir toutes les méthodes présentes dans une classe, mais aussi de séparer les fonctionnalités de leur implémentation.

Comme Python permet d'hériter de plusieurs classes et qu'il utilise une philosophie de *duck typing* (si ça ressemble à un canard, si ça marche et cancane comme un canard, c'est un canard https://fr.wikipedia.org/wiki/Duck_typing), les interfaces ne sont pas nativement implémentées en Python. On peut utiliser deux solutions :

- Première solution : utiliser une classe mère qui n'implémente pas les méthodes. Un peu comme une classe abstraite (qui est une classe qu'on ne peut pas instancier, mais les classes abstraites n'existent pas non plus nativement en Python)

La convention veut qu'on note une interface par commencant par un I puis en lui donnant un nom en PascalCase

---

```
class ICalcul:
  def add(self,a,b):
    raise NotImplementedError("Method add not implemented")

class Calcul(ICalcul):
  pass

test = Calcul()
print(test.add(3,5))
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
Cell In [19], line 9
      6   pass
      8 test = Calcul()
----> 9 print(test.add(3,5))

Cell In [19], line 3, in ICalcul.add(self, a, b)
      2 def add(self,a,b):
----> 3   raise NotImplementedError("Method add not implemented")

NotImplementedError: Method add not implemented
```

```
class ICalcul:
  def add(self,a,b):
    raise NotImplementedError("Method add not implemented")

class Calcul(ICalcul):
  def add(self,a,b):
    return a+b

test = Calcul()
print(test.add(3,5))
```

```
8
```

- deuxième solution : utiliser le module externe zope qui permet de simuler les comportement d'interfaces. Voir https://zopeinterface.readthedocs.io/en/latest/

```
from zope.interface import Interface, implementer

class ICalcul(Interface):
  def add(a,b):
    pass
```

```
@implementer(ICalcul)
class Calcul:
  pass

test = Calcul()
print(test.add(4,5))
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [28], line 12
      9   pass
     11 test = Calcul()
---> 12 print(test.add(4,5))

AttributeError: 'Calcul' object has no attribute 'add'
```

```
from zope.interface import Interface, implementer

class ICalcul(Interface):
  def add(a,b):
    pass

@implementer(ICalcul)
class Calcul:
  def add(self,a,b):
    return a+b

test = Calcul()
print(test.add(4,5))
```

```
9
```

Il est également possible d'utiliser le module *abc* (abstract base class) pour avoir le comportement d'une classe abstraite. Voir https://docs.python.org/3/library/abc.html

23.4 Agregation

Le concept d'agrégation signifie qu'un attribut d'un objet peut lui-même être un objet. Comme tout est objet en Python, nous avons déjà utilisé ce concept. Mais on peut aller plus loin : un attribut peut être également une instance d'une classe qu'on a nous-même créé.

```
class Adresse:
  def __init__(self, numero = "unknown", rue = "unknown"):
```

```
    self.numero = numero
    self.rue = rue

  # il faudrait faire des GETTERS et des SETTERS

  def __str__(self):
    return f"{self.numero} {self.rue}"

class Salarie:
  def __init__(self, nom, salaire, adresse):
    self.nom = nom
    self.salaire = salaire
    self.adresse = adresse
    # Agrégation : association d'objet, composition.
    # Un objet peut faire parti des attributs d'un
    # autre objet

  # il faudrait faire des GETTERS et des SETTERS

  def __str__(self):
    return f"Salarié {self.nom} a un salaire de {self.salaire} et habite {self.
 ↪adresse}"

a = Adresse("1","rue bidon")
c = Salarie("Pascal",10000, a)
print(c)
```

```
Salarié Pascal a un salaire de 10000 et habite 1 rue bidon
```

23.5 Introspection

L'instrospection permet d'avoir des informations sur une classe à partir d'une instance. Voici quelques exemples

---

```
data = "chaîne de caractères"
# Fonction pour vérifier le type d'une variable
print(isinstance(data, str))
print(issubclass(str, object))
print(issubclass(str, int))
print(type(data))

print("_____")

# Fonction pour accéder aux attributs d'une variable
print(hasattr(data, "manger"))
print(hasattr(data, "upper"))
```

```
print(getattr(data, "upper"))

print("_____")

# Fonctions pour accéder à toutes les variables locales
# ou globales
# print(locals())
# print(globals())

# Fonction qui affiche le texte d'aide pour une
# fonction, méthode etc.
print(help(data.upper))
```

```
True
True
False
<class 'str'>
-----
False
True
<built-in method upper of str object at 0x000001D7AE5B11B0>
-----
Help on built-in function upper:

upper() method of builtins.str instance
    Return a copy of the string converted to uppercase.

None
```

L'introspection est utile si on souhaite accéder à des informations sur une classe par le code.

```
chaine = "Bonjour"
# Une chaîne a toujours une méthode capitalize
if hasattr(chaine, "capitalize"):
  print("L'objet a bien une méthode `capitalize`.")
  print(getattr(chaine, "capitalize"))
# mais une chaîne n'a pas d'attribut doesnotexist.
if hasattr(chaine, "doesnotexist"):
  print("L'objet a un attribut `doesnotexist`.")
else:
  print("L'objet n'a pas d'attribut `doesnotexist`.")
```

```
L'objet a bien une méthode `capitalize`.
<built-in method capitalize of str object at 0x000001D7AE6636F0>
L'objet n'a pas d'attribut `doesnotexist`.
```

On peut utiliser la fonction *dir* qui nous renvoie l'ensemble des attributs et des méthodes disponibles sur un objet

---

```
chaine = "Bonjour"

print(dir(chaine))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

À la place de *dir* on peut utiliser *pdir* qui est un module du package *pdir2*.

Il faut bien faire

```
> pip install pdir2
```

Contrairement à *dir*, il ne fait pas partie de Python, mais *pdir* affiche la liste des attributs et méthodes de l'objet de façon beaucoup plus lisible.

```
import pdir
# En python, tout est un objet, et a donc des attributs
# et des méthodes. Le truc bien, c'est qu'en Python,
# on peut aussi manipuler et retrouver les propriétés
# de ces attributs et méthodes.
chaine = "Bonjour"

print(pdir(chaine))
```

```
special attribute:
    __class__, __doc__
abstract class:
    __subclasshook__
arithmetic:
```

```
    __add__, __mod__, __mul__, __rmod__, __rmul__
object customization:
    __format__, __hash__, __init__, __new__, __repr__, __sizeof__, __str__
rich comparison:
    __eq__, __ge__, __gt__, __le__, __lt__, __ne__
attribute access:
    __delattr__, __dir__, __getattribute__, __setattr__
class customization:
    __init_subclass__
container:
    __contains__, __getitem__, __iter__, __len__
pickle:
    __getnewargs__, __reduce__, __reduce_ex__
static method:
    maketrans: staticmethod(function) -> method  Convert a function to be a
static method.
function:
    capitalize: Return a capitalized version of the string.
    casefold: Return a version of the string suitable for caseless comparisons.
    center: Return a centered string of length width.
    count: S.count(sub[, start[, end]]) -> int  Return the number of non-
overlapping occurrences of substring sub in string S[start:end].
    encode: Encode the string using the codec registered for encoding.
    endswith: S.endswith(suffix[, start[, end]]) -> bool  Return True if S ends
with the specified suffix, False otherwise.
    expandtabs: Return a copy where all tab characters are expanded using
spaces.
    find: S.find(sub[, start[, end]]) -> int  Return the lowest index in S where
substring sub is found, such that sub is contained within S[start:end].
    format: S.format(*args, **kwargs) -> str  Return a formatted version of S,
using substitutions from args and kwargs.
    format_map: S.format_map(mapping) -> str  Return a formatted version of S,
using substitutions from mapping.
    index: S.index(sub[, start[, end]]) -> int  Return the lowest index in S
where substring sub is found, such that sub is contained within S[start:end].
    isalnum: Return True if the string is an alpha-numeric string, False
otherwise.
    isalpha: Return True if the string is an alphabetic string, False otherwise.
    isascii: Return True if all characters in the string are ASCII, False
otherwise.
    isdecimal: Return True if the string is a decimal string, False otherwise.
    isdigit: Return True if the string is a digit string, False otherwise.
    isidentifier: Return True if the string is a valid Python identifier, False
otherwise.
    islower: Return True if the string is a lowercase string, False otherwise.
    isnumeric: Return True if the string is a numeric string, False otherwise.
    isprintable: Return True if the string is printable, False otherwise.
    isspace: Return True if the string is a whitespace string, False otherwise.
```

```
    istitle: Return True if the string is a title-cased string, False otherwise.
    isupper: Return True if the string is an uppercase string, False otherwise.
    join: Concatenate any number of strings.
    ljust: Return a left-justified string of length width.
    lower: Return a copy of the string converted to lowercase.
    lstrip: Return a copy of the string with leading whitespace removed.
    partition: Partition the string into three parts using the given separator.
    removeprefix: Return a str with the given prefix string removed if present.
    removesuffix: Return a str with the given suffix string removed if present.
    replace: Return a copy with all occurrences of substring old replaced by
new.
    rfind: S.rfind(sub[, start[, end]]) -> int  Return the highest index in S
where substring sub is found, such that sub is contained within S[start:end].
    rindex: S.rindex(sub[, start[, end]]) -> int  Return the highest index in S
where substring sub is found, such that sub is contained within S[start:end].
    rjust: Return a right-justified string of length width.
    rpartition: Partition the string into three parts using the given separator.
    rsplit: Return a list of the words in the string, using sep as the delimiter
string.
    rstrip: Return a copy of the string with trailing whitespace removed.
    split: Return a list of the words in the string, using sep as the delimiter
string.
    splitlines: Return a list of the lines in the string, breaking at line
boundaries.
    startswith: S.startswith(prefix[, start[, end]]) -> bool  Return True if S
starts with the specified prefix, False otherwise.
    strip: Return a copy of the string with leading and trailing whitespace
removed.
    swapcase: Convert uppercase characters to lowercase and lowercase characters
to uppercase.
    title: Return a version of the string where each word is titlecased.
    translate: Replace each character in the string using the given translation
table.
    upper: Return a copy of the string converted to uppercase.
    zfill: Pad a numeric string with zeros on the left, to fill a field of the
given width.
```

24 Files

24.1 Files txt

24.1.1 Lecture de fichier

Nous allons dans un premier temps lire un fichier texte créé à la main. Il va falloir récupérer le chemin vers le fichier qu'on souhaite ouvrir. On peut utiliser plusieurs méthodes pour le faire :

- donner le chemin directement,
- donner un chemin relatif au fichier actuel en utilisant les méthodes *abspath*, *dirname* et *join* de *os.path*,
- donner un chemin relatif au projet en utilisant les méthodes *join* de *os.path* et *getcwd* de *os*.

```
import os

# C:/.../dossierprojet
chemin_dossier = os.getcwd()
# C:/.../dossierprojet/myfile.txt
chemin_fichier = os.path.join(chemin_dossier, 'myfile.txt')

# Pour ouvrir un fichier on utilise la fonction open()
# Cette fonction prend plusieurs paramètres dont :
#   - le chemin vers le fichier
#   - le mode de lecture
#     * "r" pour read : lecture
#     * "w" pour write : écriture (fichier écrasé)
#     * "a" pour append : écriture (on écrit à la fin du fichier)
#   - l'encodage
# En mode ecriture: si le fichier n'existe pas il sera
# créé
# /!\ en mode 'w' le fichier est effacé à l'ouverture
fichier = open(chemin_fichier, 'r', encoding="utf-8")
contenu = fichier.read()
print(contenu)

# il faut bien fermer la ressource à la fin du traitement
# pour éviter les fuites mémoire
fichier.close()
```

```
Coucou !

Ceci est le contenu de myfile.txt
```

Plutôt que de manuellement fermer le flux après le traitement avec la méthode *close*, on peut faire appel au *Context Manager* qui se charge de le faire en fin de bloc

```python
# Context Manager: a la fin du bloc d'instructions, le
# context manager s'occupe de liberer les ressources
# allouées à l'ouverture du fichier.
with open(chemin_fichier, 'r', encoding="utf-8") as fichier:
  print(fichier.read()) # on a une tête de lecture qui a l'ouverture
  # du fichier est situé au départ. Lors de la lecture, le curseur
  # avance dans le fichier
  # Après le read(), le curseur est situé en fin de fichier,
  # et du coup il n'y a plus rien à lire
  print("_____")
  print(fichier.read())
  print("_____")
  fichier.seek(0) # positionne le curseur en position 0
  print(fichier.read())
```

```
Coucou !

Ceci est le contenu de myfile.txt
-----

-----
Coucou !

Ceci est le contenu de myfile.txt
```

On peut également récupérer toutes les lignes sous forme de liste

---

```python
fichier = open(chemin_fichier, 'r', encoding="utf-8")
contenu = fichier.readlines() # list
print(contenu)
fichier.close()
```

```
['Coucou !\n', '\n', 'Ceci est le contenu de myfile.txt']
```

On peut remarquer que les sauts de ligne sont conservés dans la liste récupérée.

24.1.2 Ecriture de fichier

Pour écrire dans un fichier, il suffit de l'ouvrir en mode d'écriture ou en mode append, et d'utiliser la méthode *write*

```
chemin_dossier = os.getcwd()
chemin_fichier = os.path.join(chemin_dossier, 'message.txt')
with open(chemin_fichier, 'w', encoding="utf-8") as fichier:
  fichier.write('Salut tout le monde !')
```

Contenu de message.txt :

```
Salut tout le monde !
```

24.1.3 Gestion des répertoires

Un point important que nous n'avons pas abordé, c'est qu'il faut bien s'assurer que les repertoires et/ou les fichiers qu'on souhaite manipuler existent. Le cas échéant, selon la situation on peut les créer, envoyer une erreur…

On a pour cela la méthode *exists* de *os.path* qui nous permet de vérifier l'existence d'un chemin sur la machine, et la méthode *os.mkdir* qui permet de le créer.

Attention : essayer de créer avec *mkdir* un repertoire qui existe déjà provoquera une exception

---

```
chemin_dossier = os.path.join(
    os.getcwd(),
    "mesfichiers"
)

if not os.path.exists(chemin_dossier):
    os.mkdir(chemin_dossier) # creation du dossier
else:
    print("Le dossier existe déjà")
```

Le code précédent a créé un répertoire "mesfichiers" dans le dossier du projet.

Si on souhaite créer pas seulement un répertoire, mais plusieurs, mkdir ne suffit pas : il ne permet de créer que le dernier repertoire cité dans le chemin.

```
chemin_dossier = os.path.join(
    os.getcwd(),
    "mesautresfichiers",
    "unautredossier",
    "encoreunautredossier"
)

if not os.path.exists(chemin_dossier):
    os.mkdir(chemin_dossier)
```

```
else:
    print("Le dossier existe déjà")
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In [52], line 9
      1 chemin_dossier = os.path.join(
      2     os.getcwd(),
      3     "mesautresfichiers",
      4     "unautredossier",
      5     "encoreunautredossier"
      6 )
      8 if not os.path.exists(chemin_dossier):
----> 9     os.mkdir(chemin_dossier)
     10 else:
     11     print("Le dossier existe déjà")

FileNotFoundError: [WinError 3] Le chemin d'accès spécifié est introuvable: 'C:
 ↪\\Users\\Admin stagiaire.
 ↪DESKTOP-8967908\\Desktop\\MesFormations\\FormationPythonInitiationApprofondissement\\NoteBo
```

Si vous souhaitez en créer plusieurs à la fois, vous pouvez utiliser *makedirs*.

```
chemin_dossier = os.path.join(
    os.getcwd(),
    "mesautresfichiers",
    "unautredossier",
    "encoreunautredossier"
)

if not os.path.exists(chemin_dossier):
    os.makedirs(chemin_dossier) # creation des
                                # intermediaires s'ils
                                # n'existent pas
else:
    print("Le dossier existe déjà")
```

24.1.4 Exercices

1) Créer une méthode read_file_txt(path) qui renvoie un string contenant le contenu d'un fichier au chemin path (à partir du dossier de travail actuel) la mettre dans le module mytools, dans files.py

See *Answer*

2) Créer une méthode write_file_txt(path, content) qui écrit le contenu du string content dans un fichier au chemin path (le créant / l'écrasant éventuellement) la mettre dans le module mytools, dans files.py

See *Answer*

Notes :

- ne pas gérer les cas où les fichiers n'existent pas
- ne pas s'embêter avec join, faire

```
file_path = os.getcwd() + path
```

Par exemple, pour récupérer le contenu du fichier message, on doit faire

```
contenu = read_file_txt("/path/to/message.txt")
```

24.2 Files json

24.2.1 Lecture de fichier

Le *JSON* est un format de fichier permettant de structurer des données en suivant une syntaxe bien précise. https://fr.wikipedia.org/wiki/JavaScript_Object_Notation L'important, c'est qu'écrire un fichier json, ce n'est que formatter une chaîne de caractère, et lire un fichier json n'est qu'intepréter correctement le contenu du json.

Il existe un module livré avec Python, json, qui se charge précisément de faire ces conversions.

Pour cet exemple, on peut récupérer un json fictif sur https://jsonplaceholder.typicode.com/users et mettre le contenu dans un fichier *users.json*.

```python
import json
import os

chemin_dossier = os.getcwd()
chemin_fichier = os.path.join(chemin_dossier, 'users.json')

with open(chemin_fichier, 'r', encoding='utf-8') as fichier:
  content = json.load(fichier)
  print(f"type: { type(content) }")
  print(f"type d'un élément : { type(content[0])}")

print("_____")

for utilisateur in content:
  # print(utilisateur) #dict
  # print( type(utilisateur) )
  print("Nom utilisateur: " + utilisateur.get("name") )
  print("\tRue: " + utilisateur.get("address").get('street') )
```

```
type: <class 'list'>
type d'un élément : <class 'dict'>
-----
Nom utilisateur: Leanne Graham
        Rue: Kulas Light
Nom utilisateur: Ervin Howell
        Rue: Victor Plains
Nom utilisateur: Clementine Bauch
        Rue: Douglas Extension
Nom utilisateur: Patricia Lebsack
        Rue: Hoeger Mall
Nom utilisateur: Chelsey Dietrich
        Rue: Skiles Walks
Nom utilisateur: Mrs. Dennis Schulist
        Rue: Norberto Crossing
Nom utilisateur: Kurtis Weissnat
        Rue: Rex Trail
Nom utilisateur: Nicholas Runolfsdottir V
        Rue: Ellsworth Summit
Nom utilisateur: Glenna Reichert
        Rue: Dayna Park
Nom utilisateur: Clementina DuBuque
        Rue: Kattie Turnpike
```

On récupère une ***liste de dictionnaires***.

24.2.2 Ecriture de fichier

Le module *json* peut aussi écrire le contenu d'une ***liste de dictionnaires*** dans un fichier json.

```
chemin_dossier = os.getcwd()
chemin_fichier = os.path.join(chemin_dossier, 'sortie.json')

donnees = [
  {
    "test":True,
    "bidule":5,
    "machin":"chose",
  },
  {
    "test":False,
    "bidule":42,
    "machin":"données",
  },
]
```

```python
with open(chemin_fichier, 'w', encoding="utf-8") as fichier:
  # ensure_ascii = False permet d'éviter d'afficher les codes ascii
  # des caractères spéciaux (comportement par défaut) mais
  # les afficher directement
  # indent permet de préciser le nombre d'espaces utilisés pour
  # l'indentation. Par défaut : 0
  json.dump(donnees, fichier, indent=2, ensure_ascii=False)
```

Contenu du fichier *sortie.json* :

```
[
  {
    "test": true,
    "bidule": 5,
    "machin": "chose"
  },
  {
    "test": false,
    "bidule": 42,
    "machin": "données"
  }
]
```

24.2.3 Exercices

1) Créer une méthode read_file_json(path) qui renvoie une liste contenant le contenu d'un fichier au chemin path (à partir du dossier de travail actuel) la mettre dans le module mytools, dans files.py

See *Answer*

2) Créer une méthode write_file_json(path, content) qui écrit le contenu du dictionnaire content dans un fichier au chemin path (le créant / l'écrasant éventuellement) la mettre dans le module mytools, dans files.py

See *Answer*

Notes :

- ne pas gérer les cas où les fichiers n'existent pas
- ne pas s'embêter avec join, faire

`file_path = os.getcwd() + path`

Par exemple, pour récupérer le contenu du fichier message, on doit faire

`contenu = read_file_json("/path/to/data.json")`

24.3 Files csv

24.3.1 Lecture de fichier

Le *CSV* est un autre format de fichier. https://fr.wikipedia.org/wiki/Comma-separated_values Comme pour le JSON, un module est dédié à la mise en forme des données et à l'interprétation du contenu des fichiers csv : le module *csv*.

En utilisant un fichier *demo_file.csv* qui contient :

```
Prenom, Nom, Email, Age, Ville
Robert, Lepingre, bobby@exemple.com, 41, Paris
Jeanne, Ducoux, jeanne@exemple.com, 32, Marseille
Pierre, Lenfant, pierre@exemple.com, 23, Rennes
```

```
import os
import csv

spreadsheet = []
chemin_dossier = os.getcwd()
chemin_fichier = os.path.join(chemin_dossier, 'demo_file.csv')
with open(chemin_fichier,"r",encoding="utf-8") as file:
  reader = csv.reader(file, delimiter=",")
  for row in reader:
    spreadsheet.append(row)
print(spreadsheet)
chemin_fichier = os.path.join(chemin_dossier, 'duplicate.csv')
```

```
[['Prenom', ' Nom', ' Email', ' Age', ' Ville'], ['Robert', ' Lepingre', '
bobby@exemple.com', ' 41', ' Paris'], ['Jeanne', ' Ducoux', '
jeanne@exemple.com', ' 32', ' Marseille'], ['Pierre', ' Lenfant', '
pierre@exemple.com', ' 23', ' Rennes']]
```

On récupère une ***liste de liste de str***.

24.3.2 Ecriture de fichier

On utilise également le module *csv* pour écrire une ***liste de liste de str*** dans un fichier csv.

```
chemin_dossier = os.getcwd()
chemin_fichier = os.path.join(chemin_dossier, 'sortie.csv')

data = [
  ["1","truc","bidule"],
  ["2","chose","machin"],
]
```

```
with open(chemin_fichier,"w", encoding = "utf-8") as file:
  writer = csv.writer(file, delimiter = ",", quotechar='"')
  # writer.writerows(data)
  for row in data:
    writer.writerow(row)
```

Contenu du fichier sortie.csv :

```
1,truc,bidule

2,chose,machin
```

On a des lignes vides supplémentaires présentes dans le fichier final. La raison provient du fait qu'on a deux manières d'écrire les saut de lignes : \n ou \r\n. Le module csv transforme les \n en \r\n et ajoute de ce fait des sauts de lignes non désirés.

On peut préciser

```
newline = ""
```

lors de l'ouverture du fichier en écriture pour éviter ce comportement.

```
data = [
  ["1","truc","bidule"],
  ["2","chose","machin"],
]

with open(chemin_fichier,"w", encoding = "utf-8", newline = "") as file:
  writer = csv.writer(file, delimiter = ",", quotechar='"')
  # writer.writerows(data)
  for row in data:
    writer.writerow(row)

with open(chemin_fichier,"r",encoding="utf-8") as file:
  reader = csv.reader(file, delimiter=",")
  for row in reader:
    print(row)
```

```
['1', 'truc', 'bidule']
['2', 'chose', 'machin']
```

Contenu du fichier sortie.csv :

```
1,truc,bidule
```

```
2,chose,machin
```

24.3.3 Exercices

Modifier le programme de gestion de listes de courses en ajouter des possibilités d'importer/d'exporter une liste de courses en json et en csv (demander à l'utilisateur d'écrire à la main le chemin vers le fichier)

See *Answer*

25 BDD

Une BD relationnelle est une base données (un ensemble de tables) avec des éventuelles relations entre les tables. Les relations peuvent être de plusieurs types :

- 1 à 1 relation 1 à 1 (un objet de type A est associé à un seul objet de type B et inversement) par exemple : un crâne associé à un seul cerveau et inversement
- relation 1 à plusieurs / plusieurs à 1 : par exemple un employé fait partie d'une entreprise mais une entreprise a plusieurs employés
- relations plusieurs à plusieurs : par exemple un utilisateur est associé à une liste de produit (sa liste de course) et chaque produit peut être dans le panier de plusieurs utilisateurs

Pour effectuer ses relations, mais aussi pour identifer chaque élément d'une table de manière unique, on utilise des clés primaires : un id (identifiants). *Pour les relations on peut utiliser aussi d'autres clés pour symboliser les relations, c.f. un cours de SQL pour en savoir plus)*

Pour communiquer avec une base de données, on utilise le langage SQL (Structured Query Langage). *Pourquoi stucturé ?* Tous les lignes d'une table ont forcément les mêmes attributs (potentiellement None). Quand on fait du big data notamment, ce n'est pas forcément le cas, on peut utiliser des bases non structurées. Si vous ne connaissez pas le SQL, ne vous inquiétez pas, nous ne verrons que les bases. Nous allons voir comment envoyer des requêtes SQL à une base de données et éventuellement récupérer des données dans sa réponse. Nous n'utiliserons que des requêtes SQL très simples. Vous pouvez vous référer à un cours de SQL si vous voulez faire des requêtes plus avancées.

Il existe plusieurs SGBD (systeme de gestion de base de données) mySql, postgresql, sqlserver, … On va utiliser SQLite. Sa particularité est de ne pas reproduire le schéma habituel client-serveur mais d'être directement intégrée aux programmes. L'intégralité de la base de données (déclarations, tables, index et données) est stockée dans un fichier indépendant de la plateforme.

Il existe dans Python une API unifiée, où les mêmes opérations s'effectuent avec les mêmes fonctions et méthodes, quelle que soit la base de données relationnelle SQL à laquelle on accède. Pour SQLite en l'occurrence, la méthode *.connect* ne prend qu'un seul paramètre, le nom de fichier, car SQLite est une base de données embarquée sans sécurité et n'a donc pas besoin de mot de passe ou de nom d'utilisateur. Si la base de donnée n'existe pas elle va etre crée. Sinon, on se connecte.

```
import sqlite3
cnx = sqlite3.connect("database.sqlite3")
```

Si vous utilisez un autre SGBD, seule la connexion à votre base de données sera différente (et selon le SGBD, il faudra consulter la documentation pour savoir comment s'y connecter). Le reste de ce qu'on va faire sera toujours valide.

25.1 Create

Un curseur, dans une base de données, représente un endroit où l'on se trouve dans la base, notamment lorsqu'on récupère des données. C'est ce curseur qui sert à exécuter des commandes

SQL, et qu'on utilise pour parcourir les résultats envoyés par la base, s'il y en a.

```
import sqlite3

cnx = sqlite3.connect("database.sqlite3")
curseur = cnx.cursor()
# Ici on crée une nouvelle table dans notre base de
# données si elle n'existe pas déjà.
curseur.execute("""
CREATE TABLE IF NOT EXISTS personne(
  id integer PRIMARY KEY AUTOINCREMENT NOT NULL,
  nom varchar(30) NOT NULL,
  prenom varchar(20) NOT NULL,
  age integer unsigned NOT NULL,
  CONSTRAINT unique_names UNIQUE (nom,prenom)
)
""")
print("table personne créée")
```

```
table personne créée
```

```
""" ma str """
```

est une manière de définir une chaîne de caractères qui sera telle quelle, avec les retours à la ligne, les caractères spéciaux, … C'est cette syntaxe qui est également utilisée pour la *documentation*.

```
test = "une chaine\navec deux lignes"
print(test)

print("______")

test = """une chaine
avec deux lignes
    et une tabulation !"""
print(test)
```

```
une chaine
avec deux lignes
------
une chaine
avec deux lignes
    et une tabulation !
```

Autoincrement la base de données s'occuper de générer l'id au fur à mesure lors de l'ajout de données *(il faudrait sinon se charger manuellement de s'assurer que l'id est unique dans la table)*

NOT NULL équivalent de NOT NONE : on met une contrainte sur la colonne, les éléments ne peuvent pas être None

CONSTRAINT unique_names UNIQUE (nom,prenom) contrainte : pas d'homonyme dans la table

Sur VSCode, avec l'extension SQLite : ***clic droit sur le fichier -> open database*** ouvre un volet "SQLITE EXPLORER" en dessous de l'explorateur de fichiers.

Pour afficher les données d'une table : ***clic droit sur la table -> show table*** (ne pas oublier de rafraichir les bases, on a un bouton refresh databases)

Il existe aussi un outil pour gérer une base de données SQLite qui s'appelle SQLite browser https://sqlitebrowser.org/ .

On va insérer des données dans la base

```
# INSERE DANS la table personne une ligne où les colonnes
# (nom, prenom, age) ont les valeurs ('Doe','John',45)
sql = "INSERT INTO personne (nom, prenom, age) VALUES ('Doe','John',45)"
curseur.execute(sql)
print("Personne ajoutée")
```

```
Personne ajoutée
```

Si on regarde le contenu de notre base, rien a été ajoutée ! *Que s'est-il passé ?* Pour comprendre pourquoi les lignes précédentes n'ont pas modifié la base de données, il faut comprendre le concept de transaction.

Les transactions existent pour répondre à une problématique. Prenons l'exemple d'une appli de banque. Je veux vous faire un virement : 2 requêtes sont faites :

1) la première pour me retirer la somme sur mon compte
2) la seconde pour vous donner la somme sur le votre

Qu'est ce qu'il se passe si un problème se produit entre les deux requêtes ? L'argent est retiré de mon compte, mais que vous ne recevez rien.

Une transaction est un ensemble d'instructions envoyé à une base de données qui ne sera réellement exécutée sur la base uniquement si aucune erreur ne s'est produit et tous les changements sont effectués en même temps.

Si il y a un problème on fait un **rollback** de transaction (un retour en arrière : tout est annulé)

Si tout se passe bien on fait un **commit** de transaction (on applique réellement toutes les modifications)

Avec SQLite, on a pas besoin d'explicitement démarrer une transaction. Par contre, il faut soit la commit, soit faire un rollback

```
try :
  sql = "INSERT INTO personne (nom, prenom, age) VALUES ('Curie','Marie',12)"
  curseur.execute(sql)
  cnx.commit()
except sqlite3.IntegrityError:
  # Erreur en rapport avec la contrainte d'unicité de
  # (nom,prenom) de la table
  # sqlite3.IntegrityError: UNIQUE constraint failed: personne.nom, personne.
 ↪prenom
  print("Cette personne existe déjà")
  cnx.rollback()
else:
  print("Personne ajoutée")
```

Il ne faut jamis écrire les choses comme on les a écrite. En effet, si on est pas attentif on a un **problème de sécurité : injection SQL**. Ce genre d'écriture :

```
sql = "INSERT INTO personne (nom, prenom, age) VALUES ('Dupont','Maryline',32)"
```

et encore plus, ce style de choses :

```
sql = f"INSERT INTO personne (nom, prenom, age) VALUES ('{nom}','{prenom}',{age})"
```

sont à proscrire pour des raisons de sécurité. Il ne jamais faire les choses comme ça, on a des problèmes potentiels d'injection SQL. *Qu'est ce que l'injection SQL ?*

Imaginons qu'on demande à l'utilisateur son nom, son prénom, son age :

```
nom = input("...")
prenom = input("...")
age = int(input("..."))
```

et qu'on fait la requête :

```
sql = f"INSERT INTO personne (nom, prenom, age) VALUES ('{nom}','{prenom}',{age})"
```

un utilisateur malveillant peut renseigner les informations suivantes :

```
nom : X','bidule',20) DELETE FROM personne INSERT INTO personne (nom, prenom, age)
        VALUES ('Philippe
prenom : hahaha
age : 34
```

La requête exécutée par la base sera :

```
INSERT INTO personne (nom, prenom, age) VALUES ('X','bidule',20)
DELETE FROM personne
INSERT INTO personne (nom, prenom, age) VALUES ('Philippe','hahaha',34)
```

Notre table sera complètement effacée, et il ne restera plus que la ligne contenant "Philippe", "hahaha",34.

Cet exemple illustre l'idée derrière le processus d'injection SQL, mais il est incomplet. Un lecteur perspicace remarquera que l'utilisateur malveillant avait tout de même pas mal d'informations : le nom de la table, le nombre de colonnes de cette table, le nom des différentes colonnes, et que ce ne sont pas des choses qu'on peut deviner comme ça !

Cela permet de montrer que donner des informations à un utilisateur malveillant peut augmenter le risque de problèmes de sécurité, et il est donc bien important de ne pas exposer les utilisateurs à des logs de debug ou des informations sur le fonctionnement de l'application. Néanmoins, même sans aucune information, il est possible de faire une attaque en utilisant ce principe. Par injection SQL, on peut obtenir des informations sur le SGBD utilisé, et ensuite chaque base de données contient des tables (de nom, de nature et de structures différentes selon le SGBD) qui contiennent des informations sur les tables. Ainsi, un plan d'attaque est de repérer le SGBD utilisé, puis d'aller récupérer des informations sur les tables comme leur noms, le nom de leur colonnes, et les informations qui paraissaient tomber de nul part dans l'exemple ci dessus sont en réalité accessibles. Si vous souhaitez voir un exemple complet d'attaque, je vous invite à regarder cette vidéo https://www.youtube.com/watch?v=ciNHn38EyRc

Il existe un mécanisme protégé contre les injections SQL :

```
try:
  sql = "INSERT INTO personne (nom, prenom, age) VALUES (?,?,?)"
  data = ("Dupont","Maryline",32) # les points d'interrogation
  # seront remplacés de manière sécurisé par les données du tuple
  # dans le même ordre
  curseur.execute(sql,data)

  cnx.commit()
except sqlite3.IntegrityError:
  print("Cette personne existe déjà")
  cnx.rollback()
else:
  print("Personne ajoutée")

# ne pas oublier, à la fin, lorsqu'on en a plus besoin, de fermer
# la fermer la connexion à la base
# on peut aussi utiliser un Context Manager, comme pour les fichiers
cnx.close()
```

En pratique, on utilise souvent des ORM (object relational mapping) par exemple peewee, sqlalchemy qui permettent de faire le pont entre le modèle objet et la base de données.

Par exemple : en partant d'une base de données, génère le code des classes ou inversement : en partant du code des classes, génère la base de données

Avantages :

- lier le code à la base de données, de centraliser le modèle objet i.e. si on modifie l'un, l'autre

est modifié également

- d'autres languages de requêtes sont utilisés, qui permettent de profiter de la structure des objets plus simplement. une requête du type

```
SELECT * from salarie WHERE salarie.adresse.codepostal = 900000
```

est possible (en SQL, on aurait besoin de faire des jointures)

```
SELECT *  FROM personne p
          JOIN adresse a
          ON p.id = a.personneId
          WHERE a.codepostal = 900000
```

Parfois, on ne passe même pas par un langage de requête, mais on on utilise des choses du type :

```
request.table(Personne).select().where(p.adresse.codepostal == 90000)
```

Inconvénients :

- la méthode sqlite3 en utilisant le SQL pur est meilleur en terme de performances. Ces frameworks travaillent en SQL de manière cachée, la communication avec la base de données se fait en toute finalité avec du SQL.
- certaines fonctionnalités SQL ne sont parfois pas implémentées dans les langages de requêtes spécifiques aux ORM

Si on a besoin d'une performance optimale avec la base de données ou qu'on effectue un traitement spécifique, on peut avoir besoin de faire du SQL pur.

25.2 Read

Nous allons voir les opérations de base. Nous avons vu la création d'une ligne, nous allons voir la lecture, la modification et la suppression. Nous allons faire les opérations de Create, Read, Update et Delete. On peut parler de **CRUD** pour parler de ces opérations.

```python
import sqlite3

connexion =  sqlite3.connect("database.sqlite3")
curseur = connexion.cursor()

# SELECTIONNE toutes les colonnes DE LA TABL personne
# person
sql = "SELECT * FROM personne"
curseur.execute(sql)

my_utilisateurs = curseur.fetchall()  # c'est toujours
                                      # une liste
                                      # de tuples
print(my_utilisateurs)
```

```
print(curseur.fetchall()) # <- le curseur se vide

print("_____")

for utilisateur in my_utilisateurs:
  print(utilisateur)

print("_____")

for id, nom, prenom, age in my_utilisateurs:
  print(f"id : {id}")
  print("nom : " + nom)
  print("prenom : " + prenom)
  print("age : " + str(age))
```

```
[(1, 'Cury', 'Marie', 12), (2, 'Dupont', 'Maryline', 32)]
[]
-----
(1, 'Cury', 'Marie', 12)
(2, 'Dupont', 'Maryline', 32)
-----
id : 1
nom : Cury
prenom : Marie
age : 12
id : 2
nom : Dupont
prenom : Maryline
age : 32
```

Il n'est pas nécessaire de faire un *commit*, puisqu'on a pas modifié les tables, on a fait que des requêtes pour récupérer des données.

On peut récupérer uniquement les valeurs des colonnes qu'on souhaite

---

```
sql = "SELECT nom, prenom FROM personne"
curseur.execute(sql)

utilisateur = curseur.fetchone() # <- un tuple
print(utilisateur)
utilisateur = curseur.fetchone() # <- un tuple
print(utilisateur)
```

```
('Cury', 'Marie')
('Dupont', 'Maryline')
```

Enfin, on peut rajouter des conditions si on souhaite récupérer uniquement certaines données. Pour aller plus loin, il faudrait connaître plus de SQL.

```
sql = "SELECT * FROM personne WHERE nom='Curie'"
curseur.execute(sql)
utilisateur = curseur.fetchone()
print(utilisateur)

data = (2,)
sql = "SELECT * FROM personne WHERE id=?"
curseur.execute(sql, data)
utilisateur = curseur.fetchone()
print(utilisateur)


# connexion.commit() # pas necessaire, puisqu'on ne
# fait aucune modifications sur le tableaux
connexion.close()
```

```
None
(2, 'Dupont', 'Maryline', 32)
```

25.3 Update

La seule chose nouvelle dans cette partie, c'est la requête SQL utilisée.

On notera de plus qu'on peut utiliser le mécanisme de protection contre les injections SQL pour proprement insérer des valeurs dans une requête à l'aide d'un dictionnaire. Au lieu de mettre des points d'intérogations dans la requête, il suffit d'utiliser

```
:cle
```

```
import sqlite3

connexion = sqlite3.connect("database.sqlite3")
curseur = connexion.cursor()

id = input("Quel est l'id de l'utilisateur à modifier ? ")
nom = input("Quel est le nouveau nom ? ")
prenom = input("Quel est le nouveau prenom ? ")
age = int(input("Quel est le nouvel age ? "))

my_utilisateur = {
    "id": id,
    "last_name": nom,
    "first_name": prenom,
```

```
    "age": age
}

curseur.execute(
    "UPDATE personne SET nom=:last_name, prenom=:first_name, age=:age WHERE id=:
 ↪id",
    my_utilisateur
)

connexion.commit()
connexion.close()
```

```
Quel est l'id de l'utilisateur à modifier ? 2
Quel est le nouveau nom ? Truc
Quel est le nouveau prenom ? Bidule
Quel est le nouvel age ?123
```

25.4 Delete

```
import sqlite3

connexion = sqlite3.connect("database.sqlite3")
curseur = connexion.cursor()

utilisateur_id = int(input("Quel est l'id de l'utilisateur a supprimer ? "))

curseur.execute(
    "DELETE FROM person WHERE id=?",
    [utilisateur_id]
)

connexion.commit()
connexion.close()
```

25.5 Exercices

Créer un package repository, avec un module productDao.py.

Coder une classe ProductDao qui contiendra des méthodes

- addProduct(self, p)
- getProductById(self, id)
- getAllProducts(self)
- updateProduct(self, p)
- deleteProductById(self, id)

qui implémentent les fonctionnalités associées pour une table produit et qui a comme propriétés le nom de la base, le nom de la table.

Enfin, elle aura une méthode *close* pour fermer la connexion et une méthode *init_table* pour créer la table et y mettre des données initiales.

Les opérations / erreurs seront loggées dans un fichier app.log

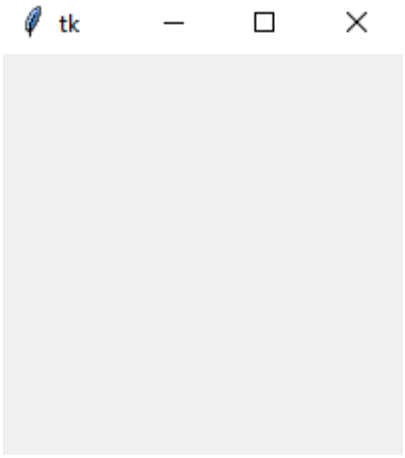
See *Answer*

26 Tkinter

Tkinter (Tool Kit INTERface) est le seul framework intégré dans la bibliothèque standard de Python permettant de créer une interface graphique.

Documentation : https://docs.python.org/3/library/tkinter.html

```python
import tkinter

# créer une fenêtre
# On instancie la classe Tk
fenetre_principale = tkinter.Tk()
# la démarrer
fenetre_principale.mainloop()
```



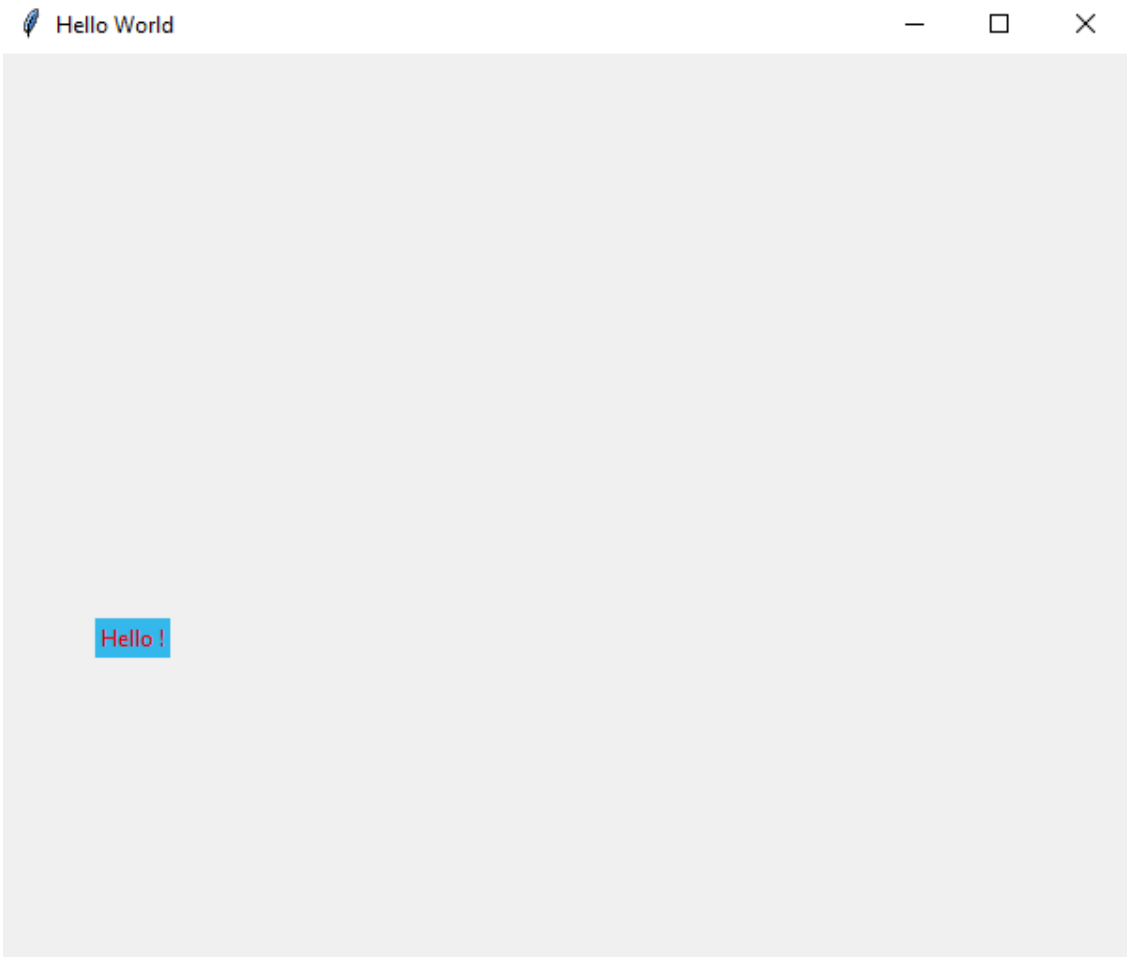
On peut configurer la fenêtre principale en changeant certains de ces attributs.

On peut également lui attacher d'autres composants, des widgets. Il y a plusieurs widgets disponibles :

- label : un texte
- button : un bouton
- entry : un champ pour rentrer une ligne de texte
- text : un champ pour rentrer du texte (plusieurs lignes)
- frame : une région rectangulaire utilisée pour regrouper des widgets, ou pour faire un padding entre les widgets
- … (voir la doc)

Il faut donc les instancier, les paramétrer et les attacher à la fenêtre.

```python
import tkinter

fenetre_principale = tkinter.Tk()
fenetre_principale.title("Hello World")
fenetre_principale.geometry("600x480")
# "Largeur x hauteur"

# Text : un label
# c'est un widget : un élément qu'on peut ajouter
# sur une fenêtre, avec lequel on peut éventuellement
# intéragir
message_label = tkinter.Label(fenetre_principale, text="Hello !")
message_label["fg"] = "red"
message_label["background"] = "#34b7eb"

message_label.pack()
# positionne l'élément dans la fenetre
message_label.place(x=50, y=300)

fenetre_principale.mainloop()
```

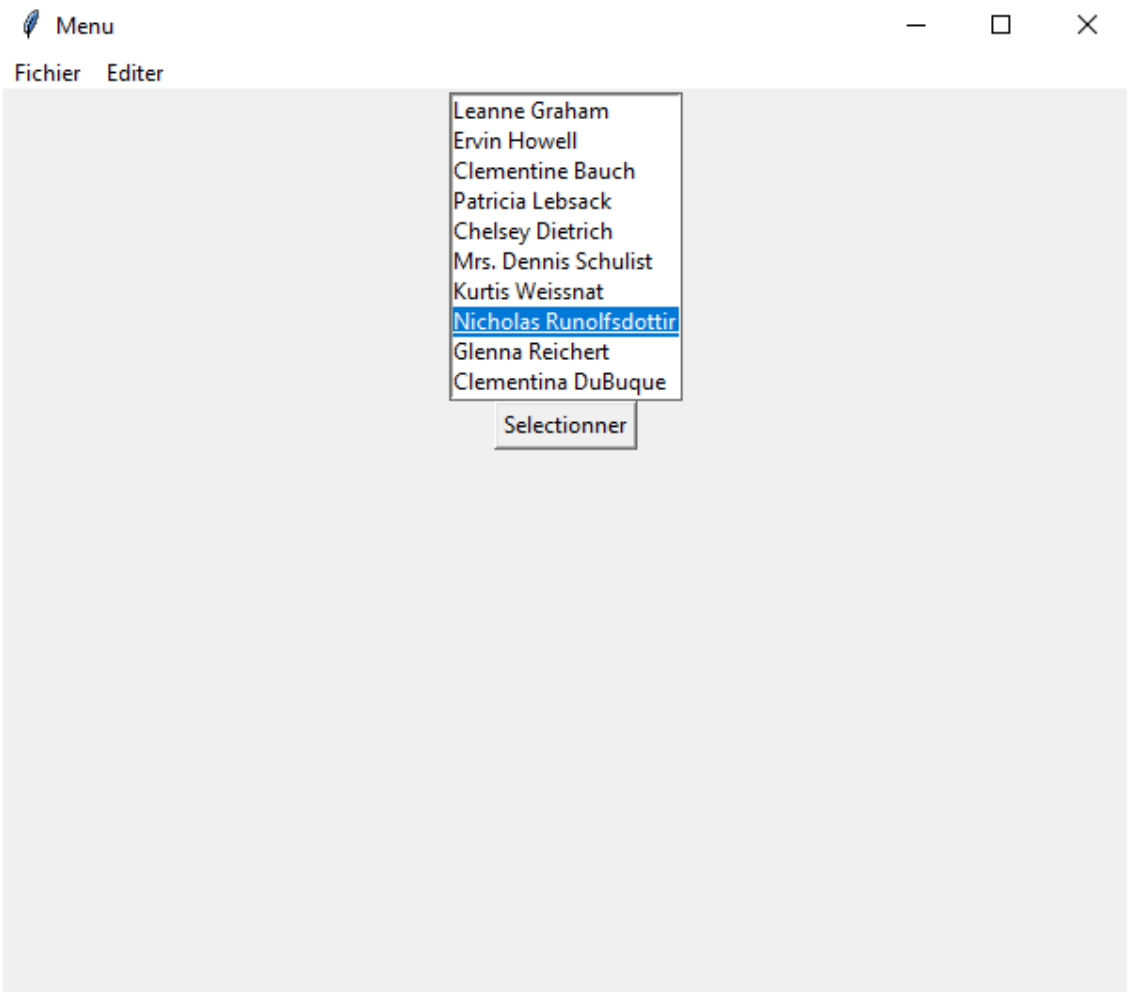


On peut créer des fenêtres plus complètes et complexes en utilisant de nombreux widgets. L'intéraction avec l'utilisateur se fait via des fonctions. Dans l'exemple suivant, on fournit au menu et au bouton leurs fonctionnalités dans leur constructeurs. Ce sont les fonctions en tant qu'objet qu'on donne en argument du constructeur.

```
import tkinter
import tkinter.filedialog as filedialog
import json

fenetre_principale = tkinter.Tk()
fenetre_principale.title("Menu")
fenetre_principale.geometry("600x480")

def importer_json():
  chemin_fichier = filedialog.askopenfilename(
    title="Choisir un fichier JSON",
    filetypes=[
```

```
      ("json","*.json"), # (label, extension)
      ("Tous les fichiers", ".*"),
    ]
  )
  with open(chemin_fichier, "r", encoding="utf-8") as fichier:
    utilisateurs = json.load(fichier)
  for u in utilisateurs:
    liste_elements.insert(tkinter.END,u.get("name"))

def show_user():
  index = liste_elements.curselection()
  element = liste_elements.get(index)
  print(element)

liste_elements = tkinter.Listbox(fenetre_principale)
liste_elements.pack()

select_btn = tkinter.Button(fenetre_principale, text="Selectionner",␣
 ↪command=show_user)
select_btn.pack()

menu_bar = tkinter.Menu(fenetre_principale)
sous_menu_fichier = tkinter.Menu(menu_bar, tearoff=0)
sous_menu_fichier_preference = tkinter.Menu(sous_menu_fichier, tearoff=0)

sous_menu_fichier_preference.add_command(label="extensions")

# Configuration du sous menu fichier
sous_menu_fichier.add_command(label="Importer un fichier JSON",␣
 ↪command=importer_json)
sous_menu_fichier.add_command(label="Exporter un fichier JSON")
sous_menu_fichier.add_cascade(label="Préferences",␣
 ↪menu=sous_menu_fichier_preference)

# add_cascade(): pour l'ajout de sous menu
menu_bar.add_cascade(label="Fichier", menu=sous_menu_fichier)
menu_bar.add_cascade(label="Editer")

# On configure notre fenetre pour integrer notre menu
fenetre_principale.config(menu=menu_bar)

fenetre_principale.mainloop()
```

Nicholas Runolfsdottir V



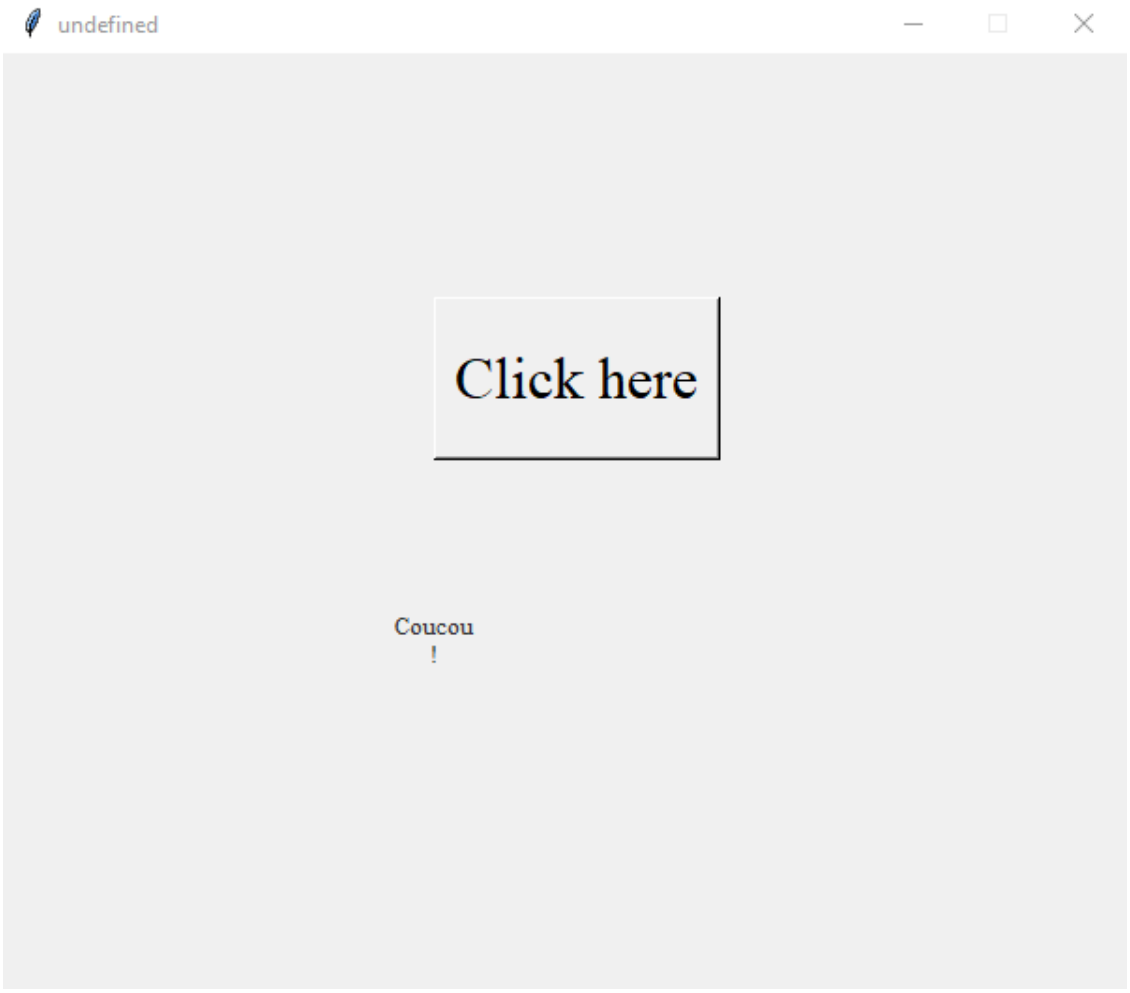
Dans la pratique, on ne code pas une interface graphique uniquement par le code comme on l'a fait ici. C'est pour cette raison que nous ne sommes pas rentrés dans les détails de tous les composants, et de tous les paramétrages disponibles. On utilise plutôt des outils en mode WYSIWYG (what you see is what you get) qui permettent de créer une IHM via une interface graphique, avec des outils pour paramétrer les différents widgets de manière plus ergonomique. Pour tkinter, il existe https://www.visualtk.com/ (qui n'est pas complet mais beaucoup plus pratique).

Par exemple, cet outil permet de générer le code suivant en quelques secondes

---

```
import tkinter as tk
import tkinter.font as tkFont

class App:
    def __init__(self, root):
        #setting title
        root.title("undefined")
```

```
        #setting window size
        width=600
        height=500
        screenwidth = root.winfo_screenwidth()
        screenheight = root.winfo_screenheight()
        alignstr = '%dx%d+%d+%d' % (width, height, (screenwidth - width) / 2,␣
 ↪(screenheight - height) / 2)
        root.geometry(alignstr)
        root.resizable(width=False, height=False)

        GButton_161=tk.Button(root)
        GButton_161["activebackground"] = "#8c1010"
        GButton_161["activeforeground"] = "#01aaed"
        GButton_161["bg"] = "#f0f0f0"
        ft = tkFont.Font(family='Times',size=23)
        GButton_161["font"] = ft
        GButton_161["fg"] = "#000000"
        GButton_161["justify"] = "left"
        GButton_161["text"] = "Click here"
        GButton_161["relief"] = "raised"
        GButton_161.place(x=230,y=130,width=153,height=87)
        GButton_161["command"] = self.GButton_161_command

        GMessage_290=tk.Message(root)
        ft = tkFont.Font(family='Times',size=10)
        GMessage_290["font"] = ft
        GMessage_290["fg"] = "#333333"
        GMessage_290["justify"] = "center"
        GMessage_290["text"] = "Coucou !"
        GMessage_290.place(x=190,y=300,width=80,height=25)

    def GButton_161_command(self):
        print("command")
        # mon code...

if __name__ == "__main__":
    root = tk.Tk()
    app = App(root)
    root.mainloop()
```



Il existe d'autres modules et outils qui permettent de faire du graphique, notamment Qt, Pyside, PyQt. Ces modules ont des interfaces de design beaucoup plus développées, et sont des extensions de tkinter avec beaucoup plus de widgets disponibles.

Ils demandent cependant un niveau de connaissance en C++, le code source de Qt pour être maîtrisés.

Il est tout de même important de comprendre la structure du code généré par ces outils, puisqu'il va falloir coder tout ce qui est interaction avec l'utilisateur, et aussi parce qu'on peut profiter des différents concepts objets qu'on connait et des différentes techniques qu'on a pu voir dans le cours pour améliorer ce code, le rendre plus flexible, plus facilement utilisable.

L'exemple suivant permet de créer son propre widget, un menu qui peut être paramétrable en spécifiant une langue dans son constructeur. Ce code pourrait encore être largement modifié et amélioré, mais il permet d'avoir un aperçu de ce qu'il est possible de faire en graphique avec tout ce que cours vous a permis d'apprendre et de la puissance de la programmation orientée objet.

app.py

```
import tkinter
from functionalities.functionalities import show_user, importer_json
from mywidgets.myMenu import MyMenu
from mywidgets.myTk import MyTk

fenetre_principale = MyTk()

def importer():
  importer_json(liste_elements)

def click():
  show_user(liste_elements)

liste_elements = tkinter.Listbox(fenetre_principale)
liste_elements.pack()

select_btn = tkinter.Button(fenetre_principale, text="Selectionner",␣
 ↪command=click)
select_btn.pack()

# On configure notre fenetre pour integrer notre menu
fenetre_principale.config(menu=MyMenu(fenetre_principale, liste_elements, "FR"))

fenetre_principale.mainloop()
```

---

functionalities.functionalities.py

```
import tkinter
import tkinter.filedialog as filedialog
from tkinter import Listbox
import json

def importer_json(liste_elements:Listbox):
  chemin_fichier = filedialog.askopenfilename(
    title="Choisir un fichier JSON",
    filetypes=[
      ("json","*.json"), # (label, extension)
      ("Tous les fichiers", ".*"),
    ]
  )
  with open(chemin_fichier, "r", encoding="utf-8") as fichier:
    utilisateurs = json.load(fichier)
  for u in utilisateurs:
    liste_elements.insert(tkinter.END,u.get("name"))
```

```
def show_user(liste_elements:Listbox):
  index = liste_elements.curselection()
  element = liste_elements.get(index)
  print(element)
```

---

messages.uimessages.py

```
menu_messages = {
  "FR": {
    "file":"Fichier",
    "import":"Importer un fichier JSON users",
    "export":"Exporter un fichier JSON users",
    "option":"Options",
    "optionChoice":"Choix d'option"
  },
  "EN" : {
    "file":"File",
    "import":"Import users JSON",
    "export":"Export JSON file",
    "option":"Options",
    "optionChoice":"Option choice"
  },
}
```

---

mywidgets.myMenu.py

```
import tkinter
from functionalities.functionalities import importer_json
from messages.uimessages import menu_messages

class MyMenu(tkinter.Menu):
  def __init__(self, parent,listbox, langage):
    super().__init__(parent)
    self.listbox = listbox

    label_file = menu_messages[langage]["file"]
    label_import = menu_messages[langage]["import"]
    label_export = menu_messages[langage]["export"]
    label_option = menu_messages[langage]["option"]
    label_option_choice = menu_messages[langage]["optionChoice"]
```

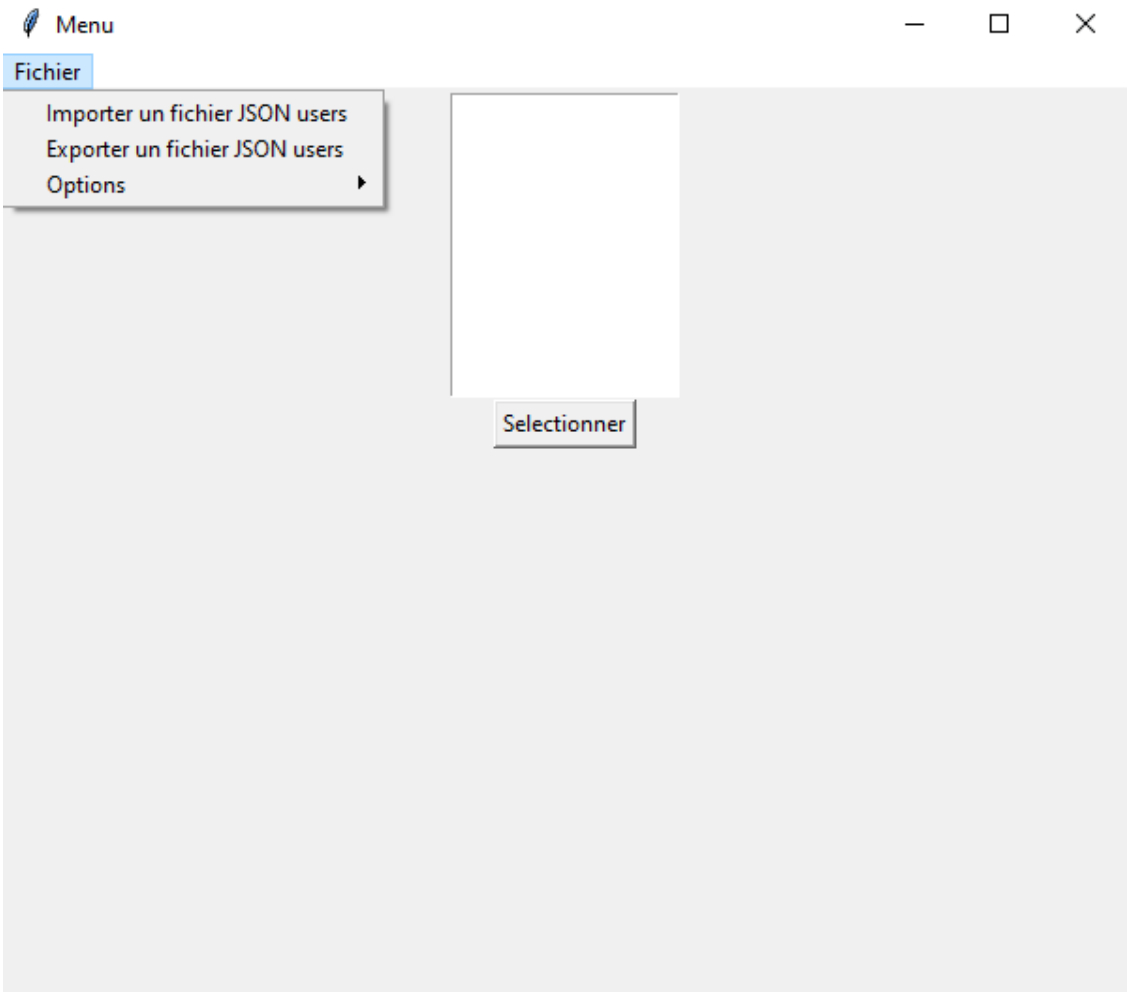
```
    sous_menu_fichier = tkinter.Menu(self, tearoff=0)
    sous_menu_fichier_preference = tkinter.Menu(sous_menu_fichier, tearoff=0)

    sous_menu_fichier_preference.add_command(label=label_option_choice)

    # Configuration du sous menu fichier
    sous_menu_fichier.add_command(label=label_import, command=self.importer)
    sous_menu_fichier.add_command(label=label_export)
    sous_menu_fichier.add_cascade(label=label_option,␣
↪menu=sous_menu_fichier_preference)

    # add_cascade(): pour l'ajout de sous menu
    self.add_cascade(label=label_file, menu=sous_menu_fichier)

  def importer(self):
    importer_json(self.listbox)
```

---

mywidgets.myTk.py

```
import tkinter

class MyTk(tkinter.Tk):
  def __init__(self):
    super().__init__()
    self.title("Menu")
    self.geometry("600x480")
```

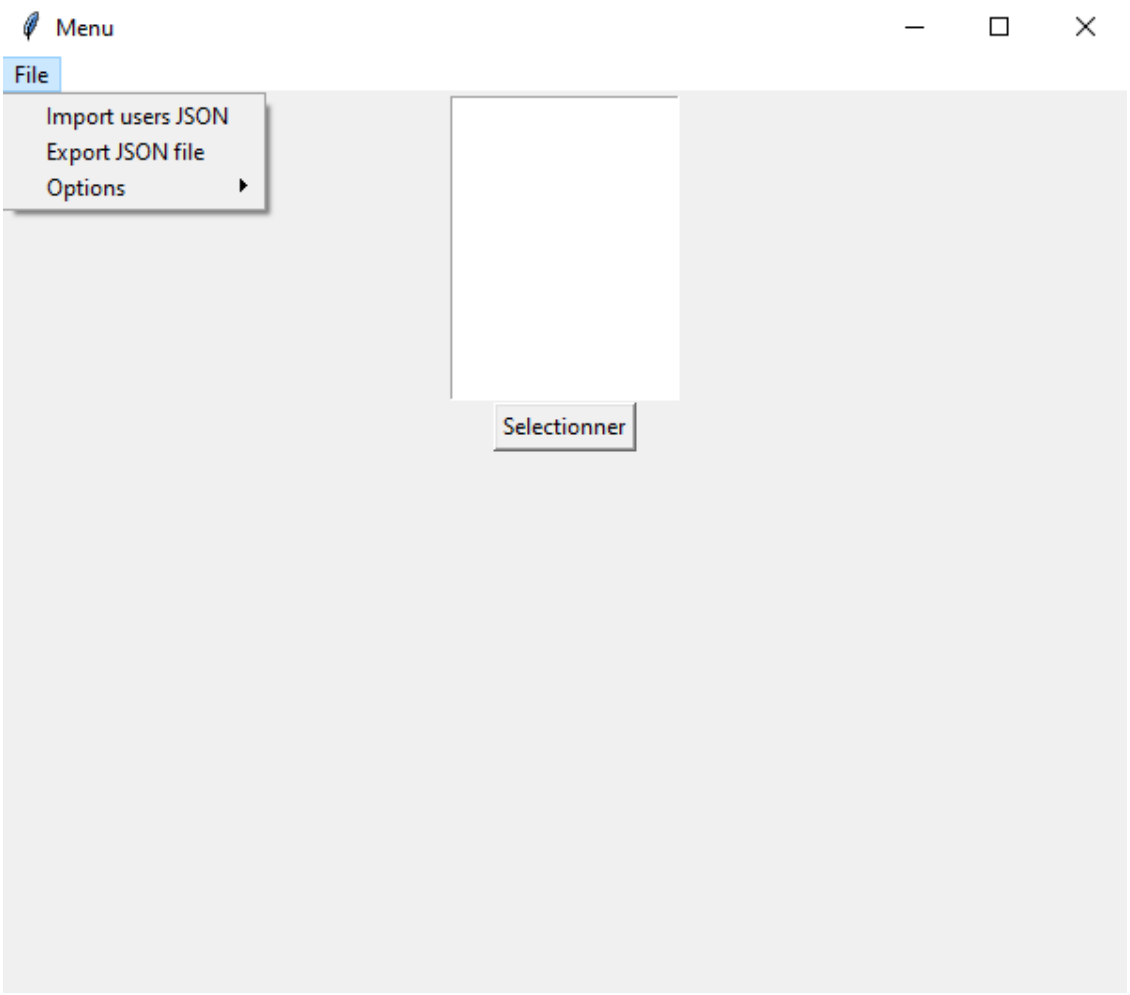
Voici un aperçu du resultat de ce code



En modifiant la langue dans le constructeur de *MyMenu* dans *app.py*

```
import tkinter
from functionalities.functionalities import show_user, importer_json
from mywidgets.myMenu import MyMenu
from mywidgets.myTk import MyTk

fenetre_principale = MyTk()

def importer():
  importer_json(liste_elements)

def click():
  show_user(liste_elements)

liste_elements = tkinter.Listbox(fenetre_principale)
```

```
liste_elements.pack()

select_btn = tkinter.Button(fenetre_principale, text="Selectionner",␣
 ↪command=click)
select_btn.pack()

# On a juste à modifier le "FR" en "EN"
fenetre_principale.config(menu=MyMenu(fenetre_principale, liste_elements, "EN"))

fenetre_principale.mainloop()
```



27 TP

27.1 Etape 1 : menu liste de courses - produits en strings - utilisation d'une liste

See *Answer*

Faire un programme qui permet d'ajouter ou retirer des produits d'une liste de courses. Lorsque le programme demarre, l'utilisateur a le choix entre:

```
1- Afficher la liste de courses.

2- Ajouter un produit a la liste de courses (chaine de caracteres ex: pomme)

3- Retirer un produit de la liste de course

4- Supprimer toute la liste de courses

5- Quitter le programme
```

Tant que l'utilisateur ne saisit pas "quitter", on continue la boucle

Si l'utilisateur tape 1 : on affiche la liste des produits

Si l'utilisateur tape 2 : on lui demande le nom du produit à ajouter et on l'ajoute à la liste de courses

Si l'utilisateur tape 3 : on lui demande le nom du produit à retirer et on le retire de la liste de courses

Si l'utilisateur tape 4 : on vide la liste

Si l'utilisateur tape 5: on affiche "Au revoir" et on quitte le programme

Utiliser une fonction pour chacune des fonctionnalités

Problème : doublons. Si un produit est ajouté plusieurs fois, il sera répété. Au moment de la suppression, la question se pose de tout supprimer, ou de n'en supprimer qu'un seul.

27.2 Etape 2 : gestion des achats multiples - utilisation de dictionnaires pour

représenter les produits

See *Answer*

Transformer la liste de courses

La liste de courses reste une liste mais on y mettra des dictionnaires. Un dictionnaire contient: un nom de produit (str) et une quantité (int). Modifier le comportement de la partie utilisateur. Lors de l'ajout ou de la suppression d'un produit, demander à l'utilisateur combien de produits ajouter / supprimer.

Pour la demande de quantité, ne pas hésiter à faire des fonctions dans un module dédié pour faire toutes les vérifications nécessaires (vérification de type : s'assurer que l'utilisateur écrit bien un nombre, mais aussi vérification de cohérence, s'assurer que l'utilisateur écrit bien une quantité positive, …)

27.3 Etape 3 : Vers l'objet - création d'une classe produit

Créer un nouveau package dans le projet : myclasses

Y mettre un fichier product.py qui contient une classe Product qui est décrit par un id, une description, un prix, qui a des méthodes pour calculer la TVA d'un produit et le prix TTC (prix tva = 0.2 * prix, et prix ttc = prix + prix tva), un compteur du nombre total de produits instanciés, une méthode qui permet de print() correctement un produit, et une méthode qui compare des produits (deux produits sont les mêmes ssi ils ont le même id).

Bien respecter les bonnes pratiques d'encapsulation des champs dans des accesseurs en lecture et en écriture

Créer deux classes d'exception personnalisées correspondant à la tentative de mettre une valeur négative pour le prix ou l'id d'un produit

27.4 Etape 4 : Structure d'un panier pour mieux structurer le code

Cart

Attributs :

`lines:List[CartLine] : représente une liste de lignes dans le panier`

Méthodes :

```
get_total_price(self) : renvoie le prix total du panier

add_product(self, p:Product, qty: int) : ajoute une certaine quantité
                                          d'un produit

remove_product(self, p:Product, qty: int) : retire une certaine quantité
                                            d'un produit

clear(self) : vide le panier
```

CartLine

Attributs :

```
product:Product : le produit dans cette ligne

quantity:int : la quantité présente dans le panier
```

Méthodes :

```
remove(self,qty:int) : retire une certaine quantité de la ligne

add(self, qty:int) : ajoute une certaine quantité à la ligne
```

`get_price(self) : renvoie le prix total de la ligne`

Ne pas hésiter à rajouter d'autres attributs / méthodes si jugé nécessaire et bien faire encapsulation / **str**, **eq** et des exceptions personnalisées (quantité négative sur les CartLine par exemple)

27.5 Etape 5 : Création d'une table Product en BDD et d'un DAO pour gérer l'accès à la table

Créer un package repository, avec un module productDao.py

Coder une classe ProductDao qui contiendra des méthodes

```
addProduct(self, p)

getProductById(self, id)

getAllProducts(self)

updateProduct(self, p)

deleteProductById(self, id)
```

qui implémentent les fonctionnalités associées pour une table produit et qui a comme propriétés le nom de la base, le nom de la table. Enfin, elle aura une méthode close() pour fermer la connexion et une méthode init_table() pour créer la table et y mettre des données initiales. Les opérations / erreurs seront loggées dans un fichier app.log

27.6 Etape 6 : Modifier la partie utilisateur - proposer uniquement les produits en BDD

Adapter le code de la partie utilisateur en utilisant les classes Cart et CartLine définies précédemment.

Ajouter l'export/l'import JSON/CSV de la liste de courses.

Finaliser la partie utilisateur. Les utilisateurs ne peuvent choisir d'ajouter que les produits disponibles dans la base de données.

27.7 Etape 7 : Ajout de la partie administrateur pour gérer la table en base et du menu principal

See *Answer*

Ajouter un menu principal (choix partie admin / utilisateur) ainsi que la partie administrateur qui permettra de gérer les produits en base de données

28 Correction des exercices

28.1 Variables

28.1.1 Exercice 1

See *Question*

```
# Créer 3 variables avec des noms corrects et les
# afficher

variable_string = "Truc"
variable_int = 4
variable_bool = False

print(variable_string)
print(variable_int)
print(variable_bool)
```

28.1.2 Exercice 2

See *Question*

```
# Demander à l'utilisateur son prénom, puis lui
# dire bonjour personnellement

prenom = input("Entrez votre prénom : ")
print("Bonjour",prenom,"!")
```

28.2 Formatage

28.2.1 Exercice 1

See *Question*

```
# Demander à l'utilisateur sa rue, son code
# postal et sa ville puis afficher l'adresse complète

rue = input("Entrez votre adresse : ")
code_postal = input("Entrez votre code postal : ")
ville = input("Entrez votre ville : ")

print(f"Vous habitez {rue}, {code_postal} {ville}")
```

28.3 Conditions

28.3.1 Exercice 1

See *Question*

```
# Demander à l'utilisateur un nombre. Afficher ensuite
# si ce nombre est pair ou impair

nombre = int(input("Entrer un nombre "))
if (nombre % 2 == 0):
  print(f"{nombre} est pair")
else:
  print(f"{nombre} est impair")
```

28.3.2 Exercice 2

See *Question*

```
# Demander à l'utilisateur un mot. Afficher si ce mot
# contient la lettre "e" ou non (utiliser l'opérateur
# in)

mot = input("Entrer un mot ")
if "e" in mot:
  print(f"{mot} contient la lettre e")
else:
  print(f"{mot} ne contient pas la lettre e")
```

28.4 Boucles

28.4.1 Exercice 1

See *Question*

```
# Afficher tous les nombres de 1 à 15

for i in range(16):
  if i != 0:
    print(i)

for i in range(15):
  print(i+1)

for i in range(1,16):
  print(i)
```

28.4.2 Exercice 2

See *Question*

```
# Afficher tous les nombres pairs entre 1 et 10

for i in range(1,11):
  if (i % 2 == 0):
    print(i)

for i in range(2,11,2):
  print(i)
```

28.4.3 Exercice 3

See *Question*

```
# Afficher les 20 premiers termes de la suite de
# fibonacci.
# Elle est définie de la manière suivante :
# Les premiers termes sont 0 et 1, et les termes suivant
# sont définis comme étant la somme des deux termes
# précédents.
# Autrement dit, fibo(0) = 0, fibo(1) = 1, et
# fibo(n+2) = fibo(n) + fibo(n+1)

fibo_1 = 0
fibo_2 = 1
print(fibo_1)
print(fibo_2)
for i in range(3,21):
```

```
  fibo_3 = fibo_1 + fibo_2
  print(fibo_3)
  fibo_1 = fibo_2
  fibo_2 = fibo_3
```

28.4.4 Exercice 4

See *Question*

```
# Afficher un menu :
#     Menu
#     1) Convertir minutes en heures
#     2) Quitter
#     Votre choix ?
# Et implémenter les fonctionnalités.
# Par exemple, la conversion transformera 90 minutes
# en 1 h 30 minutes

while True:
  user_input = input("""Menu
1) Convertir minutes en heures
2) Quitter
Votre choix ? """)
  if (user_input == "2"):
    break
  if (user_input == "1"):
    while True:
      user_input = input("Donner le nombre de minutes ")
      minutes = int(user_input)
      if (minutes >= 0):
        break
      print("Merci d'entrer un nombre positif")
    print(f"{minutes} m = {minutes // 60} h {minutes % 60} m")
    input()
```

28.5 Exceptions

28.5.1 Exercice 1

See *Question*

```
# Ecrire un programme qui demande à l'utilisateur un
# nombre entre 1 et 10 et qui affiche le double. Si
# l'utilisateur n'écrit pas un nombre, lui redemander.

while True:
  user_input = input("Entrer un nombre entre 1 et 10 ")
  try:
    number = int(user_input)
    if 1 <= number <= 10:
      break
    print("Merci d'entrer un nombre entre 1 et 10")
  except ValueError:
    print("Merci d'entrer un nombre")
print(f"Le double de {number} est {2 * number}")
```

28.5.2 Exercice 2

See *Question*

```
# Ecrire un programme qui choisit un nombre
# aléatoirement entre 0 et 100 et qui demande à
# l'utilisateur de deviner ce nombre. Après chaque
# proposition, le programme indique si la valeur est
# plus grande ou plus petite que la proposition. Le
# programme gèrera le cas où l'utilisateur n'entre pas
# un nombre. Lorsque l'utilisateur a trouvé, lui
# donner son nombre de tentatives. Si l'utilisateur
# dépasse un certain nombre de coups, il a perdu.
# Indication : pour choisir un nombre aléatoire,
# utiliser :
import random # importe le module qui contient toutes
              # les fonctions permettant de gérer
              # l'aléatoire
nombre = random.randint(0,100)  # génère un nombre
                                # aléatoire entre 0 et
                                # 100
nbr_tentatives = 0
nbr_tentatives_max = 6
while True:
  user_input = input("Devine mon nombre entre 0 et 100 ")
  try:
    guess = int(user_input)
    nbr_tentatives+=1
```

```
    if guess == nombre:
      print(f"C'est bien ça, mon nombre était {nombre}")
      print(f"Tu as trouvé en {nbr_tentatives} tentatives !")
      break
    if nbr_tentatives == nbr_tentatives_max:
      print("Perdu !")
      break
    if guess <= nombre:
      print("Plus grand !")
    else:
      print("Plus petit !")
  except ValueError:
    print("Essaye de deviner un nombre !")

print("Fin du programme...")
```

28.6 Fonctions

28.6.1 Exercice 1

See *Question*

```
# Ecrire une fonction qui renvoie le résultat de la
# multiplication de deux nombres

def multiplication(number1,number2):
  return number1 * number2

print(f"Multiplication de 3 et 5 : {multiplication(3,5)}")
```

28.6.2 Exercice 2

See *Question*

```
# Ecrire une fonction qui affiche la table de
# multiplication d'un nombr ou le premier et le dernier
# multiples sont paramétrés

def table_multiplication(number,first_multiple, last_multiple):
  for i in range(first_multiple, last_multiple +1):
    print(f"{i} x {number} = {i*number}")

print("Table de multiplication de 13 : ")
table_multiplication(13,0,10)
```

28.6.3 Exercice 3

See *Question*

```
# Ecrire une fonction qui permet de demander à
# l'utilisateur un nombre et qui renvoie à coup sur un
# nombre.

def demander_saisie_nombre():
  while True:
    user_input = input("Merci d'entrer un nombre")
    try:
      result = int(user_input)
      return result
    except ValueError:
      pass

nombre = demander_saisie_nombre()
print(f"Nombre choisi : {nombre}")
```

28.6.4 Exercice 4

See *Question*

```
# - Coder convertir_heures_en_minutes(heures,minutes)
# qui renvoie la conversion d'une heure en minutes. Par
# exemple convertir_heures_en_minutes(1,30) renvoie 90
# - Coder convertir_minutes_en_heures(minutes) qui
# renvoie la conversion de minutes en heures. Par
# exemple convertir_minutes_en_heures(90) renvoie 1, 30
# - reprendre la fonction demander_saisie_nombre
# - coder demander_saisie_nombre_positif(invite) qui
# renvoie un nombre positif entré par l'utilisateur
# - on souhaite afficher un menu
#         Menu
#         1) Convertir minutes en heures
#         2) Convertir heures en minutes
#         3) Quitter
#         Votre choix ?
# Implémenter les fonctionnalités.

def convertir_heures_en_minutes(heures,minutes):
  return heures * 60 + minutes

def convertir_minutes_en_heures(minutes):
  return minutes // 60 , minutes % 60

def demander_saisie_nombre(invite):
  while True:
    user_input = input(invite)
    try:
      result = int(user_input)
      return result
    except ValueError:
      print("Seul les caractères [0-9] sont autorisés")

def demander_saisie_nombre_positif(invite):
  invite += " (positif) "
  while True:
    nombre_input = demander_saisie_nombre(invite)
    if nombre_input >= 0:
      return nombre_input

def choix_1():
  minutes_input = demander_saisie_nombre_positif("Entrer le nombre de minutes")
  heures, minutes = convertir_minutes_en_heures(minutes_input)
  print(f"{minutes_input} m = {heures} h {minutes} m")
  input()
```

```
def choix_2():
  heures = demander_saisie_nombre_positif("Entrer le nombre d'heures")
  minutes_input = demander_saisie_nombre_positif("Entrer le nombre de minutes")
  minutes = convertir_heures_en_minutes(heures, minutes_input)
  print(f"{heures} h {minutes_input} m = {minutes} m")
  input()

while True:
  user_input = input("""Menu
1) Convertir minutes en heures
2) Convertir heures en minutes
3) Quitter
Votre choix ? """)
  if (user_input == "3"):
    break
  if (user_input == "1"):
    choix_1()
  if (user_input == "2"):
    choix_2()
```

28.6.5 Exercice 5

See *Question*

```
# Recoder le jeu où l'on devine un nombre choisi
# aléatoirement par le programme. Cette fois ci, le
# programme va demander à l'utilisateur quelles bornes
# utiliser et lorsqu'il aura deviné un nombre incorrect,
# il lui proposera les choix cohérents
# Exemple : je choisi 1 et 100 comme bornes. Le
# programme me demande de saisir un nombre entre 1 et
# 100. Je choisi 50. Le programme me répond : le nombre
# est plus grand, choisi un nombre entre 51 et 100.
# Lorsque j'ai trouvé, le programme s'arrête.
# On va cette fois ci utiliser des fonctions.
# - reprendre les fonctions demander_saisie_nombre et
# demander_saisie_nombre_borne
# - coder decider_bornes() qui renvoie les bornes
# utilisées pour le jeu
# - coder jouer_un_coup(nombre,minimum,maximum) qui
# demande à l'utilisateur un entier entre minimum et
# maximum, et qui renvoie :
#         * un boolean à vrai uniquement si l'utilisateur a
#     gagné (il a choisi le paramètre nombre)
#         * le minimum pour le coup suivant
#         * le maximum pour le coup suivant
```

```python
# - coder jouer_une_partie(nombre,minimum,maximum) qui
# fait une partie du jeu où l'entier à deviner est le
# paramètre nombre, et où les bornes sont minimum et
# maximum et qui renvoie le nombre de tentatives du
# joueur
# - coder une fonction jouer() qui fait tout le jeu :
# demander à l'utilisateur les bornes du jeu, puis
# choisi un nombre aléatoirement, puis demande à
# l'utilisateur de deviner un nombre, en lui disant si
# le nombre est plus grand ou plus petit, et en lui
# proposant au fur et à mesure des choix cohérents avec
# ces indications. Lorsque le joueur trouve le nombre,
# la fonction le félicite, lui donne son nombre de
# tentatives et s'arrête.

import random

def demander_saisie_nombre(invite):
  while True:
    user_input = input(invite)
    try:
      result = int(user_input)
      return result
    except ValueError:
      print("Seul les caractères [0-9] sont autorisés")

def demander_saisie_nombre_borne(invite,minimum = 1, maximum = 10):
  invite += f" entre {minimum} et {maximum} "
  while True:
    nombre_input = demander_saisie_nombre(invite)
    if minimum <= nombre_input <= maximum:
      return nombre_input

def decider_bornes():
  while True:
    minimum = demander_saisie_nombre("Quelle est la borne minimale ? ")
    maximum = demander_saisie_nombre("Quelle est la borne maximale ? ")
    if minimum < maximum:
      return minimum, maximum
    print("Le minimum doit être plus petit que le maximum !")

def jouer_un_coup(nombre, minimum, maximum):
  essai = demander_saisie_nombre_borne("Devinez le nombre", minimum, maximum)
  if essai < nombre:
    print("Trop petit !")
    return False, essai + 1, maximum
  if essai > nombre:
```

```
    print("Trop grand !")
    return False, minimum, essai - 1
  print("Bravo !")
  return True, nombre, nombre

def jouer_une_partie(nombre, minimum, maximum):
  nombre_coup = 0
  while True:
    victoire, minimum, maximum = jouer_un_coup(nombre, minimum, maximum)
    nombre_coup += 1
    if victoire:
      return nombre_coup

def jouer():
  minimum, maximum = decider_bornes()
  nombre = random.randint(minimum,maximum)
  nombre_coup = jouer_une_partie(nombre,minimum,maximum)
  print(f"Vous avez deviné le nombre en {nombre_coup} coups !")

jouer()
```

28.7 Listes

28.7.1 Exercice 1

See *Question*

```
# Faire un programme qui permet d'ajouter ou retirer
# des produits d'une liste de courses. Lorsque le
# programme demarre, l'utilisateur a le choix entre:
#         1- Afficher la liste de courses.
#         2- Ajouter un produit a la liste de courses (chaine
#  de caracteres ex: pomme)
#         3- Retirer un produit de la liste de course
#         4- Supprimer toute la liste de courses
#         5- Quitter le programme
# Tant que l'utilisateur ne saisit pas "quitter", on
# continue la boucle
# Si l'utilisateur tape 1 : on affiche la liste des
# produits
# Si l'utilisateur tape 2 : on lui demande le nom du
# produit à ajouter et on l'ajoute à la liste de courses
# Si l'utilisateur tape 3 : on lui demande le nom du
# produit à retirer et on le retire de la liste de
# courses
# Si l'utilisateur tape 4 : on vide la liste
# Si l'utilisateur tape 5: on affiche "Au revoir" et on
# quitte le programme
if __name__ == "__main__":
  import sys
  import os
  SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
  sys.path.append(os.path.dirname(SCRIPT_DIR))

from mytools.userinput import demander_saisie_nombre_borne

def say_goodbye():
  print("Au revoir")

def print_list_products(list_products):
  if len(list_products) == 0:
    print("La liste est vide")
  else:
    print(f"Produits : {', '.join(list_products)}")

def add_product(list_products):
  while True:
    product_to_add = input("Que voulez-vous ajouter à votre liste de courses ?
 ↪").lower()
```

```
    if product_to_add.isspace() or product_to_add == "":
      print("Merci d'entrer un produit")
      continue
    if (list_products.count(product_to_add.strip())) > 0:
      print(f"Le produit '{product_to_add}' est déja dans votre liste de␣
 ↪courses")
    else:
      list_products.append(product_to_add.strip())
      print(f"Le produit '{product_to_add}' a été ajouté à votre liste de␣
 ↪courses")
    break

def delete_product(list_products):
  while True:
    product_to_delete = input("Quel produit voulez-vous supprimer ?").lower()
    if product_to_delete.isspace() or product_to_delete == "":
      print("Merci d'entrer un produit")
      continue
    try:
      list_products.remove(product_to_delete.strip())
      print(f"Le produit '{product_to_delete.strip()}' a été supprimé de votre␣
 ↪liste de courses")
    except ValueError:
      print(f"Le produit '{product_to_delete.strip()}' n'a pas été trouvé dans␣
 ↪votre liste de courses")
    break

def delete_all_products(list_products):
  for product in list_products.copy():
    list_products.remove(product)
  print("Votre liste de course est désormais vide")

list_products = []
while True:
  user_input = demander_saisie_nombre_borne("""---------- Menu ----------
1- Afficher la liste de courses.
2- Ajouter un produit a la liste de courses
3- Retirer un produit de la liste de course
4- Supprimer toute la liste de courses
5- Quitter le programme
Votre choix ?""", 1, 5)
  if user_input == 5:
    say_goodbye()
    break
  if user_input == 1:
    print_list_products(list_products)
  if user_input == 2:
```

```
    add_product(list_products)
  if user_input == 3:
    delete_product(list_products)
  if user_input == 4:
    delete_all_products(list_products)
  input("Faites Entrée ")
```

28.7.2 Exercice 2

See *Question*

```
# Réduire ces algorithmes à l'aide, des listes en
# compréhension
# a)
nombres = [1, 21, 45, 12, 32, 65, 1002, 109, 83]
nombres_pairs = []
for i in nombres:
    if i % 2 == 0:
        nombres_pairs.append(i)
print(nombres_pairs)

nombres_pairs_comprehension = [nombre for nombre in nombres if nombre % 2 == 0]
print(nombres_pairs_comprehension)

# b)
nombres = range(-10, 10)
nombres_positifs = []
for i in nombres:
    if i >= 0:
        nombres_positifs.append(i)
print(nombres_positifs)

nombres_positifs_comprehension = [nombre for nombre in nombres if nombre >= 0]
print(nombres_positifs_comprehension)

# c)
nombres = range(5)
nombres_doubles = []
for i in nombres:
    nombres_doubles.append(i * 2)
print(nombres_doubles)

nombres_double_comprehension = [2*nombre for nombre in nombres]
print(nombres_double_comprehension)

# d)
```

```python
nombres = range(10)
nombres_inverses = []
for i in nombres:
    if i % 2 == 0:
        nombres_inverses.append(i)
    else:
        nombres_inverses.append(-i)
print(nombres_inverses)

nombres_inverses_comprehension = [nombre if nombre % 2 == 0 else -nombre for␣
 ↪nombre in nombres]
print(nombres_inverses_comprehension)
```

28.7.3 Exercice 3

See *Question*

```python
# Ecrire une fonction qui transformer une chaine de
# caracteres en accronyme
# Salut les    geeks -> SLG

import string
def accronyme(chaine: str):
  mots_avec_vides = chaine.split(' ')
  # opérateur ternaire : variable = 'truc' if nombre > 0 else 'machin'
  mots_non_vides = [mot for mot in mots_avec_vides if len(mot) != 0]
  initiales = [mot[0].upper() for mot in mots_non_vides]
  return "".join(initiales)

def accronyme_v2(chaine: str):
  accronyme_avec_ponctuation = accronyme(chaine)
  liste_initiales = [lettre for lettre in accronyme_avec_ponctuation if lettre␣
 ↪not in string.punctuation]
  return "".join(liste_initiales)

def accronyme_compact(chaine: str): # A vouloir être trop compact, on perd en␣
 ↪clarté
  return "".join([mot[0].upper() if len(mot) != 0 and mot[0] not in string.
 ↪punctuation else '' for mot in chaine.split(' ')])

print(f"{accronyme('Salut les    geeks             ') = }")
print(f"{accronyme('ceci est une phrase très longue ! ') = }")
print(f"{accronyme('Tout ! Est ! La ! ') = }")

print(f"{accronyme_v2('Salut les    geeks             ') = }")
print(f"{accronyme_v2('ceci est une phrase très longue ! ') = }")
print(f"{accronyme_v2('Tout ! Est ! La ! ') = }")
```

```
print(f"{accronyme_compact('Salut les    geeks             ') = }")
print(f"{accronyme_compact('ceci est une phrase très longue ! ') = }")
print(f"{accronyme_compact('Tout ! Est ! La ! ') = }")
```

28.8 Dictionnaires

28.8.1 Exercice 1

See *Question*

```
# Transformer la liste de courses
# La liste de courses reste une liste mais on y
# mettra des dictionnaires.
# Un dictionnaire contient: un nom de produit (str)
# et une quantité (int)
if __name__ == "__main__":
  import sys
  import os
  SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
  sys.path.append(os.path.dirname(SCRIPT_DIR))

from mytools.userinput import demander_saisie_nombre_borne,␣
 ↪demander_saisie_nombre_positif

def say_goodbye():
  print("Au revoir")

def print_list_products(list_products: list):
  if not list_products:
    print("La liste est vide")
  else:
    names = [f"{p['nom']} : {p['quantite']}x" for p in list_products]
    print(f"Produits : {', '.join(names)}")

def add_product_to_list(list_products: list, produit):
  for p in list_products:
    if p['nom'] == produit['nom']:
      p['quantite'] += produit['quantite']
      return
  list_products.append(produit)

def add_product(list_products: list):
  while True:
    product_to_add = input("Que voulez-vous ajouter à votre liste de courses ?
 ↪").lower()
    if product_to_add.isspace() or product_to_add == "":
      print("Merci d'entrer un produit")
      continue
    nbr_to_add = demander_saisie_nombre_positif("Combien voulez-vous en ajouter␣
 ↪?")
    produit = {
      'nom': product_to_add,
```

```
      'quantite': nbr_to_add
    }
    add_product_to_list(list_products, produit)
    break


def delete_product(list_products: list):
  while True:
    product_to_delete = input("Quel produit voulez-vous supprimer ?").lower()
    if product_to_delete.isspace() or product_to_delete == "":
      print("Merci d'entrer un produit")
      continue
    for p in list_products:
      if product_to_delete == p['nom']:
        number_to_delete = demander_saisie_nombre_borne("Combien voulez-vous␣
 ↪supprimer ?",0,p['quantite'])
        if (number_to_delete == p['quantite']):
          list_products.remove(p)
          return
        p['quantite'] -= number_to_delete
        return
    print("Le produit n'est pas dans la liste")
    break


def delete_all_products(list_products:list):
  list_products.clear()
  print("Votre liste de course est désormais vide")

list_products = []
while True:
  user_input = demander_saisie_nombre_borne("""__________ Menu __________
1- Afficher la liste de courses.
2- Ajouter un produit a la liste de courses
3- Retirer un produit de la liste de course
4- Supprimer toute la liste de courses
5- Quitter le programme
Votre choix ?""", 1, 5)
  if user_input == 5:
    say_goodbye()
    break
  if user_input == 1:
    print_list_products(list_products)
  if user_input == 2:
    add_product(list_products)
  if user_input == 3:
    delete_product(list_products)
```

```
  if user_input == 4:
    delete_all_products(list_products)
  input("Faites Entrée ")
```

28.8.2 Exercice 2

See *Question*

```
# Black jack simplifié. Créer un dictionnaire où les
# clés sont les cartes et les valeurs sont leur valeur
# en nombre de points
# (pour simplifier, l'as vaut forcément 11 points)
# Faire piocher une carte au joueur, lui proposer de
# continuer ou s'arrêter. Lorsqu'il s'arrête, lui
# donner son score. Si il dépasse les 21, il a perdu
# et a un score de 0.

import random

values = {"2":2,
"3":3,
"4":4,
"5":5,
"6":6,
"7":7,
"8":8,
"9":9,
"10":10,
"Jack":10,
"Queen":10,
"King":10,
"Ace":11}
cartes_values = {}
for suit in ("hearts","spades","clubs","diamonds"):
  for rank in values.keys():
    cartes_values[rank + " of " + suit] = values[rank]

liste_cartes = list(cartes_values)

def ask_to_take_another():
  while True:
    user_input = input("Take another ? Y/N ")
    if user_input.upper() in ("Y","N"):
      break
  return user_input

def play():
```

```
  score = 0
  while True:
    carte = random.choice(liste_cartes)
    liste_cartes.remove(carte)
    score += cartes_values[carte]
    print("Your card : " + carte)
    if score > 21 :
      score = 0
      break
    print("Your score : " + str(score))
    user_input = ask_to_take_another()
    if user_input.upper() == "N":
      break
  print(f"Game over ! Your score : {score}")

play()
```

28.9 Ensembles

28.9.1 Exercice 1

See *Question*

```
# Ecrire deux ensembles de prénoms
# Afficher les prénoms communs

al_friends = {"Josephine", "Meghan", "Amy", "Bob"}
kim_friends = {"Frank", "Amy", "Josephine", "Susan"}

mutual_friends = al_friends.intersection(kim_friends)
print(mutual_friends)
```

28.10 Strings

28.10.1 Exercice 1

See *Question*

```
# créer une méthode find_char(chaine, lettre) qui
# affiche "trouvé" ou "aucun résultat" selon si une
# lettre apparait dans une chaine de caractères

# Solution possible si on oublie l'opérateur "in"
# def find_char_bool(chaine: str, lettre):
#   result = False
#   for char in chaine.lower():
#     if char == lettre:
#       result = True
#       break
#   return result

# def find_char(chaine,lettre):
#   if (find_char_bool(chaine,lettre)):
#     print("Trouvée")
#   else:
#     print("Aucun résultat")

# def find_char(chaine,lettre):
#   print("Trouvée") if find_char_bool(chaine,lettre) else print("Aucun
 ↪résultat")

# Avec l'opérateur "in"
# def find_char(chaine, lettre):
#   if lettre.lower() in chaine.lower():
#     print("Trouvée")
#   else:
#     print("Aucun résultat")

def find_char(chaine,lettre):
  print("Trouvée") if lettre.lower() in chaine.lower() else print("Aucun
 ↪résultat")

print("find_char('Salut','u')")
find_char("Salut","u")

print("find_char('Salut','z')")
find_char("Salut","z")
```

28.10.2 Exercice 2

See *Question*

```python
# Créer une méthode annagramme(mot1: str, mot2: str)
# qui permet de vérifier si deux mots sont des
# annagrammes

# Solution alternative :
# def annagramme(mot1: str, mot2: str):
#   if len(mot1) != len(mot2):
#     return "Pas un annagramme"
#   for c in mot1:
#     if mot1.count(c) != mot2.count(c):
#       return "Pas un annagramme"
#   return "Ce sont des annagrammes"

# Solution alternative :
# def annagramme(mot1: str, mot2: str):
#   liste1, liste2 = list(mot1) , list(mot2)
#   liste1.sort()
#   liste2.sort()
#   return liste1 == liste2

def annagramme(mot1: str, mot2: str):
  return sorted(mot1) == sorted(mot2)

print(f"{annagramme('parisien','aspirine')=}")
print(f"{annagramme('parisien','aspirines')=}")
```

28.10.3 Exercice 3

See *Question*

```python
# Créer une méthode
# palindrome(mot, space_ignored = False)
# qui permet de vérifier si une chaine de caractères
# est un palindrome.
# Le paramètre space_ignored permet d'inclure ou
# non les espaces
# palindrome('kayak') = True
# palindrome('Esope reste ici et se repose') = False
# palindrome('Esope reste ici et se repose', True) = True
# palindrome('chien') = False

def remove_spaces(s: str) -> str:
  return "".join([x for x in s.split(" ") if len(x) > 0])

def inverser(mot: str) -> str:
  liste = list(mot)
  liste.reverse()
```

```
  return "".join(liste)

def is_reverse(mot1: str, mot2: str, space_ignored = False):
  if (space_ignored):
    mot1 = remove_spaces(mot1)
    mot2 = remove_spaces(mot2)
  mot1 = mot1.lower()
  mot2 = mot2.lower()
  return mot1 == inverser(mot2)

def palindrome(mot: str, space_ignored = False):
  return is_reverse(mot, mot, space_ignored)

print(f"{palindrome('kayak')=}")
print(f"{palindrome('Esope reste ici et se repose')=}")
print(f"{palindrome('Esope reste ici et se repose', True)=}")
print(f"{palindrome('kayaak')=}")
```

28.10.4 Exercice 4

See *Question*

```
# Créer une méthode rot(chaine, number) qui encode une
# chaine de caractères selon le code de César en
# utilisant le paramètre number.
# Code de césar : https://fr.wikipedia.org/wiki/Chiffrement_par_d%C3%A9calage
# Attention aux majuscules/minuscules !
# Pour s'aider : https://fr.wikibooks.org/wiki/Les_ASCII_de_0_%C3%A0_127/
 ↪La_table_ASCII#Descriptif3
# Exemple :
# rot("ABCD",3) = "DEFG"
# rot("Message secret",13) = "Zrffntr frperg"
# Pour décoder un message, il suffit d'utiliser
# rot(message_encode,26-ancien_number) Et donc :
# rot("DEFG",23) = "ABCD"
# rot("Zrffntr frperg",13) = "Message secret"

def rot_ascii_character(code, number, index_min, index_max):
  if (code < index_min or code > index_max):
    return code
  if (code+number>index_max):
    new_number_ascii = index_min + code + number - (index_max + 1)
  else:
    new_number_ascii = code + number
  return new_number_ascii

def rot_ascii_character_improved(code, number, index_min, index_max):
```

```
  if (code < index_min or code > index_max):
    return code
  return index_min + (code - index_min + number) % (index_max - index_min +1)

def rot_ascii_majuscule(code, number):
  return rot_ascii_character_improved(code, number, 65, 90)

def rot_ascii_minuscule(code, number):
  return rot_ascii_character_improved(code, number, 97, 122)

def rot(chaine, number):
  result=""
  for char in chaine:
    number_ascii = ord(char)
    if (number_ascii < 65) or (number_ascii > 90 and number_ascii < 97) or␣
 ↪(number_ascii > 122):
      result = result+char
    else:
      if number_ascii<=90: # majuscule
        new_number_ascii = rot_ascii_majuscule(number_ascii, number)
      else: #minuscule
        new_number_ascii = rot_ascii_minuscule(number_ascii, number)
      result = result + chr(new_number_ascii)
  return result

def rot11(chaine):
  return rot(chaine,11)

def rot15(chaine):
  return rot(chaine,15)

print(rot("ABCD",3))
print(rot("Message secret",13))
print(rot("DEFG",23))
print(rot("Zrffntr frperg",13))
```

28.10.5 Exercice 5

See *Question*

```
# Créer une méthode decimal_to_binary(decimal) qui
# converti un nombre décimal en un nombre binaire
# (renvoie une chaine de caractère)
# Créer une méthode binary_to_decimal(chaine: str)
# qui converti un nombre binaire (sous forme de
# chaine de caractères) en un nombre décimal
```

```python
def decimal_to_binary(decimal):
  if (decimal < 0):
    return "-" + decimal_to_binary(-decimal)
  if (decimal < 2):
    return str(decimal)
  return str(decimal_to_binary(decimal // 2)) +  str(decimal  % 2)

print(f"{decimal_to_binary(13)=}")
print(f"{decimal_to_binary(-27)=}")

def check_binary(chaine):
  if chaine[0] == "-":
    if (len(chaine) == 1):
      return False
    start_index = 1
  else:
    start_index = 0
  for char in chaine[start_index:]:
    if (char != "0" and char != "1"):
      return False
  return True

def binary_to_decimal_checked(chaine: str):
  if chaine[0] == "-":
    return -binary_to_decimal_checked(chaine[1:])
  if (len(chaine) == 1):
    return int(chaine[len(chaine)-1])
  return int(chaine[len(chaine)-1]) + 2 * binary_to_decimal_checked(chaine[:
 ↪len(chaine)-1])

def binary_to_decimal(chaine: str):
  if not check_binary(chaine):
    # Il faudrait plutôt créer notre propre classe
    # d'exception et lever une exception ici
    print("Ce n'est pas un nombre binaire")
  else:
    return binary_to_decimal_checked(chaine)

print(f"{binary_to_decimal('1101')=}")
print(f"{binary_to_decimal('-11011')=}")
print(f"{binary_to_decimal('12011')=}")
```

28.11 Classes

28.11.1 Exercice 1

See *Question*

```
# Créer un nouveau module dans le projet myclasses
# Y mettre un fichier product.py qui contient une
# classe Product qui est décrit par un id, une
# description, un prix, qui a des méthodes pour
# calculer la TVA d'un produit et le prix TTC
# (prix tva = 0.2 * prix,
# et prix ttc = prix + prix tva),
# un compteur du nombre total de produits instanciés,
# une méthode qui permet de print() correctement un
# produit, et une méthode qui compare des produits
# (deux produits sont les mêmes ssi ils ont le même id)

class Product:
  compteur = 0
  def __init__(self, id, description, prix):
    self.id = id
    self.description = description
    self.prix = prix
  def tva(self):
    return self.prix * 0.2
  def prixTTC(self):
    return self.prix + self.tva()
  def __str__(self):
    return f"id : {self.id} ; description : {self.description} ; price : {self.
 ↪prix}"
  def __eq__(self, other):
    return self.id == other.id

if __name__ == "__main__":
  p = Product(1,"PC Dell", 1200)
  print(p)
  p2 = Product(2,"PC HP", 1000)
  print(p2)
  print(f"{p} - TVA : {p.tva()} - prix TTC : {p.prixTTC()}")
  print(f"{p == p2 = }")
```

28.12 POO

28.12.1 Encapsulation

Exercice 1 See *Question*

```
# Réécrire la classe Product en respectant les bonnes
# pratiques d'encapsulation des champs dans des
# accesseurs en lecture et en écriture
# Vous pouvez renomer l'ancien fichier product.py en
# product_naive.py pour garder une trace du premier jet

class Product:
  compteur = 0
  def __init__(self, id, description, prix):
    self.__id = id
    self.__description = description
    self.__prix = prix

  def __get_id(self):
    return self.__id
  def __set_id(self, value):
    if id < 0:
      raise ValueError("Id negatif !")
    self.__id = value

  def __get_description(self):
    return self.__description
  def __set_description(self, value):
    self.__description = value

  def __get_prix(self):
    return self.__prix
  def __set_prix(self, value):
    if value < 0:
      raise ValueError("Prix negatif !")
    self.__prix = value

  id = property(__get_id,__set_id)
  description = property(__get_description,__set_description)
  prix = property(__get_prix,__set_prix)

  def tva(self):
    return self.prix * 0.2
  def prixTTC(self):
    return self.prix + self.tva()
  def __str__(self):
```

```
    return f"id : {self.id} ; description : {self.description} ; prix : {self.
 ↪prix}"
  def __eq__(self, other):
    return self.id == other.id

if __name__ == "__main__":
  p = Product(1,"PC Dell", 1200)
  print(p)
  p2 = Product(2,"PC HP", 1000)
  print(p2)
  print(f"{p} - prix TVA : {p.tva()} - prix TTC : {p.prixTTC()}")
  print(f"{p == p2 = }")
  p.prix = -4
```

28.12.2 Héritage

Exercice 1 See *Question*

```
# Créer deux classes d'exception personnalisées
# correspondant à la tentative de mettre une valeur
# négative pour le prix ou l'id d'un produit et adapter
# le code de la classe Product en conséquence

class IdNegatifException(Exception):
    """Exception levée lorsqu'on tente de mettre un id négatif"""
    def __init__(self, message):
        self.message = message

    def __str__(self):
        return self.message

class PrixNegatifException(Exception):
    """Exception levée lorsqu'on tente de mettre un prix négatif"""
    def __init__(self, message):
        self.message = message

    def __str__(self):
        return self.message

class Product:
  compteur = 0
  def __init__(self, id, description, prix):
    self.__id = id
    self.__description = description
    self.__prix = prix

  def __get_id(self):
```

```
    return self.__id
  def __set_id(self, value):
    if id < 0:
      raise IdNegatifException("Id negatif !")
    self.__id = value

  def __get_description(self):
    return self.__description
  def __set_description(self, value):
    self.__description = value

  def __get_prix(self):
    return self.__prix
  def __set_prix(self, value):
    if value < 0:
      raise PrixNegatifException("Prix negatif !")
    self.__prix = value

  id = property(__get_id,__set_id)
  description = property(__get_description,__set_description)
  prix = property(__get_prix,__set_prix)

  def tva(self):
    return self.prix * 0.2
  def prixTTC(self):
    return self.prix + self.tva()
  def __str__(self):
    return f"id : {self.id} ; description : {self.description} ; prix : {self.
↪prix}"
  def __eq__(self, other):
    return self.id == other.id
```

28.13 Files

28.13.1 Files txt

Exercice 1 See *Question*

```
# Créer une méthode read_file_txt(path) qui renvoie
# un string contenant le contenu d'un fichier au
# chemin path (à partir du dossier de travail actuel)

import os

def read_file_txt(path):
  cwd = os.getcwd()
  file_path = cwd + path
  with open(file_path, 'r', encoding="utf-8") as fichier:
    contenu = fichier.read()
  return contenu

if __name__ == "__main__":
  print(read_file_txt("/myfile.txt"))
```

Exercice 2 See *Question*

```
# Créer une méthode write_file_txt(path, content) qui
# écrit le contenu du string content dans un fichier au
# chemin path (le créant / l'écrasant éventuellement)

import os

def write_file_txt(path, content):
  cwd = os.getcwd()
  file_path = cwd + path
  with open(file_path, 'w', encoding="utf-8") as fichier:
    fichier.write(content)

if __name__ == "__main__":
  write_file_txt("/myfile2.txt","Ceci est un test de la méthode write_file_txt")
```

28.13.2 Files json

Exercice 1 See *Question*

```
# Créer une méthode read_file_json(path) qui renvoie
# un string contenant le contenu d'un fichier au
# chemin path (à partir du dossier de travail actuel)
# la mettre dans le module mytools, dans files.py
```

```
import os
import json

def read_file_json(path):
  cwd = os.getcwd()
  file_path = cwd + path
  with open(file_path, 'r', encoding="utf-8") as fichier:
    contenu = json.load(fichier)
  return contenu

if __name__ == "__main__":
  print(read_file_json("/sortie.json"))
```

Exercice 2 See *Question*

```
# Créer une méthode write_file_json(path, content) qui
# écrit le contenu du dictionnaire content dans un
# fichier au chemin path (le créant / l'écrasant
# éventuellement)
# la mettre dans le module mytools, dans files.py

import os
import json

def write_file_json(path, content):
  cwd = os.getcwd()
  file_path = cwd + path
  with open(file_path, 'w', encoding="utf-8") as fichier:
    json.dump(content, fichier, indent=2, ensure_ascii=False)

if __name__ == "__main__":
  write_file_json("/sortie2.json",[ {'test' : "truc", "test2" : "truc2"},␣
 ↪{'test' : 'chose', 'test2' : 'chose2'}])
```

28.13.3 Files csv

Exercice 1 See *Question*

```
# Modifier le programme de gestion de listes de courses
# en ajouter des possibilités d'importer/d'exporter une liste de courses
# en json et en csv
# (demander à l'utilisateur d'écrire à la main le chemin vers le fichier)

if __name__ == "__main__":
  import sys
  import os
  SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
```

```
  sys.path.append(os.path.dirname(SCRIPT_DIR))

from json.decoder import JSONDecodeError
from mytools.userinput import demander_saisie_nombre_borne,␣
 ↪demander_saisie_nombre_positif, demander_saisie_chaine_definie
from mytools.files import read_file_json, write_file_json, read_file_csv,␣
 ↪write_file_csv

def afficher_liste():
  if not liste_produit:
    print("La liste est vide")
  else:
    print(liste_produit)

def ajouter_produit():
  name_to_add = demander_saisie_chaine_definie("Quel produit voulez-vous␣
 ↪ajouter ?").lower()
  quantity_to_add = demander_saisie_nombre_positif("Combien voulez-vous en␣
 ↪ajouter ?")
  for produit in liste_produit:
    if produit['nom'] == name_to_add:
      produit['quantite'] += quantity_to_add
      print("Produit mis à jour")
      return
  product_to_add = {
    "nom":name_to_add,
    "quantite":quantity_to_add,
  }
  liste_produit.append(product_to_add)
  print("Le produit a été ajouté à la liste")


def retirer_produit():
  name_to_delete = demander_saisie_chaine_definie("Quel produit voulez-vous␣
 ↪supprimer ?").lower()
  for produit in liste_produit:
    if produit['nom'] == name_to_delete:
      quantity_to_remove = demander_saisie_nombre_borne("Combien voulez-vous en␣
 ↪retirer ?",0,produit["quantite"])
      if quantity_to_remove == produit["quantite"]:
        # product_to_remove = produit
        liste_produit.remove(produit)
        break
      else:
        produit['quantite'] -= quantity_to_remove
        break
```

```
  # if product_to_remove:
  #   liste_produit.remove(product_to_remove)


def supprimer_stock():
  liste_produit.clear()
  print("Liste de courses supprimée")
  # for produit in liste_produit:
  #   liste_produit.remove(produit)
  # attention à ne pas modifier la liste sur laquelle on boucle
  # ça peut mener à des comportements non attendus

def au_revoir():
  print("Au revoir")

def export_json():
  path = input("Chemin vers le fichier ?")
  write_file_json(path, liste_produit)

def export_csv():
  path = input("Chemin vers le fichier")
  write_file_csv(path, convert_to_list_of_string(liste_produit))

def import_json():
  try:
    path = input("Chemin vers le fichier ?")
    imported = read_file_json(path)
    liste_produit.clear()
    liste_produit.extend(imported)
  except JSONDecodeError:
    print("Erreur d'import. Format du fichier incorrect")
  except FileNotFoundError:
    print("Fichier introuvable")

def import_csv():
  try:
    path = input("Chemin vers le fichier ?")
    imported = read_file_csv(path)
    liste_produit.clear()
    liste_produit.extend(convert_to_list_of_products(imported))
  except (IndexError, ValueError):
    print("Erreur d'import. Format du fichier incorrect")
  except FileNotFoundError:
    print("Fichier introuvable")

def convert_to_list_of_string(liste):
  result = []
```

```
  for product in liste:
    row = []
    row.append(product['nom'])
    row.append(str(product['quantite']))
    result.append(row)
  return result

def convert_to_list_of_products(liste):
  result = []
  for row in liste:
    product = {
      'nom':row[0],
      'quantite':int(row[1]),
    }
    result.append(product)
  return result

liste_produit = [] # list()
while True:
  user_input = demander_saisie_nombre_borne("""---------- Menu ----------
1- Afficher la liste de courses.
2- Ajouter un produit a la liste de courses
3- Retirer un produit de la liste de course
4- Supprimer toute la liste de courses
5- Exporter la liste de courses (JSON)
6- Exporter la liste de courses (CSV)
7- Importer la liste de courses (JSON)
8- Importer la liste de courses (CSV)
9- Quitter le programme
Votre choix ?""",1,9)
  if user_input == 9:
    au_revoir()
    break
  if user_input == 1:
    afficher_liste()
  if user_input == 2:
    ajouter_produit()
  if user_input == 3:
    retirer_produit()
  if user_input == 4:
    supprimer_stock()
  if user_input == 5:
    export_json()
  if user_input == 6:
    export_csv()
  if user_input == 7:
    import_json()
```

```
  if user_input == 8:
    import_csv()
  input("Appuyer sur entrée")
```

28.14 BDD

28.14.1 Delete

Exercice 1 See *Question*

```
# Créer un module repository, avec un fichier
# productDao.py
# Coder une classe ProductDao qui contiendra des
# méthodes
# addProduct(self, p)
# getProductById(self, id)
# getAllProducts(self)
# updateProduct(self, p)
# deleteProductById(self, id)
# qui implémentent les fonctionnalités associées pour
# une table produit et qui a comme propriétés le nom de
# la base, le nom de la table.
# Enfin, elle aura une méthode close() pour fermer la
# connexion et une méthode init_table() pour créer la
# table et y mettre des données initiales.
# Les opérations / erreurs seront loggées dans un
# fichier app.log

import sqlite3
import logging

if __name__ == "__main__":
  import sys
  import os
  SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
  sys.path.append(os.path.dirname(SCRIPT_DIR))

from myclasses.product import Product


logging.basicConfig(
    level=logging.INFO,
    filename="app.log",
    filemode='a',
    format="%(asctime)s - [%(levelname)s]: %(filename)s: %(message)s",
    datefmt="%d/%m/%Y %H:%M:%S",
    encoding="utf-8"
)

class ProductDao:
  DB_NAME = "repository/db.sqlite3"
  TABLE_NAME = "product"
```

```
 def __init__(self):
   self.__cnx = None
   try:
     self.__cnx = sqlite3.connect(ProductDao.DB_NAME)
   except sqlite3.DatabaseError as e:
     logging.error(e)

 def init_table(self):
   cursor = self.__cnx.cursor()
   logging.info(f"Création de la table {ProductDao.TABLE_NAME} (si elle␣
↪n'existe pas)")
   try:
     sql = "CREATE TABLE IF NOT EXISTS " + ProductDao.TABLE_NAME
     sql += """ (
     id INTEGER PRIMARY KEY AUTOINCREMENT,
     description varchar(50) NOT NULL,
     prix real NOT NULL
     )"""
     cursor.execute(sql)
     # equivalent de truncate (remet le compteur d'id à 0)
     # https://sqlite.org/autoinc.html
     sql = "DELETE FROM " + ProductDao.TABLE_NAME
     cursor.execute(sql)
     sql = "delete from sqlite_sequence where name='" + ProductDao.TABLE_NAME␣
↪+ "'"
     cursor.execute(sql)
     sql = "INSERT INTO " + ProductDao.TABLE_NAME + " (id, description, prix)␣
↪VALUES (NULL, :description, :prix)"
     p1 = Product(1,"Table",50)
     p2 = Product(2, "Crayon",1.5)
     p3 = Product(3,"PC",2500)
     # p.__dict__ : dictionnaire avec tous les
     # attributs de l'objet p
     # ne fonctionne pas pour les propriétés
     # cursor.execute(sql, p1.__dict__)
     cursor.execute(sql, {"id" : p1.id, "description" : p1.description, "prix":
↪ p1.prix})
     cursor.execute(sql, {"id" : p2.id, "description" : p2.description, "prix":
↪ p2.prix})
     cursor.execute(sql, {"id" : p3.id, "description" : p3.description, "prix":
↪ p3.prix})
     self.__cnx.commit()
   except sqlite3.Error as e:
     logging.error(e)
   else:
```

```
      logging.info(f"Succès de la création de la table {ProductDao.TABLE_NAME}␣
↪(si elle n'existe pas)")
    finally:
      cursor.close()

  def close(self):
    self.__cnx.close()

  # méthode utilisée lorsque la variable est
  # supprimée par le ramasse miettes
  def __del__(self):
    self.close()

  def addProduct(self, p: Product):
    logging.info(f"Ajout du produit {p}")
    cursor = self.__cnx.cursor()
    sql = "INSERT INTO " + ProductDao.TABLE_NAME + " (id, description, prix)␣
↪VALUES (null, :description, :prix)"
    try:
      cursor.execute(sql, {"id" : p.id, "description" : p.description, "prix":␣
↪p.prix})
      self.__cnx.commit()
    except sqlite3.Error as e:
      logging.error(e)
    else:
      logging.info(f"Succès de l'ajout du produit {p}")
    finally:
      cursor.close()

  def getProductById(self, id: int) -> Product:
    logging.info(f"Récupération du produit avec l'id {id}")
    sql = f"SELECT * FROM {ProductDao.TABLE_NAME} WHERE id = ?"
    product = None
    try:
      product = self.__cnx.execute(sql,(id,)).fetchall()
    except sqlite3.Error as e:
      logging.error(e)
    else:
      if product:
        # https://sametmarx.com/operateur-splat-ou-etoile-en-python/
        # https://treyhunner.com/2018/10/
↪asterisks-in-python-what-they-are-and-how-to-use-them/
        product = Product(*product[0])
      logging.info(f"Le produit trouvé : {product}")
    return product

  def getAllProducts(self):
```

```
    logging.info(f"Récupération de tous les produits")
    sql = f"SELECT * FROM {ProductDao.TABLE_NAME}"
    try:
      result = self.__cnx.execute(sql).fetchall()
    except sqlite3.Error as e:
      logging.error(e)
    else:
      logging.info(f"Succès de récupération de tous les produits")
      return result


  def updateProduct(self, p: Product):
    logging.info(f"Update du produit avec l'id {p.id}")
    sql = "UPDATE " + ProductDao.TABLE_NAME + " SET description=:description,␣
 ↪prix=:prix WHERE id=:id"
    cursor = self.__cnx.cursor()
    try:
      cursor.execute(sql, {"id" : p.id, "description" : p.description, "prix":␣
 ↪p.prix})
      self.__cnx.commit()
    except sqlite3.Error as e:
      logging.error(e)
    else:
      pass
    finally:
      cursor.close()

  def deleteProductById(self, id: int):
    logging.info(f"Suppression du produit avec l'id {id}")
    sql = f"DELETE FROM {ProductDao.TABLE_NAME} WHERE id = ?"
    product = None
    cursor = self.__cnx.cursor()
    try:
      cursor.execute(sql,(id,))
      self.__cnx.commit()
    except sqlite3.Error as e:
      logging.error(e)
    else:
      logging.info(f"Succès de la suppression du produit avec l'id {id}")
    finally:
      cursor.close()
    return product

if __name__ == "__main__":
  dao = ProductDao()
  print("____init_table____")
  dao.init_table()
```

```
print("____getAllProducts____")
print(dao.getAllProducts())

print("____getProductById(2)____")
print(dao.getProductById(2))

print("____addProduct(Product(19,'Test', 1999))____")
product = Product(19,"Test", 1999)
dao.addProduct(product)
print(dao.getAllProducts())

print("____deleteProductById(4)____")
dao.deleteProductById(4)
print(dao.getAllProducts())

print("____updateProduct(Product(2,'Edited', 9999))____")
product = Product(2,"Edited", 9999)
dao.updateProduct(product)
print(dao.getAllProducts())

dao.close()
```

28.15 TP

28.15.1 Etape 1

See *Question*

```
if __name__ == "__main__":
  import sys
  import os
  SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
  sys.path.append(os.path.dirname(SCRIPT_DIR))

from mytools.userinput import demander_saisie_nombre_borne

def say_goodbye():
  print("Au revoir")

def print_list_products(list_products):
  if len(list_products) == 0:
    print("La liste est vide")
  else:
    print(f"Produits : {', '.join(list_products)}")

def add_product(list_products):
  while True:
    product_to_add = input("Que voulez-vous ajouter à votre liste de courses ?
 ↪").lower()
    if product_to_add.isspace() or product_to_add == "":
      print("Merci d'entrer un produit")
      continue
    if (list_products.count(product_to_add.strip())) > 0:
      print(f"Le produit '{product_to_add}' est déja dans votre liste de␣
 ↪courses")
    else:
      list_products.append(product_to_add.strip())
      print(f"Le produit '{product_to_add}' a été ajouté à votre liste de␣
 ↪courses")
    break

def delete_product(list_products):
  while True:
    product_to_delete = input("Quel produit voulez-vous supprimer ?").lower()
    if product_to_delete.isspace() or product_to_delete == "":
      print("Merci d'entrer un produit")
      continue
    try:
      list_products.remove(product_to_delete.strip())
```

```
      print(f"Le produit '{product_to_delete.strip()}' a été supprimé de votre␣
 ↪liste de courses")
    except ValueError:
      print(f"Le produit '{product_to_delete.strip()}' n'a pas été trouvé dans␣
 ↪votre liste de courses")
    break

def delete_all_products(list_products):
  for product in list_products.copy():
    list_products.remove(product)
  print("Votre liste de course est désormais vide")

list_products = []
while True:
  user_input = demander_saisie_nombre_borne("""---------- Menu ----------
1- Afficher la liste de courses.
2- Ajouter un produit a la liste de courses
3- Retirer un produit de la liste de course
4- Supprimer toute la liste de courses
5- Quitter le programme
Votre choix ?""", 1, 5)
  if user_input == 5:
    say_goodbye()
    break
  if user_input == 1:
    print_list_products(list_products)
  if user_input == 2:
    add_product(list_products)
  if user_input == 3:
    delete_product(list_products)
  if user_input == 4:
    delete_all_products(list_products)
  input("Faites Entrée ")
```

28.15.2 Etape 2

See *Question*

```
if __name__ == "__main__":
  import sys
  import os
  SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
  sys.path.append(os.path.dirname(SCRIPT_DIR))

from mytools.userinput import demander_saisie_nombre_borne,␣
 ↪demander_saisie_nombre_positif
```

```
def say_goodbye():
  print("Au revoir")

def print_list_products(list_products: list):
  if not list_products:
    print("La liste est vide")
  else:
    names = [f"{p['nom']} : {p['quantite']}x" for p in list_products]
    print(f"Produits : {', '.join(names)}")

def add_product_to_list(list_products: list, produit):
  for p in list_products:
    if p['nom'] == produit['nom']:
      p['quantite'] += produit['quantite']
      return
  list_products.append(produit)

def add_product(list_products: list):
  while True:
    product_to_add = input("Que voulez-vous ajouter à votre liste de courses ?
 ↪").lower()
    if product_to_add.isspace() or product_to_add == "":
      print("Merci d'entrer un produit")
      continue
    nbr_to_add = demander_saisie_nombre_positif("Combien voulez-vous en ajouter␣
 ↪?")
    produit = {
      'nom': product_to_add,
      'quantite': nbr_to_add
    }
    add_product_to_list(list_products, produit)
    break


def delete_product(list_products: list):
  while True:
    product_to_delete = input("Quel produit voulez-vous supprimer ?").lower()
    if product_to_delete.isspace() or product_to_delete == "":
      print("Merci d'entrer un produit")
      continue
    for p in list_products:
      if product_to_delete == p['nom']:
        number_to_delete = demander_saisie_nombre_borne("Combien voulez-vous␣
 ↪supprimer ?",0,p['quantite'])
        if (number_to_delete == p['quantite']):
          list_products.remove(p)
```

```
            return
        p['quantite'] -= number_to_delete
        return
    print("Le produit n'est pas dans la liste")
    break


def delete_all_products(list_products:list):
  list_products.clear()
  print("Votre liste de course est désormais vide")

list_products = []
while True:
  user_input = demander_saisie_nombre_borne("""__________ Menu __________
1- Afficher la liste de courses.
2- Ajouter un produit a la liste de courses
3- Retirer un produit de la liste de course
4- Supprimer toute la liste de courses
5- Quitter le programme
Votre choix ?""", 1, 5)
  if user_input == 5:
    say_goodbye()
    break
  if user_input == 1:
    print_list_products(list_products)
  if user_input == 2:
    add_product(list_products)
  if user_input == 3:
    delete_product(list_products)
  if user_input == 4:
    delete_all_products(list_products)
  input("Faites Entrée ")
```

28.15.3 Etape 7

See *Question*

```
from repository.productDao import ProductDao
from myclasses.product import Product


# dao = ProductDao()
# print("____init_table____")
# dao.init_table()

# print("____getAllProducts____")
# print(dao.getAllProducts())
```

```
# print("____getProductById(2)____")
# print(dao.getProductById(2))

# print("____addProduct(Product(19,'Test', 1999))____")
# product = Product(19,"Test", 1999)
# dao.addProduct(product)
# print(dao.getAllProducts())

# print("____deleteProductById(4)____")
# dao.deleteProductById(4)
# print(dao.getAllProducts())

# print("____updateProduct(Product(2,'Edited', 9999))____")
# product = Product(2,"Edited", 9999)
# dao.updateProduct(product)
# print(dao.getAllProducts())

# dao.close()

from gui_console.choices import chose_admin_or_user
from gui_console.admin import gerer_produit
from gui_console.user import gerer_panier

def main():
  choice = chose_admin_or_user()
  if not choice:
    return False
  if choice == 'admin':
    gerer_produit()
  if choice == 'user':
    gerer_panier()
  return True

while main():
  pass
```

gui_console.choices :

```
def chose_admin_or_user():
  while True:
    choice = input("Admin, user or quit ? (A/U/Q)")
    if choice.lower() == 'a':
      return "admin"
    if choice.lower() == 'u':
      return "user"
    if choice.lower() == 'q':
```

```
      return ""
```

gui_console.admin :

```
from myclasses.product import Product
from mytools.userinput import demander_saisie_nombre_borne,␣
 ↪demander_saisie_nombre_positif
from repository.productDao import ProductDao

def print_products(dao: ProductDao):
  products = dao.getAllProducts()
  if not products:
    return print("No products in db")
  for id,description,prix in products:
    print(f"{id = }")
    print(f"\t{description = }")
    print(f"\t{prix = }")

def add_product(dao: ProductDao):
  description = input("Description ? ")
  prix = demander_saisie_nombre_positif("Prix ?")
  product = Product(0,description,prix)
  dao.addProduct(product)

def delete_product(dao: ProductDao):
  id_to_delete = demander_saisie_nombre_positif("id to delete ? ")
  dao.deleteProductById(id_to_delete)

def edit_product(dao:ProductDao):
  id_to_update = demander_saisie_nombre_positif("id to update ? ")
  description = input("New description ?")
  prix = demander_saisie_nombre_positif("New price ?")
  product = Product(id_to_update,description,prix)
  dao.updateProduct(product)

def gerer_produit():
  dao = ProductDao()
  while True:
    user_input = demander_saisie_nombre_borne("""_____ Menu _____
1- Afficher la liste des produits.
2- Ajouter un produit
3- Supprimer un produit
4- Editer un produit
5- Quitter
Votre choix ?""",1,5)
    if user_input == 5:
      dao.close()
```

```
      break
    if user_input == 1:
      print_products(dao)
    if user_input == 2:
      add_product(dao)
    if user_input == 3:
      delete_product(dao)
    if user_input == 4:
      edit_product(dao)
    input("Press enter")
```

gui_console.user :

```
from json.decoder import JSONDecodeError
from mytools.userinput import demander_saisie_nombre_borne,␣
 ↪demander_saisie_nombre_positif
from repository.productDao import ProductDao
from myclasses.cart import Cart
from myclasses.cartline import CartLine
from mytools.files import read_file_json, read_file_csv, write_file_json,␣
 ↪write_file_csv

def afficher_liste(panier):
  print(panier)

def ajouter_produit(dao:ProductDao,panier:Cart):
  id_to_add = demander_saisie_nombre_positif("Id ? ")
  p = dao.getProductById(id_to_add)
  if not p:
    print("Produit introuvable")
    return
  quantity_to_add = demander_saisie_nombre_positif("Combien voulez-vous en␣
 ↪ajouter ?")
  panier.add_product(p,quantity_to_add)
  print("Le produit a été ajouté à la liste")


def retirer_produit(dao:ProductDao, panier:Cart):
  id_to_delete = demander_saisie_nombre_positif("Id ? ")
  p = dao.getProductById(id_to_delete)
  if (not p) or (not CartLine(p,1) in panier.lines) :
    print("Ce produit n'est pas dans votre panier")
    return
  quantity_to_delete = demander_saisie_nombre_positif("Combien voulez-vous en␣
 ↪supprimer ?")
  panier.remove_product(p,quantity_to_delete)
  print("Le produit a été supprimé de la liste")
```

```
def supprimer_stock(panier:Cart):
  panier.clear()
  print("Liste de courses supprimée")

def au_revoir():
  print("Au revoir")

def export_json(panier: Cart):
  path = input("Chemin vers le fichier ?")
  write_file_json(path, panier.to_list_dict())

def export_csv(panier: Cart):
  path = input("Chemin vers le fichier")
  write_file_csv(path, panier.to_list_list_string())

def import_json(panier: Cart):
  try:
    path = input("Chemin vers le fichier ?")
    imported = read_file_json(path)
    return Cart.from_json(imported)
  except JSONDecodeError:
    print("Erreur d'import. Format du fichier incorrect")
  except FileNotFoundError:
    print("Fichier introuvable")
  return panier

def import_csv(panier: Cart):
  try:
    path = input("Chemin vers le fichier ?")
    imported = read_file_csv(path)
    return Cart.from_csv(imported)
  except (IndexError, ValueError):
    print("Erreur d'import. Format du fichier incorrect")
  except FileNotFoundError:
    print("Fichier introuvable")
  return panier

def gerer_panier():
  dao = ProductDao()
  panier = Cart([])
  while True:
    for id,description,prix in dao.getAllProducts():
      print(f"{id = }")
      print(f"\t{description = }")
      print(f"\t{prix = }")
    user_input = demander_saisie_nombre_borne("""_____ Menu _____
```

```
1- Afficher la liste de courses.
2- Ajouter un produit a la liste de courses
3- Retirer un produit de la liste de course
4- Supprimer toute la liste de courses
5- Exporter la liste de courses (JSON)
6- Exporter la liste de courses (CSV)
7- Importer la liste de courses (JSON)
8- Importer la liste de courses (CSV)
9- Quitter le programme
Votre choix ?""",1,9)
    if user_input == 9:
      au_revoir()
      break
    if user_input == 1:
      afficher_liste(panier)
    if user_input == 2:
      ajouter_produit(dao,panier)
    if user_input == 3:
      retirer_produit(dao,panier)
    if user_input == 4:
      supprimer_stock(panier)
    if user_input == 5:
      export_json(panier)
    if user_input == 6:
      export_csv(panier)
    if user_input == 7:
      panier = import_json(panier)
    if user_input == 8:
      panier = import_csv(panier)
    input("Appuyer sur entrée")
```

myclasses.product

```
from myclasses.myexceptions import IdNegatifException, PrixNegatifException
from typing import List

class Product:
  compteur = 0
  def __init__(self, id, description, prix):
    self.__set_id(id)
    self.__set_description(description)
    self.__set_prix(prix)

  def __get_id(self):
    return self.__id
  def __set_id(self, value):
    if value < 0:
```

```
      raise IdNegatifException("Id negatif !")
    self.__id = value

  def __get_description(self):
    return self.__description
  def __set_description(self, value):
    self.__description = value

  def __get_prix(self):
    return self.__prix
  def __set_prix(self, value):
    if value < 0:
      raise PrixNegatifException("Prix negatif !")
    self.__prix = value

  id = property(__get_id,__set_id,None,"""""")
  description = property(__get_description,__set_description,None,"""""")
  prix = property(__get_prix,__set_prix,None,"""""")

  def TVA(self):
    return self.prix * 0.2
  def PrixTTC(self):
    return self.prix + self.TVA()

  def to_dict(self):
    return {
      "id":self.id,
      "description":self.description,
      "prix":self.prix
    }

  def to_list_string(self):
    return [str(self.id),self.description,str(self.prix)]

  @staticmethod
  def from_json(dict: dict):
    return Product(int(dict["id"]), dict["description"], float(dict["prix"]))

  @staticmethod
  def from_csv(line: List[str]):
    return Product(int(line[0]),line[1],float(line[2]))


  def __str__(self):
    return f"id : {self.id} ; description : {self.description} ; prix : {self.
↪prix}"
  def __eq__(self, other):
```

```
    return self.id == other.id

if __name__ == "__main__":
  p = Product(1,"PC Dell", -1200)
  print(p)
  p2 = Product(2,"PC HP", 1000)
  print(p2)
  print(f"{p} - TVA : {p.TVA()} - prix TTC : {p.PrixTTC()}")
  print(f"{p == p2 = }")
  p.prix = -4
```

myclasses.cart

```
from myclasses.product import Product
from typing import List
from myclasses.cartline import CartLine
from myclasses.myexceptions import ProductNotInCartException

class Cart:
  def __init__(self,lines: List[CartLine]):
    self.__set_lines(lines)

  def __get_lines(self):
    return self.__lines
  def __set_lines(self, value):
    self.__lines = value

  lines: List[CartLine] = property(__get_lines,__set_lines)

  def get_total_price(self):
    return sum([line.get_price() for line in self.lines])

  def add_product(self,p:Product, qty: int):
    if not CartLine(p,1) in self.lines:
      self.lines.append(CartLine(p,qty))
    else:
      index = self.lines.index(CartLine(p,1))
      self.lines[index].add(qty)

  def remove_product(self,p:Product, qty: int):
    if not CartLine(p,1) in self.lines:
      raise ProductNotInCartException("Product not found")
    else:
      index = self.lines.index(CartLine(p,1))
      if qty >= self.lines[index].quantity:
        self.lines.pop(index)
      else:
```

```
        self.lines[index].remove(qty)

  def clear(self):
    self.lines.clear()

  def to_list_dict(self):
    return [line.to_dict() for line in self.lines]

  def to_list_list_string(self):
    return [line.to_list_string() for line in self.lines]

  @staticmethod
  def from_json(liste: List[dict]):
    lines = [CartLine.from_json(l) for l in liste]
    return Cart(lines)

  @staticmethod
  def from_csv(liste: List[List[str]]):
    lines = [CartLine.from_csv(l) for l in liste]
    return Cart(lines)

  def __str__(self):
    if self.lines:
      return "\n".join([str(line) for line in self.__lines])+f"\nTotal : {self.
↪get_total_price()} €"
    return "Empty cart"
```

myclasses.cartline

```
from myclasses.product import Product
from myclasses.myexceptions import NegativeQuantityException,␣
 ↪ZeroQuantityException
from typing import List


class CartLine:
  def __init__(self,product: Product,quantity: int):
    self.__set_product(product)
    self.__set_quantity(quantity)

  def __get_product(self):
    return self.__product
  def __set_product(self, value):
    self.__product = value

  def __get_quantity(self):
    return self.__quantity
```

```
def __set_quantity(self, value):
  if value < 0:
    raise NegativeQuantityException("Can't have a negative amount in Cart")
  if value == 0:
    raise ZeroQuantityException("Can't have zero amount in Cart")
  self.__quantity = value

product: Product = property(__get_product,__set_product)
quantity: int = property(__get_quantity,__set_quantity)

def remove(self, qty:int):
  self.quantity -= qty

def add(self, qty: int):
  self.quantity += qty

def get_price(self):
  p:Product = self.product
  return p.prix * self.quantity

def to_dict(self):
  return {
    "product":self.product.to_dict(),
    "quantity":self.quantity
  }

def to_list_string(self):
  resultat = self.product.to_list_string()
  resultat.append(str(self.quantity))
  return resultat

@staticmethod
def from_json(dict: dict):
  return CartLine(
      Product.from_json(dict["product"]),
      int(dict["quantity"]))

@staticmethod
def from_csv(line: List[str]):
  return CartLine(
      Product.from_csv(line),
      int(line[3])
    )

def __str__(self):
  p:Product = self.product
```

```
    return f"({p.id}) {p.description} - {self.quantity} x : total {self.
↪get_price()} €"

  def __eq__(cl1, cl2):
    return cl1.product == cl2.product
```

myclasses.myexceptions

```
class CustomException(Exception):
  def __init__(self, message):
    super().__init__(message)
    self.message = message

  def __str__(self):
    return self.message

class IdNegatifException(CustomException):
  """Exception levée lorsqu'on tente de mettre un id négatif"""

class PrixNegatifException(CustomException):
  """Exception levée lorsqu'on tente de mettre un prix négatif"""

class NegativeQuantityException(CustomException):
  pass

class ProductNotInCartException(CustomException):
  pass

class ZeroQuantityException(CustomException):
  pass
```

mytools.userinput

```
def demander_saisie_nombre(invite):
  while True:
    user_input = input(invite)
    try:
      result = int(user_input)
      return result
    except ValueError:
      print("Seul les caractères [0-9] sont autorisés")

def demander_saisie_nombre_borne(invite,minimum = 1, maximum = 10):
  invite += f" entre {minimum} et {maximum} "
  while True:
    nombre_input = demander_saisie_nombre(invite)
    if minimum <= nombre_input <= maximum:
      return nombre_input
```

```
def demander_saisie_nombre_positif(invite):
  invite += " (positif) "
  while True:
    nombre_input = demander_saisie_nombre(invite)
    if nombre_input >= 0:
      return nombre_input

def demander_saisie_chaine_definie(invite):
  while True:
    saisie = input(invite)
    if len(saisie.strip()) != 0:
      return saisie.strip()
    print("Merci d'entrer quelque chose")
```

mytools.files

```
import os
import json
import csv

def read_file_txt(path):
  cwd = os.getcwd()
  file_path = cwd + path
  with open(file_path, 'r', encoding="utf-8") as fichier:
    contenu = fichier.read()
  return contenu

def write_file_txt(path, content):
  cwd = os.getcwd()
  file_path = cwd + path
  with open(file_path, 'w', encoding="utf-8") as fichier:
    fichier.write(content)

def read_file_json(path):
  cwd = os.getcwd()
  file_path = cwd + path
  with open(file_path, 'r', encoding="utf-8") as fichier:
    contenu = json.load(fichier)
  return contenu

def write_file_json(path, content):
  cwd = os.getcwd()
  file_path = cwd + path
  if not os.path.exists(os.path.dirname(file_path)):
    os.makedirs(os.path.dirname(file_path))
  with open(file_path, 'w', encoding="utf-8") as fichier:
```

```
    json.dump(content, fichier, indent=2, ensure_ascii=False)

def read_file_csv(path):
  cwd = os.getcwd()
  file_path = cwd + path
  content = []
  with open(file_path, 'r', encoding="utf-8") as fichier:
    reader = csv.reader(fichier)
    for row in reader:
      if row:
        content.append(row)
  return content

def write_file_csv(path, content):
  cwd = os.getcwd()
  file_path = cwd + path
  if not os.path.exists(os.path.dirname(file_path)):
    os.makedirs(os.path.dirname(file_path))
  with open(file_path, 'w', encoding="utf-8") as fichier:
    writer = csv.writer(fichier)
    writer.writerows(content)

if __name__ == "__main__":
  # print(read_file_txt("/26_files_txt/myfile.txt"))
  # write_file_txt("/26_files_txt/myfile2.txt","Ceci est un test de la méthode␣
 ↪write_file_txt")
  # print(read_file_json("/27_files_json/sortie.json"))
  # write_file_json("/27_files_json/sortie2.json",[ {'test' : "truc", "test2" :␣
 ↪"truc2"}, {'test' : 'chose', 'test2' : 'chose2'}])
  # print(read_file_csv("/29_files_csv/mon_fichier.csv"))
  # write_file_csv("/28_files_csv/mon_fichier.csv",[ ['test', "truc", "test2" ,␣
 ↪"truc2"], ['test' , 'chose', 'test2' , 'chose2' ]])

  write_file_csv("/log_mars.csv",[['20220225','1','Défaut com'],␣
 ↪['20220305','2','Choc robot'], ['20220325','6','Défaut variateur']])
```

```
import os
import json
import csv

def get_path_from_string(path_string:str):
  cwd = os.getcwd()
  locations = [s for s in path_string.split("/") if s]
  return os.path.join(cwd, *locations)

def __read_file(path_string,read):
  path = get_path_from_string(path_string)
```

```
  with open(path, 'r', encoding="utf-8") as fichier:
    contenu = read(fichier)
  return contenu

def __write_file(path_string, content, write):
  path = get_path_from_string(path_string)
  if not os.path.exists(os.path.dirname(path)):
    os.makedirs(os.path.dirname(path))
  with open(path, 'w', encoding="utf-8", newline="") as fichier:
    write(fichier,content)

def read_file_txt(path_string):
  def read(fichier):
    return fichier.read()
  return __read_file(path_string,read)

def write_file_txt(path_string,content):
  def write(fichier,content):
    fichier.write(content)
  __write_file(path_string,content,write)

def read_file_json(path_string):
  def read(fichier):
    return json.load(fichier)
  return __read_file(path_string,read)

def write_file_json(path_string, content):
  def write(fichier,content):
    json.dump(content,fichier,indent=2,ensure_ascii=False)
  __write_file(path_string,content,write)

def read_file_csv(path_string, delimiter = ","):
  def read(fichier):
    contenu = []
    reader = csv.reader(fichier, delimiter=delimiter)
    for row in reader:
      if row:
        contenu.append(row)
    return contenu
  return __read_file(path_string,read)

def write_file_csv(path_string, content, delimiter = ","):
  def write(fichier,content):
    writer = csv.writer(fichier,delimiter=delimiter)
    writer.writerows(content)
  __write_file(path_string,content,write)
```

```
if __name__ == "__main__":
  print(read_file_txt("/04_files_txt/myfile.txt/"))
  write_file_txt("04_files_txt/myfile2factored.txt","Ceci est un test de la␣
 ↪méthode write_file_txt")
  write_file_txt("04_files_txt/newfolder/otherfolder/myfile3factored.txt","Ceci␣
 ↪est un test de la méthode write_file_txt")
  print(read_file_json("/05_files_json/sortie.json"))
  donnees = [
    {
      'nom' : "Doe",
      "prenom" : "John"
    },
    {
      'nom' : 'Clapton',
      'prenom' : 'Eric'
    }
  ]
  write_file_json("05_files_json/myfilefactored.json",donnees)
  write_file_json("05_files_json/newfolder/otherfolder/myfile2factored.
 ↪json",donnees)
  print(read_file_csv("/06_files_csv/myfile.csv"))
  donnees = [
    [ "nom","prenom" ],
    [ "Doe", "John" ],
    [ 'Clapton', 'Eric' ]
  ]
  write_file_csv("06_files_csv/myfile3factored.csv",donnees)
  write_file_csv("06_files_csv/newfolder/otherfolder/myfile4factored.
 ↪csv",donnees)
```

repository.productDao

```
# Créer un module repository, avec un fichier
# productDao.py
# Coder une classe ProductDao qui contiendra des
# méthodes
# addProduct(self, p)
# getProductById(self, id)
# getAllProducts(self)
# updateProduct(self, p)
# deleteProductById(self, id)
# qui implémentent les fonctionnalités associées pour
# une table produit et qui a comme propriétés le nom de
# la base, le nom de la table.
# Enfin, elle aura une méthode close() pour fermer la
# connexion et une méthode init_table() pour créer la
# table et y mettre des données initiales.
```

```python
# Les opérations / erreurs seront loggées dans un
# fichier app.log

import sqlite3
import logging

if __name__ == "__main__":
  import sys
  import os
  SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
  sys.path.append(os.path.dirname(SCRIPT_DIR))

from myclasses.product import Product


logging.basicConfig(
    level=logging.INFO,
    filename="app.log",
    filemode='a',
    format="%(asctime)s - [%(levelname)s]: %(filename)s: %(message)s",
    datefmt="%d/%m/%Y %H:%M:%S",
    encoding="utf-8"
)

class ProductDao:
  DB_NAME = "repository/db.sqlite3"
  TABLE_NAME = "product"
  def __init__(self):
    self.__cnx = None
    try:
      self.__cnx = sqlite3.connect(ProductDao.DB_NAME)
    except sqlite3.DatabaseError as e:
      logging.error(e)

  def init_table(self):
    cursor = self.__cnx.cursor()
    logging.info(f"Création de la table {ProductDao.TABLE_NAME} (si elle␣
 ↪n'existe pas)")
    try:
      sql = "CREATE TABLE IF NOT EXISTS " + ProductDao.TABLE_NAME
      sql += """ (
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      description varchar(50) NOT NULL,
      prix real NOT NULL
      )"""
      cursor.execute(sql)
      # equivalent de truncate (remet le compteur d'id à 0)
```

```
      # https://sqlite.org/autoinc.html
      sql = "DELETE FROM " + ProductDao.TABLE_NAME
      cursor.execute(sql)
      sql = "delete from sqlite_sequence where name='" + ProductDao.TABLE_NAME␣
↪+ "'"
      cursor.execute(sql)
      sql = "INSERT INTO " + ProductDao.TABLE_NAME + " (id, description, prix)␣
↪VALUES (NULL, :description, :prix)"
      p1 = Product(1,"Table",50)
      p2 = Product(2, "Crayon",1.5)
      p3 = Product(3,"PC",2500)
      # p.__dict__ : dictionnaire avec tous les
      # attributs de l'objet p
      # ne fonctionne pas pour les propriétés
      # cursor.execute(sql, p1.__dict__)
      cursor.execute(sql, {"id" : p1.id, "description" : p1.description, "prix":
↪ p1.prix})
      cursor.execute(sql, {"id" : p2.id, "description" : p2.description, "prix":
↪ p2.prix})
      cursor.execute(sql, {"id" : p3.id, "description" : p3.description, "prix":
↪ p3.prix})
      self.__cnx.commit()
    except sqlite3.Error as e:
      logging.error(e)
    else:
      logging.info(f"Succès de la création de la table {ProductDao.TABLE_NAME}␣
↪(si elle n'existe pas)")
    finally:
      cursor.close()

  def close(self):
    self.__cnx.close()

  # méthode utilisée lorsque la variable est
  # supprimée par le ramasse miettes
  def __del__(self):
    self.close()

  def addProduct(self, p: Product):
    logging.info(f"Ajout du produit {p}")
    cursor = self.__cnx.cursor()
    sql = "INSERT INTO " + ProductDao.TABLE_NAME + " (id, description, prix)␣
↪VALUES (null, :description, :prix)"
    try:
      cursor.execute(sql, {"id" : p.id, "description" : p.description, "prix":␣
↪p.prix})
```

```
      self.__cnx.commit()
    except sqlite3.Error as e:
      logging.error(e)
    else:
      logging.info(f"Succès de l'ajout du produit {p}")
    finally:
      cursor.close()

  def getProductById(self, id: int) -> Product:
    logging.info(f"Récupération du produit avec l'id {id}")
    sql = f"SELECT * FROM {ProductDao.TABLE_NAME} WHERE id = ?"
    product = None
    try:
      product = self.__cnx.execute(sql,(id,)).fetchall()
    except sqlite3.Error as e:
      logging.error(e)
    else:
      if product:
        # https://sametmarx.com/operateur-splat-ou-etoile-en-python/
        # https://treyhunner.com/2018/10/
↪asterisks-in-python-what-they-are-and-how-to-use-them/
        product = Product(*product[0])
      logging.info(f"Le produit trouvé : {product}")
    return product

  def getAllProducts(self):
    logging.info(f"Récupération de tous les produits")
    sql = f"SELECT * FROM {ProductDao.TABLE_NAME}"
    try:
      result = self.__cnx.execute(sql).fetchall()
    except sqlite3.Error as e:
      logging.error(e)
    else:
      logging.info(f"Succès de récupération de tous les produits")
      return result


  def updateProduct(self, p: Product):
    logging.info(f"Update du produit avec l'id {p.id}")
    sql = "UPDATE " + ProductDao.TABLE_NAME + " SET description=:description,␣
↪prix=:prix WHERE id=:id"
    cursor = self.__cnx.cursor()
    try:
      cursor.execute(sql, {"id" : p.id, "description" : p.description, "prix":␣
↪p.prix})
      self.__cnx.commit()
    except sqlite3.Error as e:
```

```
      logging.error(e)
    else:
      pass
    finally:
      cursor.close()

  def deleteProductById(self, id: int):
    logging.info(f"Suppression du produit avec l'id {id}")
    sql = f"DELETE FROM {ProductDao.TABLE_NAME} WHERE id = ?"
    product = None
    cursor = self.__cnx.cursor()
    try:
      cursor.execute(sql,(id,))
      self.__cnx.commit()
    except sqlite3.Error as e:
      logging.error(e)
    else:
      logging.info(f"Succès de la suppression du produit avec l'id {id}")
    finally:
      cursor.close()
    return product

if __name__ == "__main__":
  dao = ProductDao()
  print("____init_table____")
  dao.init_table()

  print("____getAllProducts____")
  print(dao.getAllProducts())

  print("____getProductById(2)____")
  print(dao.getProductById(2))

  print("____addProduct(Product(19,'Test', 1999))____")
  product = Product(19,"Test", 1999)
  dao.addProduct(product)
  print(dao.getAllProducts())

  print("____deleteProductById(4)____")
  dao.deleteProductById(4)
  print(dao.getAllProducts())

  print("____updateProduct(Product(2,'Edited', 9999))____")
  product = Product(2,"Edited", 9999)
  dao.updateProduct(product)
  print(dao.getAllProducts())
```

```
dao.close()
```

Sommaire