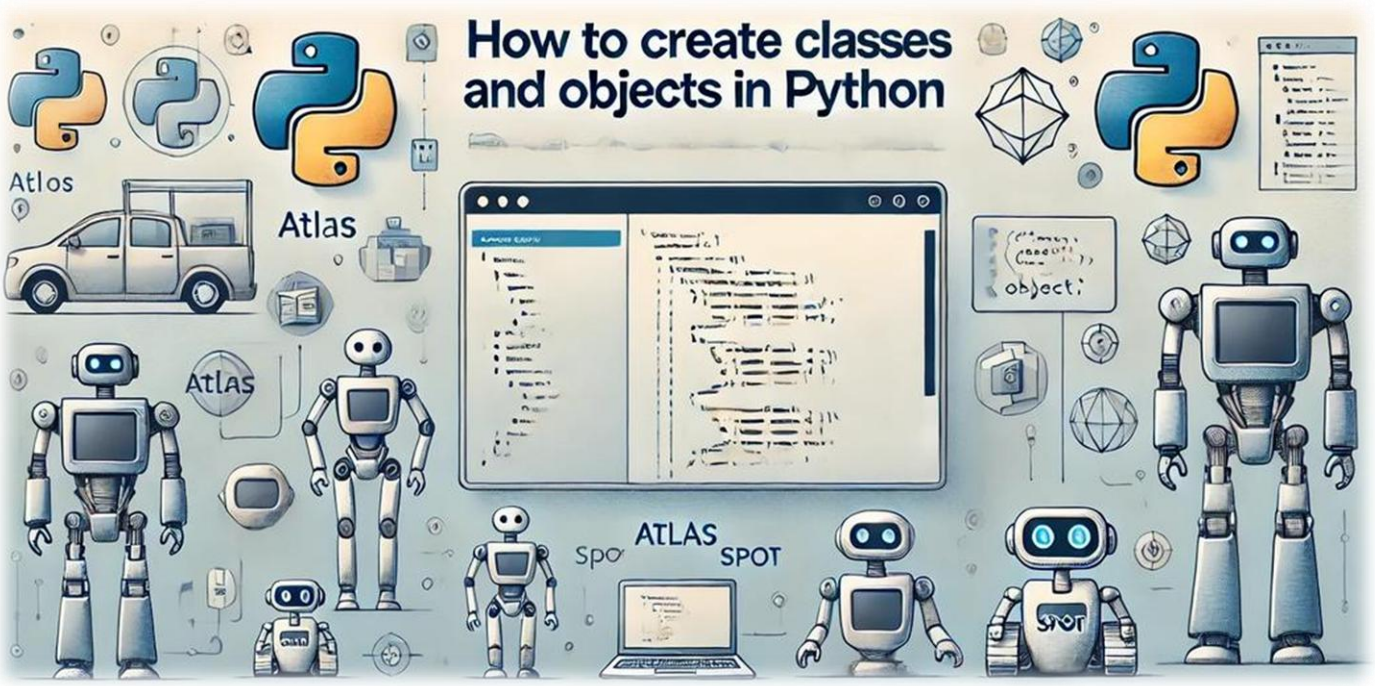


Mon livret d'entrainement à la POO

Exercice 1

Dans le tableau ci-dessous, on donne les caractéristiques nutritionnelles, pour une quantité de 100 grammes, de quelques aliments.

Aliment	Énergie (kcal)	Protéines (g)	Glucides (g)	Lipides (g)
Lait entier UHT	65.1	3.32	4.85	3.63
Farine de blé	343	11.7	69.3	0.8
Huile de tournesol	900	0	0	100

Pour chaque aliment, on souhaite stocker les informations dans un objet de la classe `Aliment`, définie comme suit :

```
class Aliment:
    def __init__(self, e, p, g, l):
        self.energie = e    # énergie en kcal
        self.protéines = p  # protéines en grammes
        self.glucides = g   # glucides en grammes
        self.lipides = l    # lipides en grammes
```

1. Instanciation et manipulation d'objets

a. Écrire, à l'aide du tableau des caractéristiques nutritionnelles, l'instruction en langage Python pour instancier l'objet `lait`.

b. Donner l'instruction qui permet d'obtenir la valeur **65.1** de l'objet `lait` instancié dans la question précédente.

c. Une erreur s'est introduite dans le tableau : la masse de protéines dans le lait est **3.4** au lieu de **3.32**. Donner l'instruction qui modifie la masse de protéines de l'objet `lait`.

2. Ajout d'une méthode à la classe Aliment

On souhaite ajouter une méthode `energie_reelle` à la classe `Aliment`, qui calcule l'énergie en kcal d'un aliment en fonction d'une masse donnée.

Par exemple, pour **245 grammes de lait**, l'énergie réelle sera :

$$245 \times 65.1 \div 100 = 159.495\ kcal$$

L'instruction `lait.energie_reelle(245)` doit renvoyer **`159.495`**.

Recopier et compléter les lignes **n°1** et **n°2** dans la méthode ci-dessous :

```
def energie_reelle(self, .....):
    return .....
```

3. Utilisation d'un dictionnaire

On regroupe les caractéristiques nutritionnelles du tableau dans le dictionnaire suivant, où les clés sont des chaînes de caractères représentant le nom de l'aliment et les valeurs associées sont des objets de la classe `Aliment` :

```
nutrition = {
    'lait': Aliment(65.1, 3.4, 4.85, 3.63),
    'farine': Aliment(343, 11.7, 69.3, 0.8),
    'huile': Aliment(900, 0, 0, 100)
}
```

a. Donner l'instruction qui permet d'obtenir la valeur énergétique en kcal du lait à partir des données de ce dictionnaire.
b. Donner l'instruction qui permet d'obtenir la valeur énergétique réelle de **220 grammes de lait** à partir des données de ce dictionnaire.

4. Calcul de l'énergie réelle totale d'un gâteau

Une recette de gâteau (sans œuf) utilise les ingrédients suivants :

- **230 g de farine**,
- **220 g de lait**,
- **100 g d'huile**.

Les quantités d'ingrédients, en grammes, sont regroupées dans le dictionnaire suivant :

```
recette_gateau = {'lait': 220, 'farine': 230, 'huile': 100}
```

Écrire, en utilisant la classe `Aliment` et la méthode `energie_reelle`, les instructions nécessaires pour calculer l'énergie réelle totale du gâteau.

Exercice 2

Un fabricant de brioches décide d'informatiser sa gestion des stocks. Il écrit pour cela un programme en langage Python. Une partie de son travail consiste à développer une classe `Stock` dont la première version est la suivante :

```
class Stock:
    def __init__(self):
        self.qt_farine = 0  # quantité de farine initialisée à 0 g
        self.nb_oeufs = 0   # nombre d'œufs (0 à l'initialisation)
        self.qt_beurre = 0  # quantité de beurre initialisée à 0 g
```

1. Ajout de beurre au stock

Écrire une méthode `ajouter_beurre(self, qt)` qui ajoute la quantité `qt` de beurre à un objet de la classe `Stock`.

(On admet que l'on a écrit deux autres méthodes `ajouter_farine` et `ajouter_oeufs` qui ont des fonctionnements analogues.)

2. Affichage du stock

Écrire une méthode `afficher(self)` qui affiche la quantité de farine, d'œufs et de beurre d'un objet de type `Stock`. L'exemple ci-dessous illustre l'exécution de cette méthode dans la console :

```
>>> mon_stock = Stock()
>>> mon_stock.afficher()
farine: 0
oeuf: 0
beurre: 0
>>> mon_stock.ajouter_beurre(560)
>>> mon_stock.afficher()
farine: 0
oeuf: 0
beurre: 560
```

3. Vérification du stock pour une brioche

Pour faire une brioche, il faut 350 g de farine, 175 g de beurre et 4 œufs.
Écrire une méthode `stock_suffisant_brioche(self)` qui renvoie un booléen :

- `True` s'il y a assez d'ingrédients dans le stock pour faire une brioche.
- `False` sinon.

4. Production de brioches

On considère la méthode supplémentaire `produire(self)` de la classe `Stock` donnée par le code suivant :

```
def produire(self):
    res = 0
    while self.stock_suffisant_brioche():
        self.qt_beurre = self.qt_beurre - 175
        self.qt_farine = self.qt_farine - 350
        self.nb_oeufs = self.nb_oeufs - 4
        res = res + 1
    return res
```

On considère un stock défini par les instructions suivantes :

```
>>> mon_stock = Stock()
>>> mon_stock.ajouter_beurre(1000)
>>> mon_stock.ajouter_farine(1000)
>>> mon_stock.ajouter_oeufs(10)
```

a. On exécute ensuite l'instruction :

```
>>> mon_stock.produire()
```

Quelle valeur s'affiche dans la console ? Que représente cette valeur ?

b. On exécute ensuite l'instruction :

```
>>> mon_stock.afficher()
```

Que s'affiche-t-il dans la console ?

5. Gestion de plusieurs stocks

L'industriel possède `n` lieux de production distincts et donc `n` stocks distincts. On suppose que ces stocks sont dans une liste dont chaque élément est un objet de type `Stock`.

Écrire une fonction Python `nb_brioches(liste_stocks)` possédant pour unique paramètre la liste des stocks et qui renvoie le nombre total de brioches produites.

Exercice 3

Les participants à un jeu de LaserGame sont répartis en équipes et s'affrontent dans ce jeu de tir, revêtus d'une veste à capteurs et munis d'une arme factice émettant des infrarouges.

Les ordinateurs embarqués dans ces vestes utilisent la programmation orientée objet pour modéliser les joueurs. La classe **`Joueur`** est définie comme suit :

```
class Joueur:
    def __init__(self, pseudo, identifiant, equipe):
        '''Constructeur'''
        self.pseudo = pseudo
        self.equipe = equipe
        self.id = identifiant
        self.nb_de_tirs_emis = 0
        self.liste_id_tirs_recus = []
        self.est_actif = True


    def tire(self):
        '''Méthode déclenchée par l'appui sur la gâchette'''
        if self.est_actif == True:
            self.nb_de_tirs_emis = self.nb_de_tirs_emis + 1

    def est_determine(self):
        '''Méthode qui renvoie True si le joueur réalise un grand nombre
        de tirs
        '''
        return self.nb_de_tirs_emis > 500

    def subit_un_tir(self, id_recu):
        '''Méthode déclenchée par les capteurs de la veste'''
        if self.est_actif == True:
            self.est_actif = False
            self.liste_id_tirs_recus.append(id_recu)
```

1. Instanciation d'un objet Joueur

Parmi les instructions suivantes, recopier celle qui permet de déclarer un objet **`joueur1`**, instance de la classe **`Joueur`**, correspondant à un joueur dont le pseudo est **`"Sniper"`**, dont l'identifiant est **`319`** et qui est intégré à l'équipe **`"A"`** :

- Instruction 1 : `joueur1 = ["Sniper", 319, "A"]`
- Instruction 2 : `joueur1 = new Joueur["Sniper", 319, "A"]`
- Instruction 3 : `joueur1 = Joueur("Sniper", 319, "A")`
- Instruction 4 : `joueur1 = Joueur["pseudo":"Sniper", "id":319, "equipe":"A"]`

2. Méthodes supplémentaires pour la classe Joueur

La méthode **`subit_un_tir`** réalise les actions suivantes : Lorsqu'un joueur actif subit un tir capté par sa veste, l'identifiant du tireur est ajouté à l'attribut **`liste_id_tirs_recus`** et l'attribut **`est_actif`** prend la valeur **`False`** (le joueur est désactivé). Il doit alors revenir à son camp de base pour être de nouveau actif.

a. Écrire la méthode **`redevenir_actif`** qui rend à nouveau le joueur actif **uniquement s'il était précédemment désactivé**.
b. Écrire la méthode **`nb_de_tirs_recus`** qui renvoie le nombre de tirs reçus par un joueur en utilisant son attribut **`liste_id_tirs_recus`**.

3. Classe `Base`

Lorsque la partie est terminée, les participants rejoignent leur camp de base respectif où un ordinateur, qui utilise la classe **`Base`**, récupère les données.

La classe **`Base`** est définie par :

- **Ses attributs :**
  - **`equipe`** : nom de l'équipe (`str`). Par exemple, `"A"`.
  - **`liste_des_id_de_1_equipe`** : liste (`list`) des identifiants connus des joueurs de l'équipe.
  - **`score`** : score (`int`) de l'équipe, dont la valeur initiale est 1000.

- **Ses méthodes :**
  - **`est_un_id_allie`** : renvoie **`True`** si l'identifiant passé en paramètre est un identifiant d'un joueur de l'équipe, **`False`** sinon.
  - **`incremente_score`** : fait varier l'attribut **`score`** du nombre passé en paramètre.
  - **`collecte_information`** : récupère les statistiques d'un participant passé en paramètre (instance de la classe **`Joueur`**) pour calculer le score de l'équipe.

```
def collecte_information(self, participant):
    if participant.equipe == self.equipe:  # test 1
        for id in participant.liste_id_tirs_recus:
            if self.est_un_id_allie(id):  # test 2
                self.incremente_score(-20)
            else:
                self.incremente_score(-10)
```

a. Indiquer le numéro du test (**test 1** ou **test 2**) qui permet de vérifier qu'en fin de partie un participant égaré n'a pas rejoint par erreur la base adverse.
b. Décrire comment varie quantitativement le score de la base lorsqu'un joueur de cette équipe a été touché par le tir d'un coéquipier.

4. Bonus pour les joueurs déterminés

On souhaite accorder à la base un bonus de **40 points** pour chaque joueur particulièrement déterminé (qui réalise un grand nombre de tirs).

Recopier et compléter, en utilisant les méthodes des classes **`Joueur`** et **`Base`**, les 2 lignes de code suivantes qu'il faut ajouter à la fin de la méthode **`collecte_information`** :

```
......  # si le participant réalise un grand nombre de tirs
    ......  # le score de la Base augmente de 40
```

Exercice 4

Dans un jeu de plateforme, des bulles de couleurs et de diamètres différents se déplacent de manière aléatoire. Chaque fois qu'une bulle touche une bulle plus grande, la petite bulle cède son contenu à la plus grande, et celle-ci augmente de surface. Par exemple, si une bulle de 1 cm^2 rencontre une bulle de 4 cm^2, la petite bulle disparaît et la plus grande a désormais une surface de 5 cm^2. À chaque collision, la vitesse de la grande bulle est réduite de moitié.

Le développeur a choisi de coder en Python. Chaque bulle est un objet disposant des attributs suivants :

- **`xc, yc`** : deux entiers, les coordonnées du pixel placé au centre de la bulle,
- **`rayon`** : un entier, le rayon de la bulle en pixels,
- **`couleur`** : un entier, la couleur de la bulle,
- **`dirx, diry`** : deux décimaux (`float`) qui déterminent les déplacements à l'horizontale et à la verticale à chaque fois que la bulle se déplace. Ces deux valeurs déterminent donc la direction et la vitesse de la bulle.

On suppose que toutes les fonctions de la bibliothèque **`math`** ont déjà été importées par l'instruction `from math import *`.

1. Création et gestion des bulles

Pour simplifier, on se limitera à un jeu de six bulles. Au départ, on crée une liste appelée **`Mousse`** de longueur six contenant six emplacements vides :

```
Mousse = [None, None, None, None, None, None]
```

Le code ci-dessous montre le début du programme et notamment la structure de la classe nommée **`Cbulle`** ainsi que le code permettant le déplacement d'une bulle.

```
from random import randint
from math import *

class Cbulle:
    def __init__(self):
        self.xc = randint(0, 100)
        self.yc = randint(0, 100)
        self.rayon = randint(0, 10)
        self.dirx = float(randint(-1, 1))   # dirx et diry valent -1.0, 0.0 ou 1.0
        self.diry = float(randint(-1, 1))
        self.couleur = randint(1, 65535)

    def bouge(self):
        # déplace la bulle
        self.xc = self.xc + self.dirx
        self.yc = self.yc + self.diry
```

On crée les six bulles une à une et ces objets sont stockés dans les emplacements vides de la liste **`Mousse`** :

```
Mousse = [bulle1, bulle2, bulle3, bulle4, bulle5, bulle6]
```

Lors d'une collision, la bulle la plus petite disparaît et est remplacée dans la liste par la valeur **`None`**, tandis que la plus grosse voit sa surface augmenter. Au cours d'une partie, si une ou plusieurs bulles ont disparu, le programme peut en introduire de nouvelles dans le jeu : dans ce cas, lorsqu'une nouvelle bulle apparaît, elle remplace le premier **`None`** de la liste **`Mousse`**.

a. Recopier les quatre dernières lignes et compléter les _____ du code Python ci-dessous :

```
def donnePremierIndiceLibre(Mousse):
    """
    Mousse est une liste.
    La fonction doit renvoyer l'indice du premier emplacement libre
    contenant None) dans la liste Mousse ou renvoyer 6 en l'absence
    d'un emplacement libre dans Mousse.
    """
    i = 0
    while ______ and Mousse[i] != None:
          _____
```

b. Lorsque le jeu crée une bulle (instance de la classe **`Cbulle`**), il doit ensuite la placer dans la liste **`Mousse`** à la place d'un **`None`**. Écrire la fonction **`placeBulle(B)`** qui reçoit en paramètre un objet de type **`Cbulle`** et qui place cet objet dans la liste **`Mousse`**. Cette fonction ne renvoie rien, mais la liste **`Mousse`** est modifiée. Si aucun emplacement n'est disponible, la fonction ne modifie rien.

2. Détection des collisions entre bulles

Pour le bon déroulement du jeu, on a besoin d'une fonction **`bullesEnContact(B1, B2)`** qui renvoie **`True`** si la bulle **`B2`** touche la bulle **`B1`**, et **`False`** dans le cas contraire.

On peut remarquer que deux bulles sont en contact si la distance qui sépare leur centre est inférieure ou égale à la somme de leurs rayons.

On dispose de la fonction **`distanceEntreBulles(B1, B2)`** qui calcule et renvoie la distance entre les centres des bulles **`B1`** et **`B2`**.

Écrire la fonction **`bullesEnContact(B1, B2)`**.

3. Gestion des collisions

Quand une petite bulle touche une plus grosse bulle, on appelle la fonction **`collision`**, ci-dessous, où **`indPetite`** est l'indice de la petite bulle et **`indGrosse`** l'indice de la grosse bulle dans **`Mousse`**.

Recopier et compléter les _____ de la fonction **`collision`** :

```
def collision(indPetite, indGrosse, mousse):
    """
    Absorption de la plus petite bulle d'indice indPetite
    par la plus grosse bulle d'indice indGrosse. Aucun test
    n'est réalisé sur les positions.
    """
    # calcul du nouveau rayon de la grosse bulle
    surfPetite = pi * Mousse[indPetite].rayon ** 2
    surfGrosse = pi * Mousse[indGrosse].rayon ** 2
    surfGrosseApresCollision = _______
    rayonGrosseApresCollision = sqrt(surfGrosseApresCollision / pi)
    # réduction de 50 % de la vitesse de la grosse bulle
    Mousse[indGrosse].dirx = _______
    Mousse[indGrosse].diry = _______
    # suppression de la petite bulle dans Mousse
    _______
```

Exercice 5

Simon souhaite créer en Python le jeu de cartes **« La Bataille »** pour deux joueurs. Les questions suivantes demandent de reprogrammer quelques fonctions du jeu.

Règles du jeu de la Bataille :

- **Préparation :**
  - Distribuer toutes les cartes aux deux joueurs.
  - Les joueurs ne prennent pas connaissance de leurs cartes et les laissent en tas, face cachée devant eux.
- **Déroulement :**
  - À chaque tour, chaque joueur dévoile la carte du haut de son tas.
  - Le joueur qui présente la carte ayant la plus haute valeur emporte les deux cartes qu'il place sous son tas.
  - Les valeurs des cartes sont, dans l'ordre de la plus forte à la plus faible : **As, Roi, Dame, Valet, 10, 9, 8, 7, 6, 5, 4, 3, 2** (la plus faible).
  - Si deux cartes sont de même valeur, il y a **« bataille »** :
    - Chaque joueur pose alors une carte face cachée, suivie d'une carte face visible sur la carte dévoilée précédemment.
    - On recommence l'opération s'il y a de nouveau une bataille. Sinon, le joueur ayant la valeur la plus forte emporte tout le tas.
  - Lorsque l'un des joueurs possède toutes les cartes du jeu, la partie s'arrête et ce dernier gagne.

Simon crée une classe Python **Carte**. Chaque instance de la classe a deux attributs : un pour sa **valeur** et un pour sa **couleur**. Il donne :

- Au **valet** la valeur **11**,
- À la **dame** la valeur **12**,
- Au **roi** la valeur **13**,
- À l'**as** la valeur **14**.

La **couleur** est une chaîne de caractères : **« trèfle »**, **« carreau »**, **« cœur »** ou **« pique »**.

1. Création de la classe Carte

Simon a écrit la classe Python **`Carte`** suivante, ayant deux attributs **`valeur`** et **`couleur`**, et dont le constructeur prend deux arguments : **`val`** et **`coul`**.

a. Recopier et compléter les _____ des lignes 3 et 4 ci-dessous :

```
class Carte:
    def __init__(self, val, coul):
        self.valeur = ......  # Ligne 3
        self.couleur = ......  # Ligne 4
```

b. Parmi les propositions ci-dessous, quelle instruction permet de créer l'objet **« 7 de cœur »** sous le nom **`c7`** ?

- `c7.__init__(self, 7, "coeur")`
- `c7 = Carte(self, 7, "coeur")`
- `c7 = Carte(7, "coeur")`
- `from Carte import 7, "coeur"`

2. Initialisation du jeu de cartes

On souhaite créer le jeu de cartes. Pour cela, on écrit une fonction **`initialiser()`** :

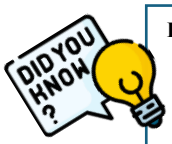
- Sans paramètre,
- Qui renvoie une liste de **52 objets** de la classe **`Carte`** représentant les **52 cartes du jeu**.

Voici une proposition de code. Recopier et compléter les lignes suivantes pour que la fonction réponde à la demande :

```
def initialiser():
    jeu = []
    for c in ["coeur", "carreau", "trefle", "pique"]:  # Couleur carte
        for v in range(......):  # Valeur carte
            carte_cree = ......
            jeu.append(carte_cree)
    return jeu
```

3. Modélisation du tas de cartes

Parmi les structures linéaires de données suivantes : **Tableau, File, Pile**, quelle est celle qui modélise le mieux un **tas de cartes** dans ce jeu de la bataille ? Justifier votre choix.



Pile (Stack) :

- **Principe :** *« Dernier arrivé, premier sorti »* (LIFO).
- **Opérations :**
  - **Empiler** (ajouter un élément au sommet).
  - **Dépiler** (retirer l'élément du sommet).

File (Queue) :

- **Principe :** *« Premier arrivé, premier sorti »* (FIFO).
- **Opérations :**
  - **Enfiler** (ajouter un élément à la fin).
  - **Défiler** (retirer l'élément du début).

4. Comparaison des cartes

Écrire une fonction **`comparer(carte1, carte2)`** qui prend en paramètres deux objets de la classe **`Carte`**. Cette fonction renvoie :

- **`0`** si la force des deux cartes est identique,
- **`1`** si la carte **`carte1`** est strictement plus forte que **`carte2`**,
- **`-1`** si la carte **`carte2`** est strictement plus forte que **`carte1`**.