TD1 : Programmation Orientée Objet – POO

Exercice 1 : Classes fondamentales, méthodes et attributs

1- Définissez une classe **`Voiture()`** qui permette **d'instancier** des objets reproduisant le comportement des voitures automobiles. Le constructeur de cette classe initialisera les **attributs d'instance** avec les valeurs par défaut indiquées : **`marque = 'Ford'`**, **`couleur = 'rouge'`**. Lorsque l'on instanciera un nouvel objet ***`Voiture()`***, on pourra choisir sa **marque** et sa **couleur**, mais pas sa **vitesse**, ni le nom de son **pilote**.

2- Définir la méthode d'instance ***`choix_onducteur(nom)`*** qui permettra de désigner (ou changer) le nom du **pilote**,

3- Définir la méthode d'objet ***`accelerer(taux, duree)`*** qui permettra de faire varier la vitesse de la voiture. La variation de vitesse obtenue sera égale au produit : ***Δv = taux x duree***. Par exemple, si la voiture accélère au taux de 1,3 m/s^2 pendant 20 secondes, son gain de vitesse doit être égal à 26 m/s. Des taux négatifs seront acceptés (ce qui permettra de décélérer). La variation de vitesse ne sera pas autorisée si le **pilote** est **`'personne'`**.

4- Définir la méthode ***`affiche_tout()`*** qui permettra de faire apparaître les propriétés présentes de la voiture, c'est-à-dire sa **marque**, sa **couleur**, le nom de son **pilote** et sa **vitesse**.

Exemples d'utilisation de cette classe :

```
>>> v1 = Voiture('Peugeot', 'bleue')
>>> v2, v3 = Voiture(couleur = 'verte'), Voiture('Mercedes')
>>> v1.choix_conducteur('Ahmed')
>>> v2.choix_conducteur('Ali')
>>> v2.accelerer(1.8, 12)
>>> v3.accelerer(1.9, 11)
Cette voiture n'a pas de conducteur !
>>> v2.affiche_tout()
Ford verte pilotée par Ali, vitesse = 21.6 m/s.
>>> v3.affiche_tout()
Mercedes rouge pilotée par personne, vitesse = 0 m/s.
```

Exercice 2 : Simulation d'un système physique simple, Encapsulation

1- Définissez une classe **`Satellite()`** qui permette d'instancier des objets simulant des satellites artificiels lancés dans l'espace, autour de la terre. Le constructeur de cette classe initialisera les attributs d'instance suivants, avec les **valeurs par défaut** indiquées : masse = 100, vitesse = 0. Lorsque l'on instanciera un nouvel objet **`Satellite()`**, on pourra choisir son **nom**, sa **masse** et sa **vitesse**.

2- l'attribut « masse » doit être protégé contre l'accès direct externe. Donner une solution d'encapsulation de cet attribut avec les deux méthodes d'interface accesseur **`getMasse()`** et mutateur **`setMasse()`**.

3- **`impulsion(force, duree)`** : permettra de faire varier la vitesse du satellite. Pour savoir comment, rappelez-vous votre cours de physique : la variation de vitesse « ***Δv*** » subie par un objet de masse « m » soumis à l'action d'une force « F » pendant un temps « t » vaut ***Δv = (F x Δt)/m***. Par exemple : un satellite de 300 kg qui subit une force de 600 Newtons pendant 10 secondes voit sa vitesse augmenter (ou diminuer) de 20 m/s.

4- **`affiche_vitesse()`** : affichera le nom du satellite et sa vitesse courante (réel de 6 chiffres dont 3 après la virgule).

5- **`energie()`** : renvoi au programme appelant la valeur de l'énergie cinétique du satellite. Rappel : l'énergie cinétique « Ec » se calcule à l'aide de la formule ***Ec = 1/2 m v^2***.

Voici un exemple d'utilisation de cette classe :

```
>>> s1 = Satellite('Zoé', masse =250, vitesse =10)
>>> s1.impulsion(500, 15)
>>> s1.affiche_vitesse()
vitesse du satellite Zoé = 40.000 m/s.
>>> print(s1.energie())
200000
>>> s1.masse
méthode accesseur...
250
>>> s1.masse = 100
```

```
11.   méthode mutateur...
```

Exercice 3 : Réalisation d'un jeu de cartes

Définissez une classe **`JeuDeCartes()`** permettant d'instancier des objets dont le comportement soit similaire à celui d'un vrai jeu de cartes. La classe devra comporter au moins les quatre méthodes suivantes :

1- Méthode **`constructeur`**: Création et remplissage d'une liste de 52 éléments, qui sont eux-mêmes des tuples de 2 entiers. Cette liste de tuples contiendra les caractéristiques de chacune des 52 cartes. Pour chacune d'elles, il faut en effet mémoriser séparément un entier indiquant la valeur (2, 3, *4*, *5*, *6*, *7*, *8*, *9*, 10, 1l, 12, 13, 14, les 4 dernières valeurs étant celles des VALET, DAME, ROI et AS), et un autre entier indiquant la couleur de la carte (c'est-à-dire 0, 1, 2, 3 pour Cœur, Carreau, Trèfle et Pique). Dans une telle liste, l'élément (11,2) désigne donc le valet de Trèfle, et la liste complète doit être du type : [(2,0), (3,0), (4,0), ... (12,3), (13,3), (14,3)]. Donner le code de cette méthode.

2- Méthode **`nomCarte()`** : Cette méthode doit renvoyer, sous la forme d'une chaîne, l'identité d'une carte quelconque dont on lui a fourni le tuple descripteur en argument.
Par exemple, l'instruction : **`print(jeu.nomCarte((14,3)))`** doit provoquer l'affichage : **`As de pique`**. Donner le code de cette méthode.

3- Méthode **`battre()`** : Comme chacun sait, battre les cartes consiste à les mélanger. Cette méthode sert donc à mélanger les éléments de la liste contenant les cartes, quel qu'en soit le nombre. Donner le code de cette méthode.

4- Méthode **`tirer()`** : Lorsque cette méthode est invoquée, une carte est retirée du jeu. Le tuple contenant sa valeur et sa couleur est renvoyé au programme appelant. On retire toujours la première carte de la liste. Si cette méthode est invoquée alors qu'il ne reste plus aucune carte dans la liste, il faut alors renvoyer l'objet spécial **`None`** au programme appelant. Donner le code de cette méthode.

Exemple d'utilisation de la classe ***JeuDeCartes()*** :

```
1.  jeu = JeuDeCartes() # instanciation d'un objet
2.  jeu.battre() # mélange des cartes
3.  for n in range(53): # tirage des 52 cartes :
4.     c = jeu.tirer()
5.     if c == None: # il ne reste plus aucune carte
6.        print('Terminé !') # dans la liste
7.     else:
8.        Print(jeu.nomCarte(c)) # valeur et couleur de la carte
```

5- En fin, pour Compléter cet exercice, définir deux joueurs A et B. Instancier deux jeux de cartes (un pour chaque joueur) et les mélanger. Ensuite, à l'aide d'une boucle, tirer 52 fois une carte de chacun des deux jeux et comparer leurs valeurs. Si c'est la première des deux qui a la valeur la plus élevée, on ajoute un point au joueur A. Si la situation contraire se présente, on ajoute un point au joueur B. Si les deux valeurs sont égales, on passe au tirage suivant. Au terme de la boucle, comparer les comptes de A et B pour déterminer le gagnant.

Exercice 4 : Héritage simple et polymorphisme

1. Ecrire une classe **`Cercle`** qui permet de manipuler des objets de type Cercle caractérisé chacun par un rayon « **r** ». La classe contiendra :
   a. Un constructeur,
   b. Une méthode **`surface`** » qui calcule la surface du cercle,
   c. Une méthode **`perimetre`** qui calcule le périmètre du cercle,
2. Redéfinir la méthode spéciale __str__() qui retourne une chaîne de caractères représentant un objet de type **`Cercle`** sous le format suivant : **`Cercle <rayon : périmètre : surface>.`**
   Les champs **`rayon, périmètre et surface`** sont des réels à représenter sur 6 chiffres avec 3 chiffres après la virgule.
3. Définir ensuite une classe **`Cylindre`** dérivée de la classe **`Cercle`** précédente. Le constructeur de cette nouvelle classe comportera les deux paramètres **`rayon`** et **`hauteur`**.
4. Ecrire une méthode **`volume`** qui devra renvoyer le volume du cylindre (Volume d'un cylindre = surface de section x hauteur).

5. Redéfinir la méthode spéciale **`__repr__()`** qui retourne une chaîne de caractères comme représentation canonique d'un objet de type **`Cylindre`** sous le format suivant : **`Cylindre (rayon : Hauteur : Volume).`** Les champs réels sont représentés sur 7 chiffres et avec 2 chiffres après la virgule.
6. Définir une classe **`Cone`**, qui devra dériver cette fois de la classe **`Cylindre`**, et dont le constructeur comportera lui aussi les deux paramètres **`rayon`** et **`hauteur`**. Cette nouvelle classe possédera sa propre méthode **`volume`**, laquelle devra renvoyer le volume du cône (rappel: volume d'un cône = volume du cylindre / 3).
7. Redéfinir la méthode spéciale **`__str__()`** qui retourne une chaîne de caractères représentant un objet de type **`Cone`** sous le format suivant : **`Cone (rayon : Hauteur : Volume).`** Les champs réels sont représentés sur 7 chiffres et avec 2 chiffres après la virgule.

Exercice 5 : Gestion d'un magasin

Dans cet exercice, on s'intéresse à la création de trois classes permettant de modéliser un **magasin** composé par une liste de Stocks de Produits. Un produit admet **une référence qui l'identifie** et un **prix d'achat**. Un produit est vendu avec un taux de bénéfice, tel que le prix de vente peut être modifié lors des périodes de solde. Dans la suite, nous allons écrire d'abord une classe **Produit**, puis une **classe Stock** et enfin une classe **Magasin**.

Partie 1. La classe Produit :

1.1- Cette classe possède un attribut de classe **`POURCENTAGE_GAIN`** égal à 20. Ce pourcentage permettra dans la suite de calculer le prix de vente du produit (la valeur par défaut du gain est 20%). Ensuite**,** définir le constructeur **__init__(self, refProd, prixAchat**) tel que les deux paramètres initialisent les deux attributs suivants :

- **`ref`** : une chaîne, qui représente la **référence (l'identifiant)** du produit,
- **`prixAchat :`** un nombre (entier/réel), qui représente le prix d'achat du produit.

1.2- Définir une méthode **`getPrixVente(self, pourcentage = None)`** qui permet de calculer et retourner le prix de vente d'un Produit. Le paramètre **pourcentage** ayant **par défaut** la valeur **None.** Si le **pourcentage** est égal à **None**, alors il prend la valeur de l'attribut de classe **POURCENTAGE_GAIN**. Le prix de vente est calculé **à partir du prix d'achat** du produit et du **pourcentage** en appliquant la formule suivante : **prix de vente = prix d'achat + prix d'achat × (pourcentage/100)**

1.3- Définir une méthode **`setPrixAchat`** qui permet de modifier le **`prixAchat`** du Produit. Elle prend en paramètre la valeur du prix. Cette méthode doit déclencher une exception intitulée **TypeError** si le type du paramètre n'est pas réel, ou une exception intitulée **ValueError** si la valeur du paramètre n'est pas positive. L'instruction **`raise nom_exception(msg)`** déclenche l'exception **nom_exception** en affichant le message **msg**.

1.4- Définir une méthode **`__eq__(self,other)`** qui permet de tester l'égalité entre deux Produits **`self`** et **`other`** en fonction de leurs identifiants.

1.5- Définir une méthode de la représentation textuelle **__str__** qui permettra d'avoir les différentes informations (les champs réels avec 3 chiffres apprès la virgule) d'un objet de type **`Produit`** sous la forme :"**(refProduit ; prixAchat ; prixVente) "**

Partie 2. La classe Stock :

Définir une classe **Stock** qui comprend **deux attributs** : un attribut **`prod`** instance de la classe **`Produit`** et un attribut **`quantite`** représentant la quantité en stock correspondante.

2.1- Ecrire la méthode constructeur **`__init__(self,refProd,prixAchat,quantité=0)`** qui permet d'initialiser :

- L'attribut **`prod`** est un objet de type **`Produit`** construit à partir des paramètres **`refProd`** et **`prixAchat,`**
- L'attribut **`quantite`** est initialisé à partir d'un paramètre ayant une **valeur par défaut égale à zéro.**

Par Exemple, L'objet « **s** » de type Stock contenant le produit de **référence** "**P1**", ayant un **prix** **100**, avec une quantité **500**, est instancié comme suit : **`s = Stock ("P1", 100, 500)`**

2.2- Définir une méthode **`modifierPrixProd`** qui prend en paramètre un **`prix`** et qui permet de modifier **le prix d'achat** du produit en Stock.
2.3- Définir une méthode **`__eq__(self, other)`** qui permet de tester l'égalité entre deux Stock **self** et **other.**
2.4- Définir une méthode **`estDisponible`** qui prend un paramètre **`qte`**, et qui permet de tester si la quantité en stock est supérieur à **`qte.`**
2.5- Définir une méthode **setQuantite** qui permet, à partir d'un paramètre **`qte`**, de mettre à jour la quantité en stock. Cette méthode déclenche une exception si **`qte`** **n'est pas positive.**
2.6- Une méthode de représentation textuelle **`__str__`** qui permettra de retourner le contenu du stock sous la forme **`"(refProduit ; prixAchat ; prixVente) : quantité en stock".`**

Partie 3. La classe Magasin :

Cette classe **`Magasin`** contient les deux attributs suivants :

- Attribut **`adresse`** de type chaine, représentant l'adresse postale du magasin,
- Attribut **`LS`** de type **`list`** tel que chaque élément de **`LS`** est un objet de type **`Stock`** (Autrement dit, la liste **`LS`** contiendra des **instances** de la classe **`Stock`).**

3.1- Ecrire un Constructeur **`__init__(self, adresse, Dprods)`** paramètré par l'**`adresse`** postale du magasin, et un dictionnaire **`Dprods`** où :

- Les **clés** sont les **références** des **produits** à stocker dans le magasin,
- Les **valeurs** sont des **tuples** de la forme **(prixAchat, quantité)** de ces produits.

Ce constructeur utilise les données du dictionnaire **`Dprods`** pour créer les instances de **`Stock`** à ajouter comme éléments de l'attribut liste **`LS.`**
3.2- Ecrire une méthode **`__contains__`** : qui reçoit en paramètre un objet de type **`Stock`** et vérifie s'il est présent dans le magasin.
3.3- Définir une méthode **`ajouterStock`** : qui ajoute un objet de type **`Stock`** passé en paramètre au magasin (En faisant la gestion des exceptions nécessaires).
3.4- Définir une méthode **`__len__`** : qui retourne le nombre de produits présents dans le magasin.
3.5- Définir une méthode **`__getitem__`** : qui permet, à partir d'une référence de produit **`ref`** passée en paramètre, de rechercher le stock de ce produit dans le magasin. Elle retourne un objet de type **`Stock`** s'il existe et retourne **`None`** sinon.
3.6- Définir une méthode **majQteStock** qui permet, à partir de la référence d'un produit **`ref`** et d'une quantité **`qte`**, de mettre à jour la quantité en stock d'un produit existant. Cette méthode prend de plus un paramètre **type_maj** de type entier, il est égal à « **1** » dans le cas d'un ajout et « **-1** » dans le cas d'une réduction.
3.7- Définir une méthode **`__str__`** qui permettra de **représenter textuellement** les stocks des produits du magasin**.** Le but est d'afficher le nombre de produits en magasin, puis l'affichage sera fait **ligne par ligne** où chaque ligne est de la forme suivante :
`"(refProduit ; prixAchat ; prixVente) : quantité en stock"`
De plus, l'affichage sera **trié selon les quantités en stock** en ordre **croissant**. Utilisez la fonction **sorted** :

`sorted(it, key = fct)` : retourne une liste contenant les éléments de l'itérable **`it`** dans l'ordre croissant où le paramètre **`key`** est une fonction qui définit le critère de comparaison pour effectuer le tri.
Exemple :

```
>>> L = [('ax', 2), ('aa', 8), ('ca', 5), ('da', 0)]
>>> sorted(L,key=lambda e:e[1])
[('da', 0), ('ax', 2), ('ca', 5), ('aa', 8)]
```

3.8- Définir une méthode **`supprimerStock`** qui, à partir du **référence d'un produit**, permet de supprimer le stock correspondant.
3.9- Définir une méthode **`get_stocks_par_prix`** qui permet de retourner la liste des stocks de produits dont le prix de vente est compris entre les valeurs de deux paramètres **`prixInf`** et **`prixSup`** .
3.10- Définir une méthode **`get_produits_solde_qte`** qui permet de retourner une **liste de tuples** où chaque tuple comprend la **`ref`** du produit ainsi que son **prix d'achat**, son **prix de vente** initial et son **prix de vente après remise**. Les produits qui seront soldés sont ceux dont la quantité en stock est **inférieure** à une valeur **`qt_min.`** Le taux de **`remise`** et la valeur **`qt_min`** sont donnés en paramètres.