Cours - Programmation orientée objet - POO -

1 Introduction

1.1 La notion d'objet et de classe

— Jusqu'ici, les programmes ont été réalisés en **programmation procédurales**, c'est à dire que chaque programme a été décomposé en plusieurs fonctions réalisant des tâches simples.
Cependant lorsque plusieurs programmeurs travaillent simultanément sur un projet, il est nécessaire de programmer autrement afin d'éviter les conflits entre les fonctions.

— Un **objet** se caractérise par 3 choses :
  — son nom
  — des attributs
  — des méthodes

Par exemple, pour un arbre binaire :

son nom : ArbreBinaire

ses attributs : racine, fils gauche et fils droit

ses méthodes :

Il y a des méthodes nommées :
- *accesseurs (celles qui ne modifient pas l'objet) :*
- *mutateurs (celles qui modifient l'objet) :*

— En **programmation orientée objet**, on fabrique de nouveau types de données correspondant aux besoin du programme. On réfléchit alors aux caractéristiques (nom, attributs et méthodes) possibles pour cet objet.
Ces caractéristiques et ces actions sont regroupées dans un code spécifique associé au type de données, appelé **classe**.

1.2 Classe : un premier exemple avec le type list

Le type de données *list* est une classe .

```
# Dans la console PYTHON
>>> l=[1,5,2]
>>> type(l)
<class list>
```

Une action possible sur les **objets de type liste** est le tri de celle-ci avec la **méthode** nommée **sort()**.
On parle alors de **méthode** et la syntaxe est :

nom_objet . nom_méthode()

```
# Dans la console PYTHON
>>> l=[1,5,2]
>>> l.sort()
>>>l
[1,2,5]
```

1.3 Classe : vocabulaire

— Le type de données avec ses **caractéristiques** et ses **actions** possibles s'appelle **classe**.
— Les **caractéristiques (ou variables)** de la classe s'appellent les **attributs**.
— Les **actions possibles** à effectuer avec la classe s'appellent les **méthodes**.
— Un objet du type de la classe s'appelle une **instance de la classe**
— Lorsqu'on définit les **attributs** d'un objet de la classe, on parle d'**instanciation**.
— On dit que les attributs et les méthodes sont **encapsulés** dans la classe.
On peut afficher les méthodes associées à un objet avec la fonction **dir(objet)** :

```
# Dans la console PYTHON
>>> L=[1,5,2]
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',
'__doc__','__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__','__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy','count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

On retrouve les méthodes connues concernant les listes : **sort(), count(), append(), pop()** ...
Citer un accesseur et un mutateur.
La **programmation orientée objet**, qui fait ses débuts dans les années 1960 aujourd'hui elle est même incontournable dans des langages récents comme *Java*.

2 Création d'une classe : pas à pas

2.1 Un constructeur

— On va créer une classe simple, la **classe Carte** correspondant à une carte d'un jeu.
Par convention, le nom d'une classe commence toujours par une majuscule.

```
# Dans l'éditeur PYTHON
class Carte:
    "Une carte d'un jeu de 32 ou 52 cartes"
```

— Une **méthode constructeur** qui va définir **les attributs de la cartes** qui seront :
— sa valeur $2, 3 \cdots, 10, 11$ pour Valet, 12 pour Dame, 13 pour Roi et14 pour As ;
— et sa couleur (Carreau, Coeur, Trèfle, Pique).

```
# Dans l'éditeur PYTHON
class Carte:   # Définition de la classe
    "Une carte d'un jeu de 32 ou 52 cartes"
    def __init__(self,valeur,couleur):    # méthode 1 : constructeur
        self.valeur=valeur # 1er attribut  valeur {de 2  à 14 pour as}
        self.couleur=couleur # 2e attribut {'pique','carreau','coeur','trefle'}
```

Avec deux tirets bas ou underscores (AltGr 8) de part et d'autre de init.
— Création d'une **instance** de la **classe Carte** :

```
# Dans la console PYTHON
>>>x=Carte(5,'carreau')
>>> x
<__main__.Carte object at 0x7f7f57d4ae90>
```

Lorsque l'on créé un objet, son constructeur est appelé implicitement et l'ordinateur alloue de la mémoire pour l'objet et ses attributs. On peut d'ailleurs obtenir l'adresse mémoire de notre objet créé x.

— La valeur de l'instance x et la couleur de l'instance x s'obtient avec *nom_objet.nom_attribut*

```
# Dans la console PYTHON
>>>x=Carte(5,'carreau')
>>>x.valeur
5
>>>x.couleur
'carreau'
```

2.2 Encapsulation : les accesseurs ou "getters"

On ne va généralement pas utiliser la méthode précédente *nom_objet.nom_attribut* permettant d'accéder aux valeurs des attributs car on ne veut pas forcement que l'utilisateur ait accès à la représentation interne des classes.

Pour utiliser ou modifier les attributs, on utilisera de préférence des méthodes dédiées dont le rôle est de faire l'interface entre l'utilisateur de l'objet et la représentation interne de l'objet (ses attributs).

Pour obtenir la valeur d'un attribut nous utiliserons la méthode des **accesseurs** (ou "getters") dont le nom est généralement : *getNom_attribut()*. Par exemple ici :

```
# Dans l'éditeur PYTHON

class Carte:    # Définition de la classe
    "Une carte d'un jeu de 32 ou 52 cartes"
    def __init__(self,valeur,couleur):    # méthode 1 : constructeur
        self.valeur=valeur # 1er attribut  valeur {de 2  à 14 pour as}
        self.couleur=couleur # 2e attribut {'pique','carreau','coeur','trefle'}

    def getAttributs(self):
        return (self.valeur,self.couleur)
```

```
# Dans la console PYTHON
>>>x=Carte(5,'carreau')
>>>y=Carte(14,'pique')
>>>x.getAttributs()
(5,'carreau')
>>>y.getAttributs()
(14,'pique')
```

— **L'encapsulation** désigne le principe de **regrouper des données brutes** avec un ensemble de **routines (méthodes)** permettant de les lire ou de les manipuler.

— But de l'encapsulation : cacher la représentation interne des classes.
  — pour simplifier la vie du programmeur qui les utilise ;
  — pour masquer leur complexité (diviser pour régner) ;
  — pour permettre de modifier celle-ci sans changer le reste du programme.
  — la liste des méthodes devient une sorte de mode d'emploi de la classe.

Exercice nº 1

Créer les accesseurs *getCouleur()* et *getValeur()* permettant de récupérer la valeur de la carte et la couleur.

2.3 Modifications contrôlées des valeurs des attributs : les mutateurs ou "setters"

On va devoir contrôler les valeurs attribuées aux attributs. Pour cela, on passe par des méthodes particulières appelées **mutateurs** (ou "setters") qui vont modifier la valeur d'une propriété d'un objet.

Le nom d'un mutateur est généralement : *setNom_attribut()*.

```
# Dans l'éditeur PYTHON
class Carte:    # Définition de la classe
        "Une carte d'un jeu de 32 ou 52 cartes
    couleurs= [chr(9824),chr(9825),chr(9826),chr(9827)]
    valeurs = {'2':2,'3':3,'4':4,'5':5,'6':6,'7':7,'8':8,'9':9,'10':10,
                        'V':11,'D':12,'R':13,'A':14}

    def __init__(self,valeur,couleur):    #constructeur
        self.valeur=valeur # 1er attribut  {de 2  à 14}
        self.couleur=couleur # {'pique', 'carreau', 'coeur', 'trefle'}

    def getAttributs(self): # méthode 2 : accesseur
        return (self.valeur,self.couleur)

    def getValeur(self): # méthode 3 : accesseur
        return   self.valeur

    def getCouleur(self): # méthode 4 : accesseur
        return   self.couleur

    def setValeur(self,v):  # méthode 5  : mutateur de l'attribut valeur
        if 2<=v<=14: # contrôle la validite de v
            self.valeur=v
            return True
        else:
            return False

    def __repr__(self):
        '''(Carte)->str
        retourne une representation de l'objet (print)'''
        return 'Carte : '+self.valeur+' de '+self.couleur

     def __eq__(self, autre):
        '''(Carte,Carte)->bool
        self == autre si la valeur et la couleur sont les memes'''
        return

     def __lt__(self,autre):  #(< less than)
            '''
            (Carte,Carte)->bool
        self <  autre  si la valeur de self est < à la valeur de autre'''
        return
```

Par exemple on va créer une instance **c1** : un 7 de cœur puis modifier sa valeur en la passant à 10.

```
# Dans la console PYTHON
>>>c1=Carte(7,'coeur')
>>>c1.getAttributs()
(7, 'coeur')
>>>c1.setValeur(10)
True
>>>c1.getAttributs()
(10, 'coeur')
```

Exercice n° 2

1. Créer le mutateur de l'attribut couleur : *setCouleur(self,c)* .
2. Créer une carte **c2** : un 4 de coeur.
3. Modifier la couleur de la carte **c2** en la passant à pique .
4. Modifier la valeur de la carte **c2** en la passant à 8.
5. Afficher les attributs de **c2**.

3 Premier bilan

Classe, Attributs, Méthodes, Accesseur et mutateurs

— Le type de données avec ses **caractéristiques** et ses **actions** possibles s'appelle **classe**.
— Les **caractéristiques (ou variables)** de la classe s'appellent les **attributs**.
— Les **actions possibles** à effectuer avec la classe s'appellent les **méthodes**.
— La **classe** définit donc les **attributs** et les actions possibles sur ces attributs, les **méthodes**.
— **Constructeur** : la manière ≪ normale ≫ de spécifier l'initialisation d'un objet est d'écrire un constructeur .
— **L'encapsulation** est un ensemble de **routines (méthodes)** permettant de lire ou de manipuler un objet.
— **Accesseur** ou ≪ **getter** ≫ : fournissent des informations relatives à l'état d'un objet, c'est-à-dire aux valeurs de certains de ses attributs (généralement privés) sans les modifier ;
— Un **Mutateur** ou **setter** : les mutateurs (en anglais mutator) qui modifient l'état d'un objet, donc les valeurs de certains de ses attributs.

Exercice n° 3 A faire en Python ...

1. Implémenter la class **Carte**.
2. Tester la fonction ci-dessous :

```
def testCarte():
        valetCoeur=Carte('Valet','COEUR')
        print('Nom:', valetCoeur.getNom())
        print('Couleur:', valetCoeur.getCouleur())
        print('Valeur:', valetCoeur.getValeur())
        valetCoeur.setNom('Dame')
        print('Nom modifie:', valetCoeur.getNom())
        print('Valeur modifiee:', valetCoeur.getValeur())
```

3. Écrire un programme qui va :
   a. Créer la carte 3 de COEUR que l'on nommera **c1**.
   b. Créer la carte 10 de PIQUE que l'on nommera **c2**.
   c. Afficher **c1** et **c2**.
   d. Tester si la valeur de **c1** est supérieure à celle de **c2**.
   e. Modifier la valeur de la carte **c2** en 4.
   f. Tester si les cartes **c1** et **c2** ont la même valeur.
   g. Tester si la valeur de **c1** est supérieure à celle de **c2**.

4 Notion d'agrégation

La conception d'une classe a pour but généralement de pouvoir créer des objets qui suivent tous le même modèle de fabrication. Un objet dans la vraie vie, par exemple votre stylo, est composé d'autres objets : une pointe (ou plume), un réservoir d'encre, éventuellement un capuchon et un ressort.... Votre stylo est ce qu'on appelle un **objet agrégat** et son réservoir d'encre est donc un **objet composant**. Nous avons créé une classe **Carte** qui permet de créer une carte à jouer standard. Un jeu de cartes est donc un **objet agrégat** de 32 ou 52 cartes à jouer. Si un jeu de carte est un objet, on peut donc définir une classe **JeuDeCartes**.

Cette classe a (entre autre) pour attributs :

— 32 ou 52 instances (objets) de la classe Carte.

— le nombre de cartes que le jeu contient.

```
# Dans l'éditeur PYTHON
from random import shuffle
class JeuDeCartes:
    def __init__(self):
        'initialise le paquet de 52 cartes'
        self.paquet = []          # paquet vide au debut
        self.nombreCarte=0
        for couleur in Carte.couleurs:
            # Carte.couleurs -> Va chercher la variable couleurs dans la classe Carte
            for valeur in Carte.valeurs:
            # Carte.valeurs -> Va chercher la variable valeurs dans la classe Carte
                # ajoute une Carte de valeur et couleur
                self.paquet.append(Carte(valeur,couleur))
                self.nombreCarte=self.nombreCarte+1

    # Accesseur de l'attribut self.nombreCarte
    def getNombreCartes(self):
        return

    # Accesseur de self.paquet
    def getPaquet(self):
        return

    def tireCarte(self):
        '''(JeuDeCartes)->Carte
        distribue une carte, la premiere du paquet'''
        return

    def battre(self):
        '''(JeuDeCartes)->None
        pour battre le jeu des cartes'''
        return

    def __repr__(self):
        '''retourne une representation de l'objet'''
        return str(self.paquet)
```

Exercice nº 4

1. Quel est le type de **self.paquet** ?
2. Quelle commande Python permet de :
   — Créer un jeu de carte **Jeu.**
   — Afficher le nombre de cartes de **Jeu**
   — Tirer une carte **c1**
   — Battre **Jeu**
   — Tirer une carte **c2**
   — Afficher **Jeu**

Exercice nº 5 A faire avec Python ...

1. Implémenter la classe **JeuDeCartes**
2. Vérifier le bon fonctionnement des commandes de l'exercice précédent.
3. Tester la fonction ci-dessous :

```
1  def testJeuDeCarte():
2          jeu52=JeuDeCartes()
3          jeu52.battre()
4          L=jeu52.getPaquet()
5          print(jeu52.getNombreCartes())
6          carte=L[2]
7          print('Nom:', carte.getNom())
8          print('Couleur:', carte.getCouleur())
9          print('Valeur:', carte.getValeur())
10         jeu52.tireCarte()
11         print(jeu52.getNombreCartes())
```

Exercice nº 6 A faire avec Python ...

1. Créer une classe **Joueur** ayant les attributs suivants :
   — **nom** : Nom du joueur ;
   — **nbCartes** : Correspond au nombre de cartes dans la main du joueur ;
   — **mainJoueur** : Liste des cartes (objets de type Carte) dans la main du joueur.
   Cette classe devra implémenter les méthodes suivantes :
   — **getNom()** : Accesseur de l'attribut **nom** ;
   — **getNbCartes()** : Accesseur du champ **nbCartes** ;
   — **setMain()** : Définit la main du joueur, donc la liste de ses cartes au début du jeu ;
   — **jouerCarte**() : Enlève et renvoie la dernière carte (objet de type Carte) de la main du joueur pour la jouer, ou retourne **None** s'il n'y a plus de cartes dans la main du joueur ;
   — **remporteCarte**(liste de cartes) : Fonction qui insère les cartes de la liste de cartes donnée en paramètre.
   — **__repr__()** : Qui affiche les cartes de la main du joueur.
2. Comme pour les exercices précedents, créer une fonction **testJoueur()** qui doit permettre de vérifier la bonne implémentation de la classe **Joueur**.

```
1  def testJoueur():
2
3
4
5
6              .
```

5 Exercice bilan

Exercice n° 7 Bilan

On suppose écrites les classes **Piece** et **Appartement**, dont on vous donne les en-têtes de méthodes :

```
# Dans l'éditeur PYTHON
class Piece:
    # nom est une string et surface est un float
    def __init__(self,nom,surface):
        self.nom=nom
        self.surface=surface

    def getSurface(self):
        return self.surface
    def getNom(self):
        return self.nom

    def setNom(self,nom): # nom est un string,
                ...

    def setSurface(self,s): # s est un float,
        ...

class Appartement:
    # nom est une string
    def __init__(self,nom):
        self.listeDePieces=[] # Liste d'instances de la classe Piece
        self.nom=nom

    def getNom(self):
        return self.nom

    def ajouter(self,piece):
        # pour ajouter une pièce de classe Piece
        ...

    def nbPieces(self):
        # pour avoir le nombre de pièces de l'appartement
        ...


    def getListePieces(self):
        # retourne la liste des pièces avec les surfaces
            ...


    def SurfaceTotale(self):
        # retourne la surface totale de l'appartement (un float)









                ...
```

1. En utilisant les deux classes ci-dessus, écrire (sur feuille) le code Pythan qui va :
   - **a.** créer une pièce « chambre1 » , de surface 20 m^2 et une pièce « chambre2 » », de surface 15 m^2 ,
   - **b.** créer une pièce « séjour » », de surface 25 m^2 et une pièce « sdb » », de surface 10 m^2 ,
   - **c.** créer une pièce « cuisine » », de surface 12 m^2 ,
   - **d.** créer un appartement « appart205 » qui contiendra toutes les pièces créées,
   - **e.** afficher la surface totale de l'appartement créé.
   - **f.** afficher la liste des pièces et surfaces de l'appartement créé.
2. **a.** Compléter les méthodes accesseurs et mutateurs de la classe Piece.
   - **b.** Compléter la méthode **ajouter(self,piece)**, qui permet d'ajouter une pièce de la liste de pièces présentes dans l'appartement.
   - **c.** Compléter la méthode **nbPieces(self)**, qui permet de retourner le nombre de pièce présentes dans l'appartement.
   - **d.** Compléter la méthode **getListePieces(self)**, qui renvoie la liste des pièces de l'appartement.
   - **e.** Compléter la méthode **getSurfaceTotale(self)**, qui renvoie la surface totale de l'appartement.
   - **f.** Créer un tableau qui classe les méthodes de ces deux classes selon leur type : constructeur, accesseur, mutateur ou autre.
3. Implémenter les classes ci-dessus et valider votre travail avec le test suivant :

```
# Dans l'éditeur PYTHON
a=Appartement('appt25')
p1=Piece("chambre", 11.1)
p2=Piece("sdbToilettes", 7)
p3=Piece("cuisine", 7)
p4=Piece("salon", 21.3)
print(p4.getNom(),p4.getSurface())
p1.setSurface(12.6)
a.ajouter(p1)
a.ajouter(p2)
a.ajouter(p3)
a.ajouter(p4)
print(a.getNom(),a.getListePieces())
print('nb pieces =', a.nbPieces(),', Surface totale =',a.SurfaceTotale())
```

Et devra retourner :

```
# Dans la console PYTHON

salon 21.3
appt25 [('chambre', 12.6), ('sdbToilettes', 7), ('cuisine', 7),
('salon', 21.3)]
nb pieces = 4 , Surface totale = 47.9
```

4. En s'inspirant du dernier **print**, créer une méthode __**repr**__ permettant d'avoir un descriptif de l'appartement. (nom de l'appartement, liste des pièces avec surface de chacune, nombre de pièces et surface totale)