Math93.com

TD NSI - Algorithmique Programmation Objet

Première partie

Cours - Programmation orientée objet

1 Introduction : Programmation procédurale, programmation orientée objet

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 état
- son comportement
- son identité

L'état est défini par les valeurs des attributs de l'objet à un instant t.

Par exemple, pour un téléphone, certains attributs sont variables dans le temps comme allumé ou éteint, d'autres sont invariants comme le modèle de téléphone.

Le comportement est défini par les méthodes de l'objet : en résumé, les méthodes définissent à quoi sert l'objet et/ou permettent de modifier son état.

L'identité est définie à la déclaration de l'objet (instanciation) par le nom choisi, tout simplement.

— En **programmation orientée objet**, on fabrique de nouveau types de données correspondant aux besoin du programme. On réfléchit alors aux caractéristiques des objets qui seront de ce type et aux actions possibles à partir de ces objets.

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()	comme avec la méthode de tri *liste.sort()*

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

1.3 Classe : vocabulaire

Classe

- 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**.
- Un objet du type de la classe s'appelle une **instance de la classe** et la création d'un objet d'une classe s'appelle une instanciation de cette 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()** ...
Les autres méthodes encadrées par les underscores sont spéciales.

1.4 Un peu d'histoire

La **programmation orientée objet**, qui fait ses débuts dans les années 1960 avec les réalisations dans le langages *Lisp*, a été formellement définie avec les langages *Simula* (vers 1970) puis *SmallTalk*.
Puis elle s'est développée dans les langages anciens comme le *Fortran*, le *Cobol* et 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 de 32 ou 52 cartes.
Par convention, une classe s'écrit toujours avec une majuscule.

```
# Dans l'éditeur PYTHON

class Carte:
    "Une carte d'un jeu de 32 ou 52 cartes"
```

— Une **méthode constructeur** commence toujours par :

```
def __init__(self,...):
```

Le paramètre particulier self est expliqué en fin de chapitre 2.1.
Avec deux tirets bas ou underscores (AltGr 8) de part et d'autre de init.

— On 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'}
```

— Création d'une **instance** de la **classe Carte** :

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

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.

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

— Par ailleurs, l'obtention de la valeur d'un attribut d'un objet se fait par l'utilisation de l'opérateur d'accessibilité point : | *nom_objet.nom_attribut* |
Cela peut se lire ainsi de droite à gauche nom_attribut appartenant à l'instance nom_objet

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

— La **variable self**.

self

La variable **self** , dans les méthodes d'un objet, désigne l'objet auquel s'appliquera la méthode. Elle représente l'objet dans la méthode en attendant qu'il soit créé.

```
# 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): # constructeur
        self.valeur=valeur    # 1er attribut
        self.couleur=couleur # 2e attribut
```

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

Dans cet exemple, la méthode **__init__ (constructeur)** est appelée implicitement. "**self**" fait référence à l'objet x dans la première ligne et à l'objet y dans la seconde.

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).
Les attributs sont alors en quelque sorte encapsulés dans l'objet, c'est à dire non accessibles directement par le programmeur qui a instancié un objet de cette classe.

Encapsulation

- **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.

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): # méthode 2 : permet d'accéder aux valeurs des attributs
        return (self.valeur,self.couleur)
```

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

On appelle le constructeur __init__ ( ) pour instancier (ou créer) les objets x et y, notamment en affectant des valeurs aux attributs.

On appelle les instances(=objets) x et y pour obtenir leurs adresses
ce sont bien deux objets différents !

On appelle la méthode getAttributs() de la classe Carte afin d'afficher les valeurs de chaque attribut des objets x et y.

```
> x=Carte(5,'carreau')
> y=Carte(14,'pique')
> x
<__main__.Carte object at 0x7f7f57d4ae90>
> y
<__main__.Carte object at 0x7f7f57634310>
> x.getAttributs()
(5, 'carreau')
> y.getAttributs()
(14, 'pique')
```

Exercice 2.1

Créer deux autres méthodes permettant de récupérer la valeur de la carte et la couleur avec les "getters" (accesseurs) : *getCouleur()* et *getValeur()*.

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"
    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.valeur

    def setValeur(self,v):  # mutateur avec contrôle
        if 2<=v<=14:
            self.valeur=v
            return True
        else:
            return False
```

Par exemple on va créer une carte c1, un 7 de coeur 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 2.2

1. Créer le mutateur de l'attribut couleur sous la forme *setCouleur(self,c)* .
2. Créer une carte c2, un Roi de coeur puis modifier sa valeur en la passant à une Dame.
3. Modifier la couleur de la carte c2 en la passant à pique .
4. Modifier la carte c2 en la passant à 8 de carreau.

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** désigne le principe de **regrouper des données brutes** avec un ensemble de **routines (méthodes)** permettant de les lire ou de les manipuler.
- **Accesseur** ou « **getter** » : une fonction qui retourne la valeur d'un attribut de l'objet. Par convention son nom est généralement sous la forme : *getNom_attribut()*.
- Un **Mutateur** ou **setter** : une procédure qui permet de modifier la valeur d'un attribut d'un objet. Son nom est généralement sous la forme : *setNom_attribut()*.

Exemples :

```
# 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}
        self.couleur=couleur    # 2e dans {'pique', 'carreau', 'coeur', 'trefle'}

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

    def setValeur(self,v):  # mutateur avec contrôle
        if 2<=v<=14:
            self.valeur=v
            return True
        else:
            return False
```

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**.

4.1 Exemple d'une agrégation par valeur (composition)

En parlant de stylo, voici un exemple très simple. Commençons par la définition de la **classe composant** Reservoir :

```
# Dans l'éditeur PYTHON
# Fichier reservoir.py

class Reservoir:
    ''' classe permettant de construire un réservoir d'encre
        pour des stylos toutes marques, toutes dimensions '''

    def __init__(self, couleur):
        ''' On se contente d'un seul paramètre pour l'exemple
            les dimensions ne seront donc pas incluses dans
            cette description '''
        # un seul attribut toujours par souci de clarté
        self.couleur = couleur

    # Accesseur de self.couleur
    def getCouleur(self):
        return self.couleur

    # Mutateur de self.couleur
    def setCouleur(self, couleur):
        self.couleur = couleur
```

Maintenant, voyons la **classe agrégat** Stylo et son utilisation de la classe Reservoir :

```
# Dans l'éditeur PYTHON
# Fichier stylo.py
from reservoir import *

class Stylo:
    ''' classe permettant de construire un stylo
        avec un réservoir d'encre.
        On ne s'occupe pas de ses autres caractérisques'''

    def __init__(self, couleur):
        ''' On se contente d'un seul paramètre pour l'exemple
            les dimensions ou autres composants ne seront donc
            pas inclus dans cette description '''
        self.reservoir = Reservoir(couleur)

    # Accesseur du self.couleur de self.reservoir
    def getCouleur(self):
        return self.reservoir.getCouleur()

    # Mutateur du self.couleur de self.reservoir
    def setCouleur(self, couleur):
        self.reservoir.setCouleur(couleur)
```

Maintenant, voyons comment on créé simplement un stylo rouge.

```
# Dans l'éditeur PYTHON
# Fichier main.py
from stylo import *

pen = Stylo("Rouge")
print(pen.getCouleur())
# Changeons la cartouche d'encre
pen.setCouleur("Bleu")

''' Attention, à éviter absolument même si possible,
on casse ici le principe d'encapsulation
mais le résultat est le même à l'affichage'''
print(pen.reservoir.getCouleur())
```

Le résultat donne ceci :

```
# Dans la console PYTHON
>>> %Run main.py
Rouge
Bleu
```

Notez bien une chose : dans le fichier principal de votre programme, ici **main.py**, vous n'avez pas importé le fichier **reservoir.py**, vous n'avez même pas besoin de savoir qu'il existe et encore moins de savoir comment il est conçu. Et pourtant, vous l'utilisez indirectement : en instanciant la classe **Stylo**, vous instanciez également la classe **Reservoir**. Et vous obtenez un objet stylo un peu plus complexe qu'il n'y parait.

Imaginez maintenant que vous vouliez un stylo quatre couleurs : oui, il vous faudra 4 instances de **Reservoir** dans **Stylo**. Ce qui entraînera une modification des **accesseurs** et **mutateurs**. Et certainement une méthode de sélection de la couleur. Les possibilités sont grandes.

Décomposition en fichiers

Cette architecture nécessite en général la création d'un fichier par classe. Elle permet de transformer une classe sans toucher aux autres. Où tout simplement, de se partager le travail dans une équipe.

4.2 Application à un jeu de cartes

Nous avons créé une classe **Carte** qui permet de créer une carte à jouer standard. Le paquet ou jeu de cartes est donc un objet constitué (composé, agrégé) 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. Ces instances sont toutes différentes en valeur et en couleur : on ne veut pas de doublon.

— le nombre de cartes que le jeu contient, soit 32, soit 52

```
# Dans l'éditeur PYTHON
#jeudecarte.py

class JeuDeCartes:
    def __init__(self,nombreCarte):
        # Tous les attributs ne sont peut être pas représentés
        self.nombreCarte=nombreCarte
        self.paquetCarte=[]
        ...

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

    # Accesseur de self.paquetCarte
    def getPaquet(self):
        ...

    # Méthode de création du paquet de cartes
    # on remplit la liste self.paquetCarte
    def creerPaquet(self):
        ...

    # Méthode de distribution d'une carte à la fois
    # retourne une instance de Carte
    def distribuerUneCarte(self):
        ...

    #méthode pour mélanger self.paquetCarte
    def melanger(self):
        ...
```

Exercice 4.3

1. Quel est le type de **self.paquetCarte** ?
2. Dans quelle méthode instancie t'on la classe **Carte** ?
3. Que peut on ajouter à la classe **JeuDeCartes** pour empêcher de donner une valeur différente de 32 ou 52 à l'attribut **self.nombreCarte** ?
4. Pourquoi est-ce important de faire cette vérification ?
5. Que pourrait on faire si la valeur passée dans le constructeur n'est pas correcte ?
6. Pourquoi la méthode **creerPaquet()** n'a pas besoin de paramètre autre que self ?
7. Quand utilise t'on la méthode **creerPaquet()** ?
8. Que manque t'il à ce code (en supposant les méthodes totalement définies) pour pouvoir être utilisé ? Faites une analogie avec l'exemple du stylo.
9. Instancier la classe **JeuDeCarte** et utiliser successivement les méthodes **melanger()** et **distribuerUneCarte()**

Deuxième partie

TD - Programmation orientée objet

Exercice 1 : utilisation d'objet

On suppose écrite la classe **Carte** dont on vous donne les en-têtes de méthodes. Cette classe diffère légèrement de la classe **Carte** vue plus haut, prenez le temps d'analyser ce code et de voir les différences :

```
class Carte:
  def __init__(self, nom, couleur):
    # Affectation de l'attribut nom  et  de l'attribut couleur
    couleur = ('CARREAU', 'COEUR', 'TREFLE', 'PIQUE')
    noms = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'Valet',  'Dame',
      'Roi', 'As']
    valeurs = {'2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9,
    '10': 10, 'Valet': 11, 'Dame': 12, 'Roi': 13, 'As': 14}

  def setNom(self, nom):
    # Mutateur de l'attribut nom (de la liste noms)

  def getNom(self):
    # renvoie le  nom  de la carte (de la liste noms): Accesseur

  def getCouleur(self):
    # renvoie la couleur de la carte (de la liste couleur): Accesseur

  def getValeur(self):
    # renvoie la valeur de la carte (du dictionnaire valeurs) : Accesseur

  def egalite(self, carte):
    ''' Renvoie True si les cartes self et carte ont même valeur, False sinon
    carte: Objet de type Carte    '''

  def estSuperieureA(self, carte):
    ''' Renvoie True si la valeur de self est supérieure à celle de carte,
    False sinon
    carte: Objet de type Carte
    '''

  def estInferieureA(self, carte):
    ''' Renvoie True si la valeur de self est inférieure à celle de carte,
    False sinon
    carte: Objet de type Carte
    '''
```

Écrire (sur feuille) un programme principal qui va :

1. Créer la carte Valet de COEUR que l'on nommera c1.
2. Afficher le nom, la valeur et la couleur de c1.
3. Créer la carte As de PIQUE que l'on nommera c2.
4. Afficher le nom, la valeur et la couleur de c2.
5. Modifier le nom de la carte c2 en Roi et afficher le nom, la valeur et la couleur de c2.
6. Créer la carte 8 de TREFLE que l'on nommera c3.
7. Comparer les cartes c1 et c2 puis c1 et c3.

Exercice 2 : utilisation d'objet

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

    # Accesseurs: retournent les attributs d'un objet de cette classe
    def getSurface(self):
        return self.surface
    def getNom(self):
        return self.nom

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

class Appartement:
    # nom est une string
    def __init__(self,nom):
        # L'objet est une liste de pièces (objets issus de la classe Piece)
        self.listeDePieces=[]
        self.nom=nom

    def getNom(self):
    # Accesseurs:
        return self.nom

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

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

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

    # retourne la liste des pièces avec les surfaces
    def getListePieces(self): # sous forme d'une liste de tuples
        ...
```

Écrire (sur feuille) un programme principal utilisant ces deux classes qui va :

1. créer une pièce « chambre1 » , de surface 20 m^2 et une pièce « chambre2 » », de surface 15 m^2 ,
2. créer une pièce « séjour » », de surface 25 m^2 et une pièce « sdb » », de surface 10 m^2 ,
3. créer une pièce « cuisine » », de surface 12 m^2 ,
4. créer un appartement « appart205 » qui contiendra toutes les pièces créées,
5. afficher la surface totale de l'appartement créé.
6. afficher la liste des pièces et surfaces de l'appartement créé.

Exercice 3 : rédaction des méthodes

On repose sur les mêmes classes que dans l'exercice 1 : **Piece** et **Appartement**. Les en-têtes de méthodes sont les mêmes mais vous allez devoir les compléter(...) :

```
# Proposition   expert:
# Dans l'éditeur PYTHON
class Piece:
    # nom est une string et surface est un float
    def __init__(self,nom,surface):
        # chaque objet a pour attributs le nom de la pièce(string)
        # et la surface de celle ci(float) en m2.
        # on doit rentrer le couple nom de la pièce et la surface
        # pour chaque pièce.
        ...
    # Accesseurs: retournent les attributs d'un objet de cette classe
    def getNom(self):
        ...
    def getSurface(self):
        ...
    # Mutateur: modifient les attributs, ici la surface d'une pièce déjà
    # renseignée
    def setSurface(self,s): # s est un float
        ...

class Appartement:
    # nom est une string
    def __init__(self,nom):
    #nomme l'appartement et une liste de pièces vide à remplir
        ...
    def ajouter(self,piece):
    # ajoute une pièce (instance(=objet) de la classe Piece)

    def nbPieces(self):
    # retourne le nombre de pièces de l'appartement
        ...
    def getSurfaceTotale(self):
    # retourne la surface totale de l'appartement (un float)
        ...
    def getListePieces(self): # retourne la liste des pièces
        ...
```

Consignes :

1. Écrire les méthodes constructeurs des deux classes.
   En cas de difficulté, rendez vous à la version intermédiaire. (Différenciation)
2. Finaliser la classe Piece.
   (a) Écrire les méthodes accesseurs et mutateurs de la classe Piece.
3. Finaliser la classe Appartement.
   (a) Écrire la méthode qui permet d'ajouter une pièce(ajouter(self,piece)) de la liste de pièces présentes dans l'appartement. (Rappel utile a)
   (b) Écrire la méthode qui permet de retourner le nombre de pièces(nbPieces(self)) présentes dans l'appartement. (Rappel utile b)
   (c) Écrire la méthode getSurfaceTotale(self), qui renvoie la surface totale de l'appartement. (Rappel utile b)
   (d) Écrire la méthode getListePieces(self), qui renvoie la liste des pièces de l'appartement.
4. Créer un tableau qui classe les méthodes de ces deux classes selon leur type : constructeur, accesseur, mutateur ou autre.

Aide

Rappels utiles :

— (a) pour ajouter un élément à la fin d'une liste : *nomDeListe.append(élément)*;

— (b) la longueur d'une listel s'obtient avec *len(l)*.

Le test sera le 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
```

```
# Proposition  intermédiaire:
# Dans l'éditeur PYTHON

class Piece:
    # nom est une string et surface est un float
    def __init__(self,nom,surface):
        # chaque objet a pour attributs le nom de la pièce(string)
        # et la surface de celle ci(float) en m2.
        # on doit rentrer le couple nom de la pièce et la surface
        # pour chaque pièce.
        self.nom=nom
        self.surface=surface
    # Accesseurs: retournent les attributs d'un objet de cette classe
    def getNom(self):
        return self.surface
    def getSurface(self):
        ...
    # Mutateur: modifient les attributs, ici la surface d'une pièce
    # déjà renseignée
    def setSurface(self,s): # s est un float
        ...

class Appartement:
    # nom est une string
    def __init__(self,nom):
    #nomme l'appartement et une liste de pièces vide à remplir
        self.listeDePieces=[]
        self.nom=nom
    def ajouter(self,piece):
    # ajoute une piece (instance(=objet) de la classe Piece)

    def nbPieces(self):
    # retourne le nombre de pièces de l'appartement
        ...
    def getSurfaceTotale(self):
    # retourne la surface totale de l'appartement (un float)
        ...
    def getListePieces(self): # retourne la liste des pieces
        ...
```

Troisième partie

Corrections exercices du cours

Correction de l'exercice 2.1

Créer deux autres méthodes permettant de récupérer la valeur de la carte et la couleur avec les "getters" (accesseurs) : *getCouleur()* et *getValeur()*.

```
def getValeur(self):
    return self.valeur

def getCouleur(self):
    return self.valeur
```

Correction de l'exercice 2.2

1. **Créer le mutateur de l'attribut couleur sous la forme *setCouleur(self,c)* .**

```
def setCouleur(self,c):
    self.couleur=c
```

2. **Créer une carte c2, un Roi de coeur puis modifier sa valeur en la passant à une Dame.**

```
# Dans la console PYTHON
c2=Carte(13,'coeur')
c2.setValeur(12)
```

3. **Modifier la couleur de la carte c2 en la passant à pique .**

```
# Dans la console PYTHON
c2.setCouleur('pique')
```

4. **Modifier la carte c2 en la passant à 8 de carreau.**

```
# Dans la console PYTHON
c2.setValeur(8)
c2.setCouleur('carreau')
```

Correction de l'exercice 4.3

1. **Quel est le type de self.paquetCarte ?**
   C'est une liste d'objets de type Carte.
2. **Dans quelle méthode instancie t'on la classe Carte ?**
   Dans creerPaquet, on remplit la liste self.paquetCarte et donc, c'est ici que la classe Carte est instanciée.
3. **Que peut on ajouter à la classe JeuDeCartes pour empêcher de donner une valeur différente de 32 ou 52 à l'attribut self.nombreCarte ?**
   On peut ajouter un mutateur setNombreCarte qui vérifiera la validité du nombre passé en paramètre
4. **Pourquoi est-ce important de faire cette vérification ?**
   On ne veut pas créer n'importe quoi comme jeu de carte et donc, on veut empêcher de créer un jeu de -12 cartes ou de 25634789 cartes
5. **Que pourrait on faire si la valeur passée dans le constructeur n'est pas correcte ?**
   On peut créer un paquet de 52 cartes par défaut, dans la mesure où un paquet vide semble ne pas être une bonne idée.
6. **Pourquoi la méthode creerPaquet() n'a pas besoin de paramètre autre que self ?**
   Parce que cette méthode n'utilise que les attributs de la classe et n'a donc pas besoin d'informations extérieures
7. **Quand utilise t'on la méthode creerPaquet() ?**
   Dans la méthode __init__(), juste après la déclaration de la liste self.paquetCarte
8. **Que manque t'il à ce code (en supposant les méthodes totalement définies) pour pouvoir être utilisé ? Faites une analogie avec l'exemple du stylo.**
   Si on n'ajoute pas la ligne from Carte import *, la classe JeuDeCartes ne pourra pas être utilisée
9. **Instancier la classe JeuDeCarte et utiliser successivement les méthodes melanger() et distribuerUneCarte()**

```
# Dans la console PYTHON
>>> mon_jeu = JeuDeCarte(52)
>>> mon_jeu.melanger()
>>> mon_jeu.distribuerUneCarte()
```

Quatrième partie

Correction TD - Programmation orientée objet

Correction de l'exercice 1 : utilisation d'objet

1. Créer la carte Valet de COEUR que l'on nommera c1.
2. Afficher la valeur de c1.
3. Créer la carte As de PIQUE que l'on nommera c2.
4. Afficher la valeur de c2.
5. Modifier le nom de la carte c2 en Roi et afficher la valeur de c2.
6. Créer la carte 8 de TREFLE que l'on nommera c3.
7. Comparer les cartes c1 et c2 puis c1 et c3.

```
# Dans l'éditeur PYTHON
#1. Créer la carte Valet de COEUR que l'on nommera c1.
c1=Carte('Valet', 'COEUR')

#2. Afficher le nom, la valeur et la couleur de c1.
print('c1 = ',c1.getNom(), c1.getValeur() , c1.getCouleur() )

#3. Créer la carte As de PIQUE que l'on nommera c2.
c2=Carte('As', 'PIQUE')

#4. Afficher le nom, la valeur et la couleur de c2.
print('c2 = ',c2.getNom(), c2.getValeur() , c2.getCouleur() )

#5. Modifier le nom de la carte c2 en Roi et afficher les attributs de c2
c2.setNom('Roi')
print('c2 = ',c2.getNom(), c2.getValeur() , c2.getCouleur() )

#6. Créer la carte 8 de TREFLE que l'on nommera c3.
c3=Carte('8', 'TREFLE')

#7. Comparer les cartes c1 et c2 puis c1 et c3.
print(c1.estSuperieureA(c2))
print(c1.estSuperieureA(c3))
```

Ce qui donnerait :

```
# Dans la console PYTHON
c1 =  Valet 11 COEUR
c2 =  As 14 PIQUE
c2 =  Roi 13 PIQUE
False
True
```

Correction de l'exercice 3

```
class Piece:
    def __init__(self,nom,surface):
        self.nom=nom
        self.surface=surface
    def getSurface(self):
        return self.surface
    def getNom(self):
        return self.nom
    def setSurface(self,s):
        self.surface=s

class Appartement:
    def __init__(self,nom):
        self.listeDePieces=[]
        self.nom=nom
    def getNom(self):
        return self.nom
    def ajouter(self,piece):
        self.listeDePieces.append(piece)
    def nbPieces(self):
        return len(self.listeDePieces)
    def getSurfaceTotale(self):
        total=0
        for piece in self.listeDePieces:
            surf=piece.getSurface()
            total=total+surf
        return total
    def getListePieces(self):
        L=[]
        for piece in self.listeDePieces:
            surf=piece.getSurface()
            nom=piece.getNom()
            L.append((nom,surf))
        return L
```

Tableau de classement des méthodes de ces deux classes selon leur type :

- **constructeurs** : les deux init
- **accesseurs** : Piece.getSurface(), Piece.getNom(), Appartement.getNom() Appartement.getSurfaceTotale(), Appartemnt.getListePieces()
- **mutateurs** : Piece.setSurface(), Appartement.ajouter(),
- **autre** : Appartement.nbPieces() Et oui, nbPieces N'EST PAS un attribut de la classe Appartement, cette valeur n'est stockée nulle part, elle est juste calculée par la méthode len de self.listeDePieces