La Programmation Orientée Objet

Structures de données

NSI - Tle

COURS

Contenu : Vocabulaire de la programmation objet : classes, attributs, méthodes, objets.

Capacités attendues : Écrire la définition d'une classe.

Accéder aux attributs et méthodes d'une classe.

I. Introduction : Programmation procédurale, programmation orientée objet

I.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 besoins 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**.

I.2. Classe : un premier exemple avec le type list

Le type de données *list* est une classe.

```
ma_liste = [5,7,3]
type(ma_liste)

>>>list
```

`ma_liste` est une liste, ou plus précisément un **objet** de type `list` . Et en tant qu'objet de type `list` , il est possible de lui appliquer certaines fonctions prédéfinies (qu'on appellera **méthodes**) : `ma_liste.sort()`

La syntaxe utilisée (le . après le nom de l'objet) est spécifique à la POO. Chaque fois que vous voyez cela, c'est que vous êtes en train de manipuler des objets.

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()*

```
ma_liste = [5,7,3]
ma_liste.sort()
print(ma_liste)

>>>[3,5,7]
```

I.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**.
- 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)** :

```
dir(ma_liste)
>>>['__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. Ce sont des méthodes privées, a priori non destinées à l'utilisateur. Les méthodes publiques, utilisables pour chaque objet de type *list*, sont donc *append*, *clear*, jusqu'à *sort*...

Comment savoir ce que font les méthodes ? Si elles ont été correctement codées (et elles l'ont été), elles possèdent une *_docstring_*, accessible par :

```
ma_liste.append.__doc__
>>>'Append object to the end of the list.'
m.reverse.__doc__
```

I.4. Un peu d'histoire

La **programmation orientée objet**, qui fait ses débuts dans les années 1960 avec les réalisations dans le langage *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 ou C++*.

II. Création d'une classe : pas à pas

II.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.

```
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 II.1. Avec deux tirets bas ou underscores de part et d'autre de init.

- On va définir les **attributs de la carte** qui seront :
  - sa valeur 2,3⋯ ,10,11 pour Valet,12 pour Dame,13 pour Roi et 14 pour As
  - sa couleur (Carreau, Coeur, Trèfle, Pique).

```
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** :

```
>>> carte1 = 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.

```
>>> carte1 = Carte(5,'carreau')
>>> carte1
<__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***

```
>>>carte1 = Carte(5,'carreau')
>>>carte1.valeur
5
>>> carte1.couleur
'carreau'
```

- La **variable 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.

```
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

>>> carte1 = Carte(5,'carreau')
>>> carte2 = Carte(14,'pique')
```

Dans cet exemple, la méthode **__init__ (constructeur)** est appelée implicitement. "**self**" fait référence à l'objet ***carte1*** dans la première ligne et à l'objet ***carte2*** dans la seconde.

II.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 :

```
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)
```

```
>>> carte1 = Carte(5,'carreau')
>>> carte2 = Carte(14,'pique')
>>> carte1.getAttributs()
(5,'carreau')
>>> carte2.getAttributs()
(14,'pique')
```

II.3. Exercice 1.

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

II.4. 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()*.

```
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.couleur


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

>>> carte1 = Carte(7, 'coeur')
>>> carte1.getAttributs()
(7, 'coeur')
>>>carte1.setValeur(10)
True
>>> carte1.getAttributs()
(10, 'coeur')
```

Par exemple on va créer une carte *carte1*, un 7 de coeur puis modifier sa valeur en la passant à 10.

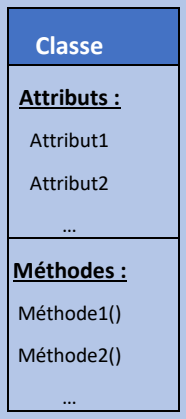
II.5. Exercice 2.

1. Créer le mutateur de l'attribut couleur sous la forme *setCouleur(self,c)* .
2. Créer une carte carte2, un Roi de coeur puis modifier sa valeur en la passant à une Dame.
3. Modifier la couleur de la carte carte2 en la passant à pique.
4. Modifier la carte carte2 en la passant à 8 de carreau.

III. 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 :

```
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
```

IV. 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**.

IV.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** :

```
# 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** :

```
# 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)
```

Création simple d'un stylo rouge :

```
# 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())

>>> %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.

IV.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 autres) 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

```
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

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 ? Faire une analogie avec l'exemple du stylo.