Chapitre 2 : Programmation orientée objet avancée

La programmation orientée objet répond aux nécessités de l'informatique professionnelle. Elle a pour but avoir une meilleure gestion des sous-programmes.

1. Objet et Classe :

L'idée fondamentale de la POO est que l'on considère des objets, à savoir une association de données et de fonctions. Les fonctions sont appelées les méthodes de l'objet, les données ses attributs. Exemple, on veut manipuler des points du plan.

Un point est défini par deux coordonnées, donc deux données. Les méthodes que l'on veut à propos d'un point sont, par exemple, de l'initialiser, le déplacer et l'afficher.

Une classe (des points) est un modèle à partir duquel on peut générer un ensemble d'objets partageant des méthodes et des attributs communs. Les objets appartenant à celle-ci sont les instances de cette classe. L'instanciation (réalisation) est la création d'un objet d'une classe.

Avant de manipuler un objet, il faut d'abord définir la classe dont il provient, c'est-à-dire décrire les différents attributs et méthodes de la classe.

Définition de la classe :

Une classe se définit grâce au mot clef "class" suivi du nom de celle-ci. A la suite, les méthodes sont déclarées.

Syntaxe:

```
""" Définition de la classe point """
class Point :
""" déclaration des méthodes """
```

Exemple :

```
# Définition d'une classe appelée Point
class Point:
```

```
    # Méthode pour initialiser les coordonnées du point
    def initialise(self, x, y):
        self.x = x  # Attribue la valeur x à l'attribut x de l'objet
        self.y = y  # Attribue la valeur y à l'attribut y de l'objet

    # Méthode pour déplacer le point de dx sur l'axe x et de dy sur l'axe y
    def deplace(self, dx, dy):
        self.x += dx  # Déplace x de dx
        self.y += dy  # Déplace y de dy

    # Méthode pour afficher les coordonnées actuelles du point
    def affiche(self):
        print("Le point est en (", self.x, ",", self.y, ")")  # Affiche les coordonnées
                                                                du point

""" Utilisation de la classe """
p = Point()             # Crée une instance de la classe Point
p.initialise(2,3)       # Initialise le point avec les coordonnées x=2 et y=3
p.affiche()             # Affiche "Le point est en ( 2 , 3 )"
p.deplace(1, -1)        # Déplace le point de +1 sur l'axe x et -1 sur l'axe y
p.affiche()             # Affiche "Le point est en ( 3 , 2 )"
```

Remarques :

- Le nom de la classe est Point.
- self désigne le nom de l'objet lui-même dans la méthode.
- La classe comprend deux attributs et trois méthodes.
- p=Point() # crée un objet de type Point.
- p=Point() # appel de la méthode initialise, Python passe automatiquement p comme premier argument puis initialise x et y.

Pour avoir des attributs privés, c'est –à-dire, ne sont pas accessibles directement en dehors de la classe. Deux soulignés (__) sont utilisés avant leur nom.

```
class Point:
    def initialise (self, x, y):
        self.__x = x
        self.__y = y

    def deplace(self, dx, dy):
        self.__x = self.__x + dx
        self.__y = self.__y + dy

    def affiche(self):
        print("Le point est en (", self.__x, ",", self.__y, ")")

p= Point()
p.initialise(2, 3)
a.affiche()
a.deplace(1, -1)
a.affiche()
```

2. Constructeur :

L'initialisation permet d'affecter des valeurs à des variables lors de leur déclaration. En POO, on déclare un objet dont les données membres sont indéterminées, puis on l'initialise en appelant la fonction membre initialise. Si on veut initialiser un objet lors de son instanciation, on définit dans la classe une fonction membre qui se substitue à la fonction membre initialise et que l'on appellera un constructeur.

Un constructeur doit porter le nom imposé par le langage Python : **__init__()**.

Exemple :

```
class Point:
```

```
    def __init__(self, a, b):
        self.x = a
        self.y = b

    def deplace(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

 def affiche(self):
 print("Le point est en (", self.x, ",", self.y, ")")

a = Point(2, 3)
a.affiche()
a.deplace(1, -1)
a.affiche()
```

*** Constructeur par défaut :**

Le constructeur par défaut est un constructeur sans paramètres (autres que self) ou avec paramètres par défauts.

Exemple :

```
Class Point :

    def __init__(self):

        self.x = 0

        self.y = 0

#Utilisation :

p = Point ()   # initialisation via constructeur

p.affiche()         # Le point est en ( 0,0)


p.deplace(1, -1)

p.affiche()         # Le point est en ( 1 ,-1)
```

Ou :

```
Class Point :

    def __init__(self, x=0, y=0):
```

```
        self.x = x
        self.y = y
```

#Utilisation :

```
p = Point (2,3) # initialisation via constructeur
p.affiche()         # Le point est en ( 2 , 3 )
p1 = Point (2) # initialisation via constructeur
p1.affiche()         # Le point est en ( 2 , 0 )
p2 = Point () # initialisation via constructeur
p2.affiche()         # Le point est en ( 0,0)
p.deplace(1, -1)
p.affiche()         # Le point est en ( 3 ,2)
```

3. Cas des objets transmis en arguments d'une fonction membre.

Une fonction membre reçoit l'objet l'ayant appelé. Mais il est possible de lui transmettre explicitement un argument (ou plusieurs) du type de sa classe.

Par exemple, supposez que nous souhaitions, au sein d'une classe vecteur, introduire une fonction membre nommée produitvect, chargée de calculer le produit scalaire de deuc vecteurs.

Son appel au sein d'un programme se présentera sous la forme : v1. produitvect (v2). C'est l'équivalent de Vecteur.produitvect (v1,v2) où v1est l'objet ayant appelé la méthode donc (self) et v2 est v.

Dans produitvect, nous devons donc calculer le produit scalaire des composantes de l'objet fourni implicitement lors de son appel (ses membres sont désignés, comme d'habitude, par self.x et self.y) avec celles de l'objet fourni en argument, dont les membres sont désignés par v.x et v.y.

```
class Vecteur :
    def __init__(self, x=0, y=0):
        self.x = a
        self.y = b

    def produitvect(self, v) :
        return self.x * v.x + self.y * v.y

    def affiche(self) :
        print("Vecteur (", self.x, ",", self.y, ")")

#utilisation
v1 = Vecteur(2, 3)
v2 = Vecteur(4, -1)

v1.affiche()      # Vecteur ( 2 , 3 )
v2.affiche()      # Vecteur ( 4 , -1 )

ps = v1.produit_scalaire(v2)
```

4. Accesseurs et mutateurs :

L'accès aux attributs privés d'une classe se fera par l'intermédiaire de méthodes de la classe telles que initialise, affiche et les constructeurs ou encore les accesseurs et les mutateurs. On les nomme respectivement get_xxx() pour l'accès et set_xxx() pour modifier le contenu d'un attribut.

4.1 Accesseurs :

Un accesseur est une méthode permettant d'accéder au contenu d'un attribut privé (protégé).

4.2 Mutateurs :

Un mutateur est une méthode permettant de modifier le contenu d'un attribut.

Exemple:

```
class Point:
    def __init__(self, a, b):
        self.__x = a
        self.__y = b

    def get_x(self):
        return self.__x

    def set_x(self, x):
        self.__x = x

    def get_y(self):
        return self.__y

    def set_y(self, y):
        self.__y = y

a = Point(3, 7)
print("a : abscisse =", a.get_x())
print("a : ordonnee =", a.get_y())
a.set_x(6)
a.set_y(10)
print("a : abscisse =", a.get_x())
print("a : ordonnee =", a.get_y())
```

5. Principes fondamentaux de la POO : les quatre principes fondamentaux de la POO sont : l'encapsulation, héritage, abstraction, et polymorphisme.

5.1. Encapsulation : consiste à protéger les données d'un objet.

Exemple : (Voir : 4. Accesseurs et mutateurs)

Si on tente de l'extérieur de la classe :

print(p.__x) # ça affiche : Erreur : attribut privé

- Les attributs __x et __y sont encapsulés(ne peuvent pas être modifiés de l'extérieur).
- On utilise des méthodes d'accès get et set,.. pour la lecture et la modification.

5.2. Héritage : Consiste à se servir d'une classe préexistante pour en créer une nouvelle, qui héritera toutes ses propriétés mais pourra modifier certaines d'entre elles.

Exemple :

```
class Point3D(Point):
    def __init__(self, x=0, y=0, z=0):
        super().__init__(x, y)  # On initialise x et y avec la classe parente
        self.z = z

    def deplace(self, dx, dy, dz):
        """Déplace le point 3D"""
        super().deplace(dx, dy)  # Déplacement (2D) via la classe parente
        self.z += dz             # Ajout du déplacement en z

    def affiche(self):
        """Affiche les coordonnées du point 3D"""
         print("Le point est en (", self.x, ",", self.y,  ",",self.z, ")")

         #Utilisation :
         # Création d'un point 2D
         p2 = Point(1, 2)
         p2.affiche()      # x = 1, y = 2
         p2.deplace(3, -1)
         p2.affiche()      # x = 4, y = 1

         # Création d'un point 3D
         p3 = Point3D(1, 2, 3)
         p3.affiche()      # x = 1, y = 2, z = 3
```

5.3. Abstraction :

Une classe abstraite en programmation orientée objet est une classe qui ne peut pas être instanciée directement, mais qui sert de modèle pour d'autres classes.

Elle contient souvent des méthodes abstraites, c'est-à-dire des méthodes déclarées mais non implémentées, que les classes dérivées doivent redéfinir.

Elle est essentielle lorsqu'on souhaite imposer certaines méthodes à toutes les classes dérivées, garantissant ainsi une certaine cohérence et structure dans le code.

Pour créer une classe abstraite en python, on utilise le module abc (Abstract Base Classes) qui fournit les mécanismes nécessaires pour définir les méthodes abstraites.

Une méthode devient abstraite quand elle est décorée avec le décorateur @abstractmethod.

Exemple :

```
from abc import ABC, abstractmethod
"""Classe abstraite."""
class Point(ABC):
    def __init__(self, x, y):
        self.x = x
        self.y = y

"""Méthode abstraite : doit être implémentée dans les sous-classes."""
   @abstractmethod
    def affiche(self):
        pass

"""Méthode abstraite : déplace le point."""
    @abstractmethod
    def deplace(self, dx, dy):
        pass

#sous classe concretes
class Point2D(Point):
    def affiche(self):
        print("Le point est en (", self.x, ",", self.y, ")")

    def deplace(self, dx, dy):
        self.x += dx
```

```
        self.y += dy

#utlilisation :

p = Point2D(3, 4)
p.affiche()       # Affiche : Point2D(3, 4)
p.deplace(2, -1)
p.affiche()       # Affiche : Point2D(5, 3)
```

5.4. Polymorphisme : permet d'utiliser une même fonction pour plusieurs types d'objets.

Exemple :

```
# Liste contenant à la fois des points 2D et 3D.
points=[Point(1,2), Point3D(0,4,2), Point(-2,2), Point3D(1,2,3)]
for p in points :
    p.affiche() #appelle automatiquement  la bonne version selon le type d'objet
```

La méthode affiche accepte comme atgument un objet de type Point ou de type 3D.