Chapitre 1

Programmation orienté objet

Table des matières

1 Définition d'une classe

1.1 Définition, exemple

Définition 1 :

Une classe définie un objet pour lequel nous allons donner des caractéristiques (les **attributs**) et des actions (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.
On dit que les attributs et les méthodes sont encapsulés dans la classe.

Exemples 1 : Construction d'un objet représentant les intervalles bornés :

```
class Intervalle :
    def __init__ ( self , debut , fin ) :
        """ premiere methode : constructeur  """
        self.debut = debut # attribut
        self.fin = fin # attribut

    def get_debut(self):
        """ methode de recuperation d un attribut : accesseur """
        return( self.debut )

    def get_fin(self):
        """ methode de recuperation d un attribut : accesseur"""
        return( self.fin )

    def set_debut(self , nouveau_debut):
        """ methode de modification : mutateur """
        self.debut = nouveau_debut

    def set_fin(self , nouveau_fin):
        """ methode de modification : mutateur """
        self.fin = nouveau_fin
```

- La variable `self` fait référence à l'objet lui-même.
- La première méthode est essentielle pour définir les attributs de l'objet (ici, les valeurs sont données en argument lors de la construction de l'objet).
- Les méthodes `get` sont appelées **accesseurs** (ou **getter**). Elles permettent de renvoyer les attributs.
- Les méthodes `set` sont appelées **mutateurs** (ou **setter**). Elles permettent de modifier les attributs.

Exemples 2 : Création d'une instance :

```
>>> i_1 = Intervalle(8,12) # on precise le debut et la fin
>>> i_1.get_debut()
8
```

```
>>> i_1.get_fin()
12
>>> i_1.set_debut(9)
>>> i_1.get_debut()
9
```

Exercice 1 : Pour la classe intervalle définie précédemment, définir les méthodes suivantes :

1. La méthode `largeur` qui renvoie la largeur d'un intervalle.
2. La méthode `centre` qui renvoie le centre d'un intervalle.
3. La méthode `translation` qui opère une translation de l'intervalle d'un entier donné en argument. (par exemple, la translation de l'intervalle [8,12] par la valeur 3 donne l'intervalle [11,15]).
4. La méthode `union` qui prend en argument un deuxième intervalle, et renvoie l'intervalle union des deux. (on suppose ici, que l'union des deux intervalles est un intervalle)

Exercice 2 :

1. Définir la classe `Cercle`, dont on donnera le centre (sous forme d'un tableau à deux arguments) et le rayon.
2. Écrire les accesseurs aux deux attributs.
3. Écrire les mutateurs aux deux attributs.
4. Écrire la méthode `surface` qui renvoie la valeur de la surface d'un cercle.
5. Écrire la méthode `est_sur_le_cercle` qui vérifie l'appartenance d'un point sur un cercle.

1.2 Histoire de l'informatique

La programmation orientée objet (POO), 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 Méthodes spéciales

2.1 Méthode d'affichage

Une méthode spécifique est utilisée pour permettre à l'objet d'être affiché grace à la fonction `print` :

Exemples 3 : Affichage d'un intervalle :

```
class Intervalle :
    def __init__ ( self , debut , fin ) :
        """ premiere methode : constructeur  """
        self.debut = debut # attribut
        self.fin = fin # attribut

    def __str__(self) :
        return "[%s,%s]" %(str(self.debut ) ,str(self.fin) )
```

Ce qui donne :

Exemples 3 : Affichage d'un intervalle :

```
>>> i_1 = Intervalle(8,12)
>>> print(i_1)
 [ 8 , 12 ]
```

2.2 Surcharge d'opérateur

La surcharge permet à un opérateur de posséder un sens différent suivant le type de ses opérandes.
Les méthodes spéciales `__add__` et `__sub__` permettent aux instances l'utilisation du + et du - :

Exemples 3 : Notion de surcharge :

```
class Intervalle :
    def __init__ ( self , debut , fin ) :
        """ premiere methode : constructeur  """
        self.debut = debut # attribut
        self.fin = fin # attribut

    def __add__(self, valeur ) :
        """ permet l utilisation du + """
        self.debut = self.debut + valeur
        self.fin = self.fin + valeur

    def __sub__(self, valeur ) :
        """ permet l utilisation du - """
        self.debut = self.debut - valeur
        self.fin = self.fin - valeur
```

Ce qui donne :

Exemples 3 : Notion de surcharge : :

```
>>> i = Intervalle(8,12)
>>> i+3
>>> i.get_debut()
11
>>> i.get_fin()
15
>>> i-7
>>> i.get_debut()
4
>>> i.get_fin()
8
```

On dispose aussi d'autres méthodes spéciales de surcharge :

Méthode spéciale	Utilisation
`__neg__`	-objet1
`__add__`	objet1 + objet2
`__sub__`	objet1 - objet2
`__mul__`	objet1 * objet2
`__div__`	objet1 / objet2
`__floordiv__`	objet1 // objet2

3 Notion d'agrégation

Définition 2 :

Une agrégation est une association non symétrique entre deux classes (l'agrégat et le composant).

Commençons par la définition de la classe composant

Exemples 4 : Définition de la classe composant :

```
class Point :
    def __init__(self , x , y ) :
        self.x = x
        self.y = y

    def __str__(self) :
        return "(" + str(self.x) + "␣,␣" + str(self.y ) + ")"
```

Une fois la classe `Point` crée, on peut l'utiliser pour une classe agrégat :

Exemples 4 : Définition de la classe agrégat :

```
class Droite :
    def __init__ ( self, a , b ) :
    """ a et b sont des points """
        self.pente = ( a.y - b.y) / ( a.x - b.x)
        self.oao = a.y - self.pente * a.x

    def appartient(self , m ) :
        return ( m.y == self.pente * m.x + self.oao)
```