Séance 4: Programmation orientée objet

Philipp Ahrendt

December 2, 2025

Contents

1 Classes et objets **1**

2 Héritage **5**

1 Classes et objets

En python, toutes les données sont représentées par des **objets**. Un objet est défini par:

- Son *type*
- Une représentation interne
- Une *interface*

La représentation interne est l'ensemble des données regroupées dans l'objet.

Exemple (en C, avec pointeurs): Une représentation interne possible pour des listes

```
struct chain {
  int head; /* premier élément de la liste */
  list *tail; /* adresse du prochain bloc, on fait une chaine */
};

struct list {
  chain *content;
};
```

Exemple python: Représentation interne pour un objet de type `coordonnee`.

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

c = coordonnee(4,2)

x = c.x
y = c.y

print(x,y)
```

L'interface sont les fonctions qui permettent d'interagir avec un objet (accéder à des valeurs, le modifier etc)

Par exemple, on interagit avec une liste `L` avec `L[i]`, `L.append`, `L.extend` ...

Une **classe** est une définition pour des objets, regroupant le type (le nom de la classe), la représentation interne, et l'interface.

Exemple: Une classe de coordonnées, avec des opérations typiques.

```
import math

def arg(x,y):
    if x == 0 and y == 0:
        return None
    elif x == 0:
        if y > 0: return math.pi/2
        else: return -math.pi/2
    elif x > 0:
        return math.atan(y/x)
    else:
        return math.atan(-y/x)


class coordonnee(object):
    # la représentation est définie dans la fonction __init__
    def __init__(self, x, y):
        # Représentation en coordonnées cartésiennes
        self.x = x
```

```
        self.y = y

        # Représentation en coordonnées polaires
        self.norme = (x**2 + y**2)**(1/2)
        self.angle = arg(x,y)

    # Interface
    def add_x(self, x):
        self.x += x
        self.norme = (self.x**2 + self.y**2)**(1/2)
        self.angle = arg(self.x, self.y)

    def add_y(self, y):
        self.y += y
        self.norme = (self.x**2 + self.y**2)**(1/2)
        self.angle = arg(self.x, self.y)

    def mult(self, a):
        self.x *= a
        self.y *= a
        self.norme *= a

    def rotation(self, theta):
        if self.angle is not None:
            if self.angle + theta > math.pi:
                self.angle += theta - 2*math.pi

            elif self.angle + theta < -math.pi:
                self.angle += theta + 2*math.pi

            else:
                self.angle += theta

            self.x = math.cos(self.angle)*self.norme
            self.y = math.sin(self.angle)*self.norme


c = coordonnee(4,4)
c.rotation(math.pi/2)
```

```
print(c.angle)
print(c.x, c.y)


# NE PAS FAIRE: manipuler directement les données internes en dehors de la définition d
c = coordonnee(1,1)
c.x += 3
c.y += 3

print(f'la norme de ({c.x},{c.y}) est {c.norme}')
```

Cet exemple illustre un principe important en programmation orientée objet:

La représentation interne d'un objet est géré uniquement par l'objet lui-même, toute interaction avec l'extérieur passe par son interface

On peut aussi définir les opération usuels, comme `+`, `==`, `print`, etc. pour des nouvelles classes, via des fonctions spéciales:

Fonction	Utilisation
`__add__(self,other)`	`self + other`
`__sub__(self,other)`	`self - other`
`__eq__(self,other)`	`self == other`
`__lt__(self,other)`	`self < other`
`__gt__(self,other)`	`self > other`
`__len__(self)`	`len(self)`
`__str__(self)`	`print(self)`

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

    def __add__(self,other):
        return coordonnee(self.x+other.x, self.y+other.y)

    def __eq__(self,other):
        return self.x == other.x and self.y == other.y

    def __str__(self):
```

```
        return f'({self.x},{self.y})'


c1 = coordonnee(1,3)
c2 = coordonnee(4,5)

print(c1+c2)
```

2 Héritage

En programmation orientée objet, le *héritage* est un moyen de définir des sous-classes d'une classe.

Une sous-classe *hérite* les données et l'interface de sa classe de base. On peut en rajouter des nouvelles, valables uniquement pour les objets de la sous-classe. On peut aussi redéfinir le méthodes de la classe de base, qui font alors autre chose si on les applique à la sous-classe.

```
class animal(object):
    def __init__(self, age): #passage "par dictionnnaire"
        self.age = age

    def anniversaire(self):
        self.age += 1

## Classe qui hérite de animal
class humain(animal):
    # Toutes les fonctions dans la definition de animal restent valables pour cette cla
    # On peut changer la definition d'une fonction en la reecrivant ici, et en mettre c

    def __init__(self, age, nom): #reecrite pour rajouter une donnée a la représentatio
        animal.__init__(self, age)
        self.nom = nom

    def parle(self): #nouvelle fonction specifique a humain
        print("bonjour")

bob = humain(23,"Bob")

bob.parle()
```

```
bob.anniversaire()
print(bob.age)
```