1 Introduction

Nous avons déjà évoqué le fait que **la programmation orientée objet** (POO) est un **paradigme de programmation** (cf cours P4-IV-Paradigmes).
La POO est au centre de la manière dont Python fonctionne même si on n'est pas obligé de mettre en œuvre ce paradigme dans un programme. De nombreuses classes et objets sont fournis par la librairie standard.
Par exemple, en manipulant les listes-Python, vous avez certainement remarqué qu'il y a deux syntaxes pour appeler des fonctions :

```
tableau = [1, 3, 5, 8]  # création d'une objet de type list
taille = len(tableau)   # 1. appel d'une fonction
tableau.append(11)      # 2. appel d'une méthode
```

- Le calcul de la taille du tableau se fait par l'appel à la fonction `len()` avec une syntaxe identique aux fonctions que vous avez l'habitude d'écrire.
- L'ajout d'un élément dans le tableau est un peu différent car la fonction `append` semble provenir du tableau lui même : dans ce cas, on ne parle pas de fonction mais de **méthode** associée à l'**objet** tableau.

> Un **objet** est une structure de donnée qui intègre des variables, que l'on nomme **attributs** ou **propriétés**, et des fonctions, que l'on nomme **méthodes**.

Nous allons voir l'intérêt de cette approche, omniprésente dans Python, en particulier lorsqu'on développe des interfaces graphiques, mais pas seulement.

2 Petit historique

La programmation en tant que telle est une matière relativement récente. Étonnamment la programmation orientée objet remonte aussi loin que les années 1960. *Simula* est considéré comme le premier langage de programmation orienté objet.

Les années 1970 voient les principes de la programmation par objet se développer et prendre forme au travers notamment du langage *Smalltalk*.

À partir des années 1980, commence l'effervescence des langages à objets : *Objective C* (début des années 1980, utilisé sur les plateformes Mac et iOS), *C++* (C with classes) en 1983 sont les plus célèbres.

Les années 1990 voient l'âge d'or de l'extension de la programmation par objet dans les différents secteurs du développement logiciel, notamment grâce à l'émergence des systèmes d'exploitation fondés sur une interface graphique (MacOS, Linux, Windows) qui font appel abondamment aux principes de la POO.

3 Un exemple pour mieux comprendre...

Supposons qu'on doive créer et « manipuler » un personnage dans un jeu informatique.
Aucune structure de donnée liée directement au langage n'existe pour décrire ce type de donnée (comme un entier ou un tuple ...).
La POO nous permet d'élaborer notre propre structure décrivant un personnage.

Un personnage pourrait par exemple être représenté par :
- un nom
- un niveau de vie
- une position (sur un plateau de jeu)

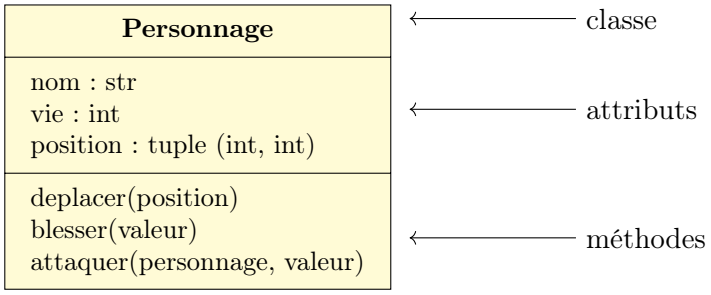
Ce sont les **attributs** d'un objet de type personnage.

Et il pourrait avoir les « capacités » suivantes :
- se déplacer (changer de position)
- perdre de la vie
- attaquer un autre personnage

Ce sont les **méthodes** associées à un objet de type personnage.

Les attributs et les méthodes sont regroupées dans une unique structure qu'on appelle **classe**.
Tous les personnages du jeu sont des représentants de cette classe : ils sont construits sur le modèle de la classe. Chaque personnage est un **objet** qui est une **instance** de la **classe** Personnage.

On peut résumer ceci sur le schéma suivant :



4 Création d'une classe

Voici comment on construit la classe précédente en Python :

```
class Personnage:

    def __init__(self, pseudo, force, depart):
        """ Crée un nouvelle instance de la classe Personnage avec les
        ↪  caractéristiques suivantes :
        - pseudo : str (nom du personnage)
        - force : int (nb de points de vie)
        - depart : (int, int) (coordonnées initiales)
        """
        self.nom = pseudo
        self.vie = force
        self.position = depart

    def deplacer(self, lieu):
        """ Définit une nouvelle position pour le personnage .
        - lieu : de type position : tuple d'entiers (des coordonnées)
        """
        self.position = lieu
```

```
    def blesser(self, valeur):
        """ Diminue les points de vie du personnage.
        - valeur : int
        """
        self.vie = self.vie - valeur

    def attaquer(self, autre, valeur):
        """ Blesse un autre personnage d'une certaine valeur.
        - autre : de type personnage
        - valeur : int
        """
        autre.blesser(valeur)
```

Et voici un exemple d'utilisation :

```
>>> p1 = Personnage('Alan', 12, (2, 5)) # instanciation d'un 1er objet personnage
>>> p2 = Personnage('Ada', 10, (3, 1))  # instanciation d'un 2eme objet personnage
>>> p1.position                         # accès à l'attribut 'position' de p1
(2, 5)
>>> p1.deplacer((3, 1))                 # application de la méthode 'deplacer' à p1
>>> p1.position                         # accès à l'attribut 'position' de p1
(3, 1)
>>> p1.position == p2.position          # test sur les attributs 'position'
True
>>> p2.vie                              # accès à l'attribut 'vie' de p2
10
>>> p1.attaquer(p2, 4)                  # application de la méthode 'attaquer' à p1
>>> p2.vie                              # accès à l'attribut 'vie' de p2
6
```

Explications :

> Une classe est créée avec le mot clef `class`.
> Elle comporte toujours une méthode appelée `__init__(self, ...)` qui est automatiquement appelée lors de l'instanciation d'un objet de la classe.

Remarque : le bon usage recommande qu'on écrive une classe avec une majuscule.

Le paramètre `self`, obligatoire, fait référence à l'objet lui-même, celui qui est cours d'instanciation. Les autres paramètres, facultatifs, permettent de transmettre des données utiles à l'instanciation de l'objet (ici, des caractéristiques initiales du personnage).

> Chaque attribut de l'objet est accessible avec la « notation pointée », sous le format « `self.nom_attribut` ».

Ensuite, la définition de la classe décrit toutes les méthodes qu'on pourra appliquer aux instance de la classe.

> Les méthodes sont écrites comme des fonctions habituelles, avec le mot clef `def`, et contiennent toujours (au minimum) le paramètre `self` faisant référence à l'objet auquel elles seront liées.

Pour instancier un objet de la classe ainsi définie, il suffit d'appeler le nom de la classe (comme une fonction habituelle) en passant comme paramètres ceux attendus par la méthode `__init__` (inutile de passer le paramètre `self`, ce processus se fait automatiquement).

Ensuite pour accéder à un attribut de l'objet ou lui appliquer une de ses méthodes, on utilise la notation pointée au format « `objet.nom_attribut` » ou « `objet.nom_methode(parametres)` ».

Remarque : Comme pour la méthode `__init__`, il ne faut pas écrire le paramètre `self` lors de l'invocation d'une méthode.

5 Pour aller plus loin

L'approche de la programmation orientée objet reste limitée à ces concepts en Terminale, mais vous pouvez savoir que d'autres caractéristiques importantes sont liées à ce paradigme.
Il est en particulier possible de « protéger » l'accès aux attributs d'un objet du point de vue de l'extérieur du programme. Ce concept d'**encapsulation** rend plus robuste la construction de programme.
Par ailleurs, il est possible de créer des classes dérivées de classes plus générales en **héritant** de leurs attributs et méthodes avec la possibilités de les particulariser (ex : un Chien pourrait être une classe qui hérite de la classe Animal).
Enfin le **polymorphisme** autorise une même méthode à s'appliquer à différents types d'objet de manière transparente.