Chapitre : Progammation orienté Object

- *Author* : **Ibrahima SY**
- *Email* : syibrahima31@gmail.com
- *Github* : Cliquez
- *Linkdin* : Cliquez
- *School* : UCAD
- *Spécialité* : M1 **MSI**

Dénition

La programmation orientée objet (on dira POO) est un paradigme de programmation informatique. Il consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique,.

Philosophie de la POO

- Dans la programmation orientée objet (POO), toutes les variables sont des objets associés à une classe
- Une classe est un type

  > qui se veut plus complexe que juste un nombre ou un caractère

- Un objet est une collection
  - de données membres (ou attributs)
  - de fonctions membres manipulant les données membres (ou méthodes)
- On peut créer autant d'objets de classe (i.e. de variables du type décrit par la classe) que l'on veut
- Un objet est aussi appelé une instance de classe
- En Python, tout est objet

L’objet et la classe

Un objet est un élément d’une classe (en quelque sorte un « type personnalisé » ; il faut entendre type au même titre qu’un nombre, une chaîne de caractères, …).

Définir une classe revient à définir une nouvelle structure de données, qui s’ajoute à celles définies par le langage.

Définition de classe

- Une classe possède un **nom** et des **membres**, c'est à dire des **attributs** (ou champs) et des **méthodes**.
- Chaque membre d'un objet objet est accessible via l'expression : `objet.membre`.
- Les attributs (ou champs) : ils sont à l'objet ce que les variables sont à un programme.

  > les attributs sont donc typés (exemple int, float, bool, str, ... ou n'importe quelle autre classe !)

- Les méthodes sont des `procédures` ou `fonctions` destinées à traiter les données. Elles servent d'interface entre les données et le programme.

  > les méthodes acceptent donc des arguments et peuvent renvoyer des valeurs

- Pour définir une classe on utilise le mot clé class

```
class Personne :
    pass
```

- On peut ensuite ajouter des données membres et des fonctions membres

```
class Personne :
    def initialiser(self, nom, prenom):
        self.prenom = prenom
        self.nom = nom

    def afficher(self):
        print(f"{self.prenom} {self.nom}")
```

`afficher` ou `inittialiser` est une fonction que l'on peut appeler sur une instance de la classe `Personne`

Création d'un objet

- On peut créer une instance (ou objet) de classe en utilisant le nom de la classe suivie de parenthèses

```
pers = Personne()
```

- On peut ensuite accéder aux `attributs` et fonctions en utilisant l'opérateur `.` (point)

```
pers.initialiser("Abdou", "Geuye")
```

Référence à l'instance

- Vous avez noter la présence du paramètre self dans la méthode

```
def initialiser(self, nom, prenom)
```

- Pourtant il n'est pas passé en paramètre lors de l'appe
- Lorsqu'on appele une méthode sur une instance de classe, cette instance est automatiquement ajoutée en tant que premier paramètre
  - L'appel

    ```
    pers.initialiser("Abdou", "Geuye")
    ```

  - est traduit automatiquement en

    ```
    Personne.initialiser(pers, "Abdou", "Geuye")
    ```

Principe d'encapsulation

L'encapsulation consiste à ne fournir aux utilisateurs d'une classe qu'un accès limité à la classe. En général, on masque tout ce qui est attribut pour ne laisser la possibilité d'utiliser que quelques méthodes.

1. En masquant les attributs à l'utilisateur, on interdit à ce dernier d'effectuer directement des modifications de valeurs ; on peut ainsi contrôler l'intégrité et la validité des valeurs.
2. On offre à l'utilisateur une interface : la signature et le rôle d'un ensemble de méthodes ; cette interface détermine les services possibles d'une classe en rendant ces services indépendants de leur implémentation.

Mise en oeuvre de l' encapsulation

Malheureusement de base Python n'offre pas vraiment les moyens de mettre en oeuvre l'encapsulation contrairement à des langages comme Java ou C++ où le concept d'attributs et méthodes privés existent.

En Python, on peut utiliser des conventions de nommage pour inciter l'utilisateur à respecter l'encapsulation.

```
class Personnage:
    def __init__(self, nom, pv, pv_max):
        self._pv = pv    # juste une convention pour dire qu'il faut pas toucher
        self.__pv_max = pv_max # c'est la maniére la plus rigoureuse mais reste accéssible
        self.nom = nom
```

Assesseurs et Mutateurs

Ce sont des méthodes qui vont permettre à l'utilisateur d'obtenir la valeur des attributs et de les modifier (lorsque c'est pertinent), sans pour autant accéder directement aux variables d'instance, qui, on le répète, doivent autant que possible rester cachées.

```
class Personnage:
    def __init__(self, nom, pv, pv_max):
        self._pv = pv    # juste une convention pour dire qu'il faut pas toucher
        self.__pv_max = pv_max # c'est la maniére la plus rigoureuse mais reste accéssible
        self.nom = nom

    def getter_pv_max(self):
        return self.__pv_max

    def setter_pv_max(self, new_value):
        self.__pv_max = new_value
```

L'héritage

L'héritage est une relation asymétrique entre deux classes : l'une est la classe mère (aussi nommée classe parente, superclasse, classe de base), l'autre la classe-fille. L'héritage permet une économie d'écriture par la réutilisation automatique, lors de la définition de la classe-fille, de tous les membres et autres éléments définis dans la classe mère. Ainsi, les objets de la classe-fille héritent de toutes les propriétés de leur classe mère.

Lorsqu'une classe-fille possède une unique classe mère on parle d'héritage simple ; dans le cas de plusieurs il s'agit d'héritage multiple. Cette dernière forme offrant son lot de difficultés, beaucoup de langages n'admettent que l'héritage simple. C'est notamment le cas de Smalltalk pourtant considéré comme l'archétype des langages orientés objet par de nombreux auteurs. C++ mais aussi Python, Perl, Eiffel proposent l'héritage multiple. Java et C# simulent l'héritage multiple par diverses techniques propres.

```python
class Personne :
    def __init__(self, nom, prenom):
        self.nom = nom
        self.prenom = prenom

    def afficher(self):
        return f"{self.prenom} {self.nom}"


class Client(Personne):
    pass
```

Polymorphisme

On peut toutefois vouloir qu'une méthode d'une classe fille n'ait pas le même comportement que son équivalent dans la classe mère. On redéfinit alors simplement la méthode dans la classe fille. On parle de `polymorphisme` et cette technique relève de la `surcharge de méthode`, ou en anglais `overriding`.

```
class Client(Personne):
    def __init__(self, nom, prenom, num_compte):
        super().__init__(self, nom, prenom)
        self.num = num_compte

    def afficher(self):
        return f" prenom = {self.prenom}, nom={self.nom} numero = {self.num_compte}"
```