# Programmation Orientée Objet (POO)

En programmation, une « classe » est une structure de données personnalisée, créée par le programmeur. C'est un « modèle», qui permet d'associer des données (« états ») avec des comportements (« méthodes ») manipulant ces données, le but étant de disposer à la fin d'objets qui gèrent « tous seuls » leur logique interne, l'utilisateur se contentant alors de déclencher les différentes actions. Par exemple :

| *À modéliser* | *État(s)* | *Action(s)* |
|---|---|---|
| Compte en banque | | |
| Chronomètre | | |

## I) Définitions

- ***Classe & interface :*** Une fois qu'on a décidé quels sont les états et les actions possibles pour tel ou tel type d'objet, on a définit ce qu'on appelle son « interface » : c'est la liste de tout ce qui est faisable avec ce type d'objet.

- ***Instance/objet :*** La classe étant un modèle, ce n'est pas l'objet concret avec lequel on travaille ensuite.

| ***Classe*** | ***Instances/objets*** |
|---|---|
| ***Compte***<br>États :<br>- somme<br>- propriétaire<br>Actions :<br>- créer()<br>- ajouter(v)<br>- retire(v) | Compte1 :<br>somme : 100<br>propriétaire :<br>Alphonse<br><br>Compte2 :<br>somme : 1765<br>propriétaire :<br>Alain<br><br>Compte3 :<br>somme : 9<br>propriétaire :<br>Alfred |

- ***Propriétés/attributs :***

- ***Méthodes :***

# II) Implantation d'une classe en python

## II.1) Le « contexte » d'une instance : « self »

Pour implanter tout cela, on se retrouve confronté à une difficulté : on doit coder la classe, c'est-à-dire le modèle, alors que les données concrètes n'existent pas encore, et de la même manière, on doit avoir un moyen pour qu'un objet concret puisse utiliser les méthodes (*rappel : méthode = fonction*) avec ses propres états/attributs.

Pour cela, la classe doit être implantée avec une référence au « contexte » de l'instance en jeu (ses données à elle). En python, le contexte est un argument noté *self* qui est passé systématiquement en premier argument de chaque méthode.

## II.2) Coder une classe

Sur l'exemple dune classe servant à modéliser un maillon d'une liste chaînée :

```
class Maillon:

    def __init__(self, valeur, suivant):
        self.valeur = valeur
        self.suivant = suivant


    def tete(self):
        return self.valeur


    def queue(self):
        return self.suivant


    def somme(self):
        if self.queue() is None:
            return self.tete()
        return self.tete() + self.queue().somme()
```

## II.3) Utiliser une classe : instancier des objets et les utiliser

### Créer un objet/une sintance :

```
obj1 = Maillon(5)
```
Crée un objet de type Maillon avec la valeur 5, sans suivant (None)

```
obj2 = Maillon(13, obj1)
```
On a maintenant une liste chaînée 13 → 5 → None

***Accéder aux propriétés d'une instance :***

En python, tous les attributs sont toujours (*ou presque… mais hors programme*) accessibles depuis « n'importe où » :

Pour la classe Maillon, si on veut « accéder au *suivant* du maillon en cours » :

| *Depuis le code de la classe :* | *Quand on utilise l'instance de Maillon* `obj2` *:* |
|---|---|

***Utiliser une méthode d'une instance :***

Toujours sur l'exemple de la classe Maillon, si on veut « demander » à l'instance en cours de « calculer la somme de sa valeur et tous ses suivants » :

| *Depuis le code de la classe :* | *Quand on utilise l'instance de Maillon* `obj2` *:* |
|---|---|

# III) Quelques intérêts des classes et de la POO

## III.1) Interface d'une classe

L'interface est généralement décrite dans le docstring de la classe et ne regroupe pas forcément tous les attributs et méthodes existante : certains peuvent être considéré comme réservés à un usage interne au cours de la classe.

## III.2) Encapsulation

***Idée générale :***

Lorsqu'une classe est bien construite, l'utilisateur n'a plus qu'à interroger les instances via leurs méthodes, sans avoir à se préoccuper de la façon dont la classe fonctionne en sous-main. La classe et ses instances se comportent alors comme de petits modules indépendants dont on n'a plus besoin de se préoccuper de savoir comment le code fonctionne. On parle d'***encapsulation***.

***Respecter l'encapsulation :***

Par exemple, pour un objet *compte* de la classe CompteEnBanque et ayant une propriété *montant*, duquel on voudrait retirer de l'argent, on pourrait faire :



*L'encapsulation, concrètement : python vs autres langages…*

Certains langages ont mis en place des outils qui permettent de forcer le respect de l'encapsulation. Par exemple, en Java, les attributs et les méthodes peuvent être déclarés comme étant « privés ». Dans ce cas, il est impossible à l'utilisateur d'y accéder depuis l'extérieur.

En python, rien de tout cela… d'un point de vue technique, l'encapsulation n'existe pas, en python ! Tout est toujours accessible depuis n'importe où, moyennant quelques acrobaties… MAIS, le principe d'encapsulation existe toujours, c'est un accord tacite entre l'utilisateur et le programmeur de la classe :

*« Je te propose une classe, tu peux l'utiliser, mais si tu commences à faire n'importe quoi avec, ne viens pas te plaindre que ça ne marche plus comme il faudrait ! »*

### *III.3) Diminution de la charge mentale pour l'utilisateur*

Le principal intérêt d'utiliser des classes et qu'une fois une classe implantée, si elle a été construite intelligemment, ***et que le principe d'encapsulation est respecté***, l'utilisateur peut s'en servir en oubliant complètement la façon dont elle fonctionne et en se limitant à la connaissance de son interface.

L'utilisation des classes dans la programmation orientée objet est donc un premier pas vers des implantations dites « modulaires », où on découpe un problème complexe en petits problèmes ***indépendants*** plus simples que l'on articule ensuite entre eux.

### *III.4) Évolutivité du code*

***Si le principe d'encapsulation est respecté***, le programmeur peut faire évoluer le code de la classe sans modifier son interface : on peut ainsi améliorer le comportement/les performances de la classe sans rien changer aux codes qui dépendent de cette classe.

*Exemple :*

*Une première version d'une classe « chronomètre » stocke le temps avec 3 données : heures, minutes et secondes. Le programmeur peut décider de changer la représentation interne en ne stockant plus que des secondes. Si l'interface a été bien conçue, elle ne change pas, et si l'encapsulation a été respectée du côté de l'utilisateur, il ne verra pas de différence et tout le code qu'il aura écrit en utilisant la classe Chronomètre marchera toujours exactement de la même façon.*