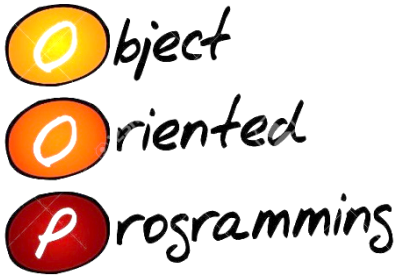
1 Introduction



Les objets: un moyen de séparer la conception de l'utilisation

La programmation orientée objet repose, comme son nom l'indique, sur le concept d'objet. Un objet dans la vie courante, vous connaissez, mais en informatique, qu'est ce que c'est ? Une variable ? Une fonction ? Ni l'un ni l'autre, c'est un nouveau concept.

Imaginez un objet très complexe (par exemple un moteur de voiture) : il est évident qu'en regardant cet objet, on est frappé par sa complexité.

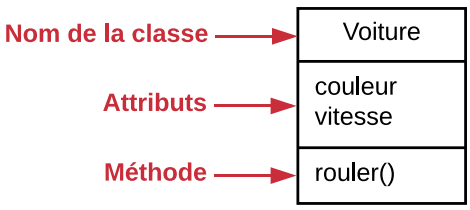
Imaginez que l'on enferme cet objet dans une caisse et que l'utilisateur de l'objet n'ait pas besoin d'en connaître son principe de fonctionnement interne pour pouvoir l'utiliser. L'utilisateur a, à sa disposition, des boutons, des manettes et des écrans de contrôle pour faire fonctionner l'objet, ce qui rend son utilisation relativement simple. C'est ce qu'on fait quand on conduit !

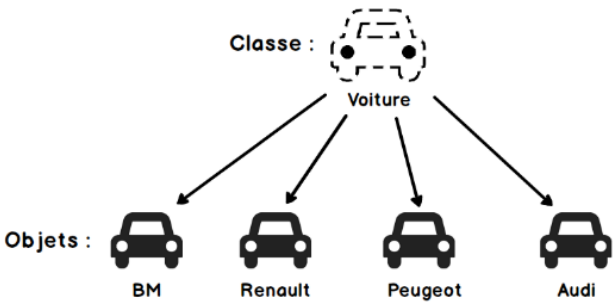
La mise au point de l'objet (par des ingénieurs) a été très complexe, en revanche son utilisation est relativement simple. Programmer de manière orientée objet, c'est un peu reprendre cette idée : utiliser des objets sans se soucier de leur complexité interne.

Pour utiliser ces objets, nous n'avons pas à notre disposition des boutons, des manettes ou encore des écrans de contrôle, mais des **attributs** *(variable d'objet)* et des **méthodes** *(fonction d'objet)*.

Il existe plusieurs approches, plusieurs façon de programmer, cela s'appelle un **paradigme**. Le paradigme avec lequel nous avons programmé jusqu'à présent s'intitule **paradigme impératif** (programmes décomposés en fonctions et lignes de codes réalisant des tâches simples). **La programmation orientée objets est un paradigme [1] de programmation comme la programmation fonctionnelle, évènementielle, impérative, ...**

Le développement de logiciels parfois complexes par une équipe d'informaticiens nécessite d'employer la programmation orientée objet. Cela permet de programmer par modules (classes) indépendants, et facilite la maintenance.





En programmation orientée objet, on fabrique de nouveaux types de données appelées **classes (class en anglais)**, correspondant aux besoins du programme.

A partir d'une classe, des **objets** (**instances de classe**) peuvent être créés (instanciés), ils héritent des actions possibles et caractéristiques de cette classe.

- Les caractéristiques des objets s'appellent des **attributs**, se sont des variables de classe.
- Les actions possibles sur ou à partir de ces objets s'appellent des **méthodes**, se sont des fonctions de classe.

Les différents objets créés à partir d'une classe n'interfèrent pas les uns avec les autres, et peuvent évolués chacun indépendament. Chaque objet enferme/encapsule ses propres attributs (variables).

---

[1] ***Paradigme*** *: représentation du monde ; manière de voir les choses ; modèle cohérent de pensée, de vision du monde qui repose sur une base définie, sur un système de valeurs.*
En informatique : Un paradigme de programmation est une façon d'approcher la programmation informatique et de traiter les solutions aux problèmes et leur formulation dans un langage de programmation approprié

1.1 Retour sur l'histoire :

Introduite dans Simula en 1962, vraiment passée dans l'usage courant en 1972 avec SmallTalk, grâce à Alan Kay (prix Turing en 2003)

Généralisée à de nombreux langages dans les années 1970 et 1980 : C++ ajoute la programmation objet au langage le plus populaire de l'époque : C.

Depuis les années 90, tous les langages "impératifs" proposent de la POO : python, java, javascript, ruby, C# (se prononce C SHARP), OCaml, ...

La programmation objet est enseignée dans tous les parcours informatiques. La grande majorité des logiciels sont écrit en POO.

1.2 Programmation orientée objet et Python :

Les langages sont très rarement « purs », ils intègrent généralement plusieurs paradigmes de programmation. Python est un langage permettant de programmer avec des paradigmes de programmation tels qu'impératif, fonctionnel, et orienté objet. D'ailleurs ***en python, tout est objet !***

Remarque :

Vous avez déjà utilisé des classes et objets de classes sans le savoir en python, par exemple en manipulant des list, tuple, dictionnaires, entiers, float, ...

Exemple :

```
>>> uneListe = [7,5,1,3]
>>> type(uneListe)
<class 'list'>
>>> isinstance(uneListe, list)
True
```

On remarque que *uneListe* est donc une instance de la classe '*list*' python, et qu'il y a deux façons de le vérifier : *type* et *isinstance*

La commande '*sort*' est une des méthodes de la classe '*list'* python comme '*append*', '*pop*', '*len*', ...

```
>>> uneListe.sort()
>>> uneListe
[1,3,5,7]
```

Pour obtenir les différentes méthodes d'une classe :

```
>>> dir(uneListe)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

De même pour illustrer que tout est objet :

```
>>> competiteur = {"nom" :  "Duchmol", "prenom" : "Robert", "age" : 42}
>>> type(competiteur)
<class 'dict'>

>>> nb = 42
>>> type(nb)
<class 'int'>
```

2 Création d'une classe

La création d'une classe en python commence toujours par le mot class.

Par convention **le nom d'une classe commence toujours par une majuscule**. Ensuite toutes les instructions de la classe seront indentées :

```
class LeNomDeMaClasse:
      # instruction de la classe
      # instruction de la classe
      ...
# La définition de la classe est terminée.
```

2.1 La méthode spéciale 'Constructeur'

Le constructeur est une méthode (fonction) spéciale de la classe qui commence toujours par :

```
class LeNomDeMaClasse:
      def __init__(self, ...):
            ...
```

A chaque création d'une instance (objet) de la classe, cette méthode 'constructeur' va initialiser les variables de l'instance.

Imaginons-nous entrain de programmer un jeu vidéo, et créons une classe *'combattant'* :

```
class Combattant:
    '''Classe définissant un combattant par :
          - Son nombre de points de vie
          - Son nombre de points d'attaque
          - Son état : vivant ou non'''

    def __init__(self, vie, attaque):
        self.vie = vie
        self.attaque = attaque
        self.vivant = True
```

Maintenant utilisons cette Classe 'combattant' en créant des instances (objets) :

```
>>> guerrier = Combatant(10,2) # création d'une instance appelée guerrier
>>> type(guerrier)
<class 'Combattant'>
>>> magicien = Combattant(5, 4)  # création d'une autre instance magicien
>>> guerrier.vie                 # chaque instance a ses valeurs d'attributs
10
>>> guerrier.vivant
True
>>> magicien.vie                 # différent de guerrier.vie
5
```

Travail 1 - Tester le code ci-dessus

Commentaires sur le code :

- **La méthode __init__ est appelée le constructeur de la classe**, c'est une méthode spéciale qui est appelée implicitement lorsque l'on créé une instance de la classe. Elle initialise les attributs (variables) de l'instance.
- **La variable self est toujours le premier paramètre d'une méthode. Il désigne l'objet auquel s'appliquera la méthode.**

  Attention ! Python n'impose pas le nom self. . . ce n'est qu'une convention. Cependant, si vous écrivez autre chose que self, personne ne vous comprendra.
- self.vie, self.attaque, et self.vivant sont des attributs

On s'aperçoit que l'on peut accéder aux attributs (`guerrier.vie, guerrier.vivant, ...`) de la classe, ce qui n'est pas une habitude à prendre. En effet rien n'empêcherait de faire des choses absurdes comme :

```
>>> guerrier.vivant
False
>>> guerrier.vivant = True
>>> guerrier.attaque = 10000000000000000000000000
>>> guerrier.vie = "BONJOUR !"
```

On ne va donc généralement pas utiliser la méthode précédente *nom_objet.nom_attribut* permettant d'accéder aux valeurs des attributs car on ne veut pas forcement que l'utilisateur ait accès à la représentation interne des classes. Pour utiliser ou modifier les attributs, on utilisera de préférence des méthodes dédiées dont le rôle est de **faire l'interface entre l'utilisateur et l'objet (API : *A*pplication *P*rogramming *I*nterface)**.

Les attributs sont alors en quelque sorte **encapsulés** dans l'objet, c'est à dire non accessibles directement par le programmeur qui a instancié un objet de cette classe.

Encapsulation

L'encapsulation désigne le principe de regrouper des données brutes (attributs) avec un ensemble de routines (méthodes) permettant de les lire ou de les manipuler.

But de l'encapsulation : cacher la représentation interne des classes.

- *Pour simplifier la vie du programmeur qui les utilise*
- *Pour masquer leur complexité (diviser pour régner)*
- *Pour permettre de modifier celle-ci sans changer le reste du programme*
- *La liste des méthodes devient une sorte de mode d'emploi de la classe (API)*

2.2 Les méthodes publiques, privées et spéciales

L'ensemble des méthodes accessibles à un utilisateur définit l'interface de la classe.

Méthodes spéciales :

Il s'agit de méthodes qui peuvent se lancer seules : l'interpréteur Python réagit à certains événements et va automatiquement y faire appel lorsque l'événement déclencheur survient. Par exemple à la création d'une instance d'une classe la méthode spéciale __init__ va être exécutée et construire la classe.

Les méthodes commençant et finissant par un double tiret bas '__' sont des méthodes spéciales, elles existent par défaut à l'élaboration d'une classe, par exemple :

__init__ : Constructeur de classe
__doc__ : documentation de la classe (docstring)
__name__ : nom de la classe
__dict__ : dictionnaire avec comme clés
les noms des attributs et leurs valeurs
__del__ : destructeur de la classe
__str__ : str(obj), print(obj)
__repr__ : dans la console >>> obj
__add__ : +
*__mul__ : **

__sub__ : -
__eq__ : ==
__ne__ : !=
__lt__ : <
__le__ : <=
__gt__ : >
__ge__ : >=
__getitem__ : obj[i]
__iter__ : for v in obj

Lorsqu'on utilise une fonction courante de Python sur un objet comme une liste par exemple, il appelle une méthode spéciale de cet objet :

```
>>> 1 + 1                    # appelle la méthode spéciale '__add__' des 'int'
>>> ma_liste = [1, 2, 3]     # appelle la méthode spéciale '__init__' des 'list'
>>> ma_liste[1]              # appelle la méthode spéciale '__getitem__' des 'list'
2
>>> ma_liste                 # appelle la méthode spéciale '__repr__' des 'list'
[1, 2, 3]
```

Il est possible de modifier le comportement de ces méthodes spéciales, cela s'appelle surcharger cette méthode spéciale. Nous verrons cela en activité.

Méthodes publiques et privées :

Il s'agit de méthodes qui sont appelées par '*nomInstance.nomMethode*'

- **Les méthodes publiques** peuvent être appelées de n'importe où dans le code (depuis l'intérieur, ou l'extérieur de la classe).
- **Les méthodes privées** tout comme les attributs privés ne peuvent être appelés/utilisés que depuis l'intérieur de la classe. Leurs noms commencent par un tiret bas '_' pour les distinguer.

Reprenons l'exemple de notre jeu vidéo et ajoutons quelques méthodes publiques :

```
class Combattant:
    '''Classe définissant un combattant par :
          -  Son nombre de points de vie
          -  Son nombre de points d'attaque
          -  Son état : vivant ou non'''

    def __init__(self, vie, attaque):
        self.vie = vie
        self.attaque = attaque
        self.vivant = True

    def _perdreVie(self, points):
        self.vie = self.vie - points
        if self.vie <= 0:
            self.vivant = False
            self.vie = 0

    def attaquer(self, autre):
        autre._perdreVie(self.attaque)
```

```
>>> guerrier = Combatant(10,2)   # création d'une instance guerrier
>>> magicien = Combattant(5, 4)   # création d'une instance magicien
>>> magicien.attaquer(guerrier)   # utilisation d'une méthode
>>> guerrier.vie
6
>>> magicien.attaquer(guerrier)
>>> guerrier.vie
2
>>> magicien.attaquer(guerrier)
>>> guerrier.vie
0
```

Remarque : La méthode attaquer prend en paramètre un objet autre que lui-même, et exécute une méthode de cet objet (autre._perdreVie(...)). Ce n'est pas le seul usage envisageable de la méthode _perdreVie, plus tard dans le développement du jeu, on peut imaginer qu'un Combattant perde de la vie en marchant dans la lave ou sur un piège. . .

Travail 2 - Tester le code ci-dessus avec les deux méthodes _perdreVie et attaquer.

Travail 3 - Créer une méthode publique nommée estVivant qui renvoie True si le combattant est vivant et False sinon

Travail 4 - Créer une méthode publique nommée seBlesse qui fait perdre un nombre de points de vie passé en argument, et utilise la méthode privée _perdreVie

Travail 5 - Créer une méthode publique nommée seSoigne qui fait récupérer au combattant un nombre de points de vie passé en argument, et utilise la méthode privée _perdreVie

Travail 6 - Surcharger la méthode spéciale __str__ pour un affichage propre des caractéristiques du combattant.

Travail 7 - Créer une méthode publique nommée combat qui prend en argument un adversaire et les fait se combattre (selon vos règles) jusqu'à ce que l'un des deux n'ait plus de points de vie et renvoie le vainqueur.