

# Chapitre 1 : Programmation orientée objet

## Notions et contenus :

- vocabulaire de la programmation objet : classes, attributs, méthodes, objets.
- la notion d'encapsulation,
- la notion d'attribut (ou de méthode) public ou privé.

## Capacités exigibles :

- Écrire la définition d'une classe en Python,
- Créer le constructeur d'une classe,
- Créer une instance d'une classe,
- Accéder et modifier les attributs d'un objet d'une classe,
- Créer de nouvelles méthodes à une classe,
- Documenter une classe.

## Table des matières

Table des matières .....	1
I. Introduction à la POO .....	2
1) Un autre paradigme de programmation.....	2
2) Un petit exemple, une histoire de chiens .....	3
II. La POO en Python .....	4
1) Définitions.....	4
2) Une méthode particulière : le constructeur .....	4
3) Afficher toutes les méthodes et attributs d'une classe ou d'un objet .....	4
4) Un exemple pratique en python : .....	5
III. Améliorer la compréhension de la classe .....	6
1) D'autres méthodes particulières : <code>__repr__</code> ou <code>__str__</code> .....	6
2) Documenter .....	6
IV. Attributs publics ou privés .....	7
1) Le principe de l'encapsulation .....	7
2) Les accesseurs.....	8
3) Les mutateurs .....	8

## I. Introduction à la POO

### 1) Un autre paradigme de programmation

Jusqu'ici, les programmes que vous avez créés l'ont été en :  
c'est une série d'instructions (assignations, boucles, conditions, opérations) qui modifient l'état d'un programme jusqu'à obtenir la solution. On donne une recette de cuisine qui est suivie ligne à ligne par l'interpréteur.

Exemple :

```
somme = 0

for i in range(14):
    somme = somme + i

print (somme)
```

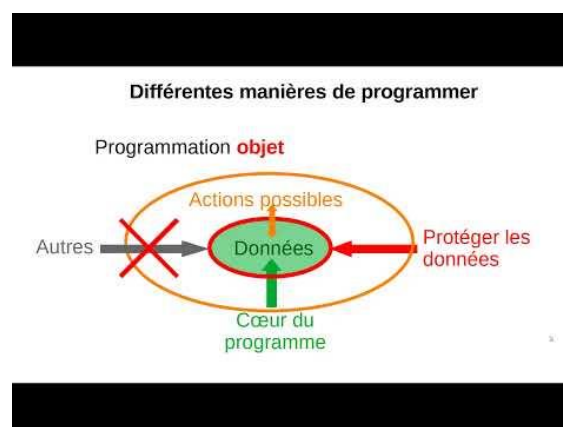
Puis vous avez utilisé la : le programme est décomposé en plusieurs fonctions, chacune réalisant des tâches simples.

Exemple :

```
def calculer_regle(valeur):
    autorise = valeur/2 + 7
    return autorise

print(calculer_regle(20))
```

La philosophie de la POO en vidéo :



À partir des années 1960, un autre type de programmation a été rendu nécessaire par le fait que, lorsque plusieurs programmeurs travaillent simultanément sur un projet, il faut éviter les conflits entre les fonctions : la est née.

Le premier langage de programmation initiant une forme de programmation orientée objet fut Simula, créé à partir de 1962. Ce langage servait à faciliter la programmation de logiciels de simulation.

Le premier langage de programmation réellement fondé sur la programmation orientée objet fut Smalltalk 71, créé au début des années 70.

La programmation orientée objet est un paradigme de programmation, c'est-à-dire une autre manière de voir les notions en programmation. En programmation procédurale, ce sont les fonctions qui sont au cœur du programme ; elles s'appliquent à modifier des données.

En programmation orientée objet, ce sont les données qui sont au cœur du programme : celles-ci vont désormais être protégées et c'est le développeur qui décide comment elles seront créées, accessibles, modifiées, ...

## 2) Un petit exemple, une histoire de chiens



Si l'on considère tous les chiens dans un chenil, alors on peut essayer de tous les caractériser en mettant en avant des caractéristiques particulières : , ... Chacun de ces chiens possède ces caractéristiques, mais elles prennent une valeur qui lui sont personnelles. De même, tous les chiens peuvent être soumis à des actions :

...

Ainsi, on peut modéliser cette situation en Programmation Orientée Objet :

La classe sera la classe , chaque représentant individuel sera appelé une de la classe, ou un objet. Les variables caractéristiques de la classe seront représentées par des , et les actions possibles par des de la classe.

## II. La POO en Python

### 1) Définitions

Une `classe` est une structure de données abstraite définissant une catégorie générique d'objets. C'est un moule grâce auquel nous pourrons créer des objets. Le nom d'une classe débute toujours par une majuscule !


Chaque élément d'une classe donnée est appelé `attribut` de la classe, ou `méthode` de la classe.

Les caractéristiques de cet objet sont des `attributs` associées à l'objet et qui sont appelées des `attributs de classe`. Ces `attributs de classe` peuvent être liés soit :

- à la classe (on dira alors des `attributs de classe`) : ils sont identiques pour chaque instance de la classe et n'ont pas vocation à être changé
- ou à l'instance (ce seront des `attributs d'instance`) : ils ne prendront pas forcément la même valeur d'une instance à l'autre

Les fonctionnalités de cet objet sont des `méthodes` associées à l'objet et qui sont appelées des `méthodes d'instance`. Les méthodes ont toujours pour premier paramètre le mot `self`.

Lorsqu'on crée un objet d'une certaine classe, on dit qu'on `instancie` un élément de la classe. Pour cela, il suffit de réaliser la commande suivante :

 <b>NomClasse</b>
+ attribut_1 : type + attribut_2 : type
+ methode_1(type): type + methode_2(type): type

### 2) Une méthode particulière : le constructeur

Le `__init__` d'une classe est une `méthode` qui a un nom imposé : `__init__` avec deux underscores de chaque côté....

Le premier paramètre est invariablement `self`. C'est-à-dire que la méthode s'applique à l'objet lui-même. Il peut y avoir, ou non, d'autres paramètres. Le constructeur ne doit jamais renvoyer de résultat contrairement aux autres méthodes qui le peuvent. Tous les paramètres du constructeur (sauf `self`) peuvent avoir une valeur par défaut, précisé de la façon suivante directement dans la définition du constructeur : `__init__(self, param1 = valeurDefaut, param2 = valeur2)...`

C'est souvent dans le constructeur que les `attributs de classe` sont déclarés.

### 3) Afficher toutes les méthodes et attributs d'une classe ou d'un objet

Lorsqu'on programme avec des bibliothèques ou en équipe, on ne maîtrise pas toujours l'ensemble des classes et objets que nous utilisons. Afin de pouvoir facilement savoir les méthodes ou attributs d'une classe ou d'un objet, il suffit d'utiliser la fonction « dir » en python :

ou

## 4) Un exemple pratique en python :

```
class Chien():
    #
    nom_scientifique = "canis lupus"

    #
    def jouer():
        print("Le chien court et est content, Wouf wouf")

    def allerChezVeterinaire():
        print("Le chien pleurniche. wOuuuuuuf")

    def __init__(self,paramAge,paramRace,paramTaille):
        #
        self.age = paramAge
        self.race = paramRace
        self.taille = paramTaille

rex = Chien(12,"Berger allemand",75)
milou = Chien(15,"Caniche",22)
```

Si l'on fait un print de l'objet « rex » (qui est une de la classe ), on obtient :

```
<__main__.Chien object at 0x00000187EF3C5FD0>
```

Et si l'on affiche le résultat de la fonction dir :

```
print(dir(rex))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
 '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'age', 'allerChezVeterinaire', 'jouer',
 'nom_scientifique', 'race', 'taille']
```

### III. Améliorer la compréhension de la classe

#### 1) D'autres méthodes particulières : `__repr__` ou `__str__`

On a vu précédemment que `print(rex)` donnait le résultat `<__main__.Chien object at 0x00000187EF3C5FD0>` qui n'est pas très parlant. Afin d'obtenir quelque chose de plus parlant, nous pouvons utiliser la fonction :

```
def __repr__(self):  
    return("Le chien de la race "+self.race+" mesure  
"+str(self.taille)+"cm et a "+str(self.age)+" ans.")
```

On peut alors lancer la commande :

```
print(rex)
```

Le chien de la race Berger allemand mesure 75cm et a 12 ans.

La fonction `__repr__` permet d'utiliser le nom de l'objet directement dans la console, et d'obtenir le même résultat que `print(monObjet)`, alors que la fonction `__str__` ne donne le bon résultat que dans le cas de l'utilisation de `print(monObjet)`. Si on utilise `monObjet` directement dans la console, on obtiendra `<__main__.Chien object at 0x00000187EF3C5FD0>`. Il est donc préférable de définir `__repr__` systématiquement.

#### 2) Documenter

Notre classe est une structure de données qui peut être utilisée dans différents programmes par différents programmeurs, il est donc important voire primordial de bien la documenter.

Cette documentation doit montrer a minima comment sont construites les instances, quels sont les attributs et les méthodes disponibles. Cette documentation est accessible via l'instruction `help()` dans la console.

La documentation d'une classe est entourée de trois apostrophes comme dans l'exemple ci-dessous :

```
class Chien():  
    '''  
    La classe chien représente un animal.  
  
    Création d'une instance :  
        monInstance = Chien(paramAge(int),paramRace(str),paramTaille(float))  
  
    attributs de classe :  
        nom_scientifique  
  
    attributs d'instance :  
        age, race, taille  
  
    méthodes :  
        jouer() permet de jouer avec le chien  
        allerChezVeterinaire() permet de l'amener chez le véto  
    '''
```

On peut alors avoir accès à cette documentation simplement en faisant :

```
print(help(Chien))
```

Ce qui donne le résultat suivant :

```
Help on class Chien in module __main__:
class Chien(builtins.object)
| Chien(paramAge, paramRace, paramTaille)
|
| La classe chien représente un animal.
|
| Création d'une instance :
|   monInstance = Chien(paramAge(int),paramRace(str),paramTaille(float))
|
| attributs de classe :
|   nom_scientifique
|
| attributs d'instance :
|   age, race, taille
|
| méthodes :
|   jouer() permet de jouer avec le chien
|   allerChezVeterinaire() permet de l'amener chez le véto
```

## IV. Attributs publics ou privés

### 1) Le principe de l'encapsulation

Le but de l'encapsulation est de cacher la représentation interne des classes, qui apparaissent alors comme des "boîtes noires" au programmeur, qui n'a besoin de connaître que leur interface, c'est à dire l'ensemble des méthodes servant à manipuler les objets.

Intérêts de l'encapsulation :

- simplifier la vie du programmeur qui utilise une classe.
- masquer la complexité de la classe.
- la liste des méthodes devient une sorte de "mode d'emploi" de la classe.

C'est le principe de l'encapsulation. L'encapsulation est un principe qui consiste à regrouper des données avec un ensemble de méthodes permettant de les lire ou de les manipuler dans le but de cacher ou de protéger certaines de ces données.

Pour se faire, les attributs et méthodes peuvent être classés dans 3 catégories :

Les attributs publics, qui sont accessibles en lecture et modifiables par simple affectation :

- Les attributs protégés, qui sont hors programme en NSI
- Les attributs privés, qui ne sont ni accessibles en lecture ni modifiables. Par convention, en python ils commencent nécessairement par

## 2) Les accesseurs

Afin de pouvoir accéder à la valeur d'un attribut privé, on va créer une méthode appelée `get_experience`.

Par convention, un `getter` commence par le verbe anglais `get`.

Exemple :

```
class Personnage:
    """
    Un personnage du jeu vidéo
    """

    def __init__(self, genre, experience=0):
        self.genre = genre
        self.__experience = experience

    def get_experience(self): # on met toujours self dans une méthode
        return self.__experience
```

## 3) Les mutateurs

Afin de pouvoir modifier la valeur d'un attribut privé, on va créer une méthode appelée `set_experience`.

Par convention, un `setter` commence par le verbe anglais `set`.

Cette méthode est souvent privée, pour n'être utilisée que par les méthodes de l'objet lui-même !

Exemple : dans l'exemple suivant, nous l'avons laissée publique pour être utilisée plus facilement

```
class Personnage:
    """
    Un personnage du jeu vidéo
    """

    def __init__(self, genre, experience=0):
        self.genre = genre
        self.__experience = experience

    def get_experience(self): # on met toujours self dans une méthode
        return self.__experience

    def set_experience(self, valeur): # valeur sera le nouveau niveau d'exp
        self.__experience = valeur
```

Remarquez qu'aucun `return` n'est nécessaire ici pour le mutateur : la valeur de l'expérience est changée sans être renvoyée. Un peu comme si vous modifiez une variable globale.