Programmation orientée objet

Cours

La **programmation orientée objet**, ou **POO**, est un paradigme de programmation informatique. Elle consiste à définir des briques logicielles appelées objets et à décrire leurs comportements et interactions.

Un **objet** en programmation orientée objet représente souvent un concept, une idée ou toute entité du monde physique (une voiture, une personne, une page d'un livre, etc.).

Exemple de langages orientés objets : Java, C++, Python, etc.

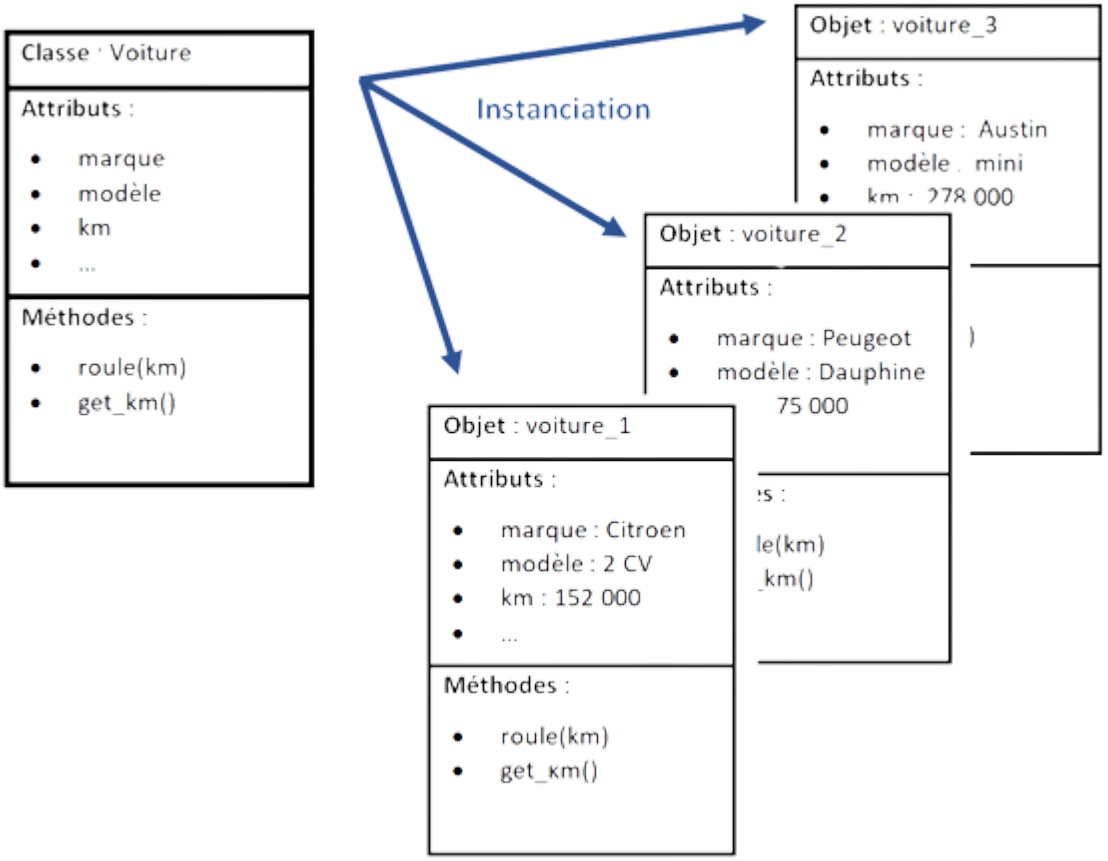
Cours

La **classe** est comme un "moule" à partir duquel des objets d'un même type peuvent être créés. Elle définit pour chaque objet de cette classe :

- des **attributs** comportant les informations concernant chaque objet ;
- des **méthodes** décrivant le comportement d'un objet.

Une fois une classe d'objet définie, il est possible de créer des objets sur le modèle de cette classe, c'est **l'instanciation**. **Les objets sont des instances de la classe**.

Prenons l'exemple d'un programme pour aider un collectionneur de vielles voitures qui possède une Citroen 2CV avec 152 000 km, une Peugeot Dauphine avec 75 000 km, etc. Une classe `Voiture`, décrivant les attributs et les méthodes d'une voiture, permet d'instancier plusieurs objets, chaque objet représentant une voiture : la Citroen 2CV (`voiture_1`), la Peugeot Dauphine (`voiture_2`), etc.



Les classes et objets

Commençons par créer la classe `Voiture` qui décrit les caractéristiques d'une voiture :

PEP 8

Les noms de classes s'écrivent en CamelCase : https://www.python.org/dev/peps/pep-0008/#class-names

```
class Voiture:
    pass
```

La classe `Voiture` n'est pas une voiture, c'est une sorte « d'usine à créer des voitures » ! Elle permet de créer par la suite des d'objets, des instances de la classe `Voiture`, sur le modèle de cette classe.

Créons maintenant les deux premières voitures du collectionneur, c'est-à-dire deux **instances** de la classe `Voiture` :

```
voiture_1 = Voiture()
voiture_2 = Voiture()
```

Cours

Pour créer un objet, ou **instancier**, `nom_objet` depuis la classe `NomClasse`, il faut écrire :

```
nom_objet = NomClasse()
```

Les attributs

Pour l'instant notre classe `Voiture` est une coquille vide et les deux objets instanciés à partir de cette classe, `voiture_1` et `voiture_2`, ne contiennent aucne information concernant ces voitures. Il est possible de leur rajouter des caractéristiques les décrivant avec des **attributs**, par exemple les attributs `marque`, `modele` et `km` pour l'instance `voiture_1` :

```
>>> voiture_1 = Voiture()
>>> voiture_1.marque = "Citroen"
>>> voiture_1.modele = "2 CV"
>>> voiture_1.km = 152000
```

et ensuite de lire les attributs ainsi :

```
>>> voiture_1.marque
'Citroen'
```

Mais les attributs `marque`, `modele` et `km` ont été créés pour l'objet `voiture_1`, ils n'existent pas pour les autres instances de `Voiture`, par exemple pour l'objet `voiture_2` :

```
>>> voiture_2.marque
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: 'Voiture' object has no attribute 'marque'
```

Cours

Un attribut `nom_attribut` d'un objet `nom_objet` s'obtient avec : `nom_objet.nom_attribut`

Constructeur

La création d'un nouvel objet avec l'instruction `voiture_1 = Voiture()` appelle une méthode particulière nommée `__init__`. C'est le **constructeur** de la classe. Il est possible de le modifier pour définir des attributs pour toutes les instances de `Voiture` dès leur instanciation :

⚠️ Attention aux deux blancs soulignés (ou tirets bas) au début et à la fin d'`init`, c'est souvent une erreur diffile à déceler quand il en manque un !

```
class Voiture:
    def __init__(self):
```

```
        self.marque = ""
        self.modele = ""
        self.km = 0
```

Noter le paramètre `self` de la fonction qui représente l'objet qui est instancié par la méthode `__init__`.

Toutes les instances de `Voiture` possèderont ces attributs, mais les valeurs peuvent être différentes pour chaque instance et évoluer différemment.

```
>>> voiture_1 = Voiture()
>>> voiture_1.modele
''
>>> voiture_1.modele = "2 CV"
>>> voiture_1.modele
'2 CV'
```

Avec ce constructeur, tous les objets sont créés avec les attributs initiés à la même valeur (`""` ou `0`). Ce n'est pas pratique. La méthode `__init__` est une méthode comme les autres, elle peut avoir des paramètres, par exemple la marque et le modèle de la voiture, ainsi que le nombre de kilomètre (assigné à la valeur 0 par défaut s'il n'est pas renseigné).

PEP 8

Comme les fonctions et les modules, les classes et les méthodes publiques comportent une docstring. Voir https://peps.python.org/pep-0008/#documentation-strings

```
class Voiture:
    """
    Classe d'objets représentants des voitures de collection

    Attributs:
    marque (str): la nom de la marque de la voiture
    modele (str): le modèle de la voiture
    km (int): les km parcourus par la voiture
    """

    def __init__(self, ma, mo, k=0):
        self.marque = ma
        self.modele = mo
        self.km = k
```

L'instruction `Voiture('Citroen', '2 CV')` appelle le constructeur `__init__` créant ainsi un nouvel objet `Voiture` en lui donnant les valeurs de ses attributs `'Citroen'`, `'2 CV'` et `152000`.

```
>>> voiture_1 = Voiture('Citroen', '2 CV', 152000)
>>> voiture_1.modele
'2 CV'
>>> voiture_1.km
152000
```

Les méthodes

Définir une classe c'est aussi définir les comportements communs aux objets de la classe : ce sont les **méthodes**. Les méthodes sont des fonctions définies dans une classe. Nous avons déjà vu la méthode `__init__` appelée à l'instanciation. On peut ajouter d'autres méthodes dans la classe `Voiture`.

Cours

Les méthodes prennent toujours comme **premier paramètre le mot réservé** `self` de façon à désigner l'objet sur lequel va s'appliquer la méthode.

Par exemple une méthode `roule(k)`, permettant d'ajouter `k` kilomètres quand un voiture, roule s'écrit :

```
class Voiture:
    """
    Classe d'objets représentants des voitures de collection

    Attributs:
    marque (str): la nom de la marque de la voiture
    modele (str): le modèle de la voiture
    km (int): les km parcourus par la voiture
    """

    def __init__(self,ma, mo, k=0):
        self.marque = ma
        self.modele = mo
        self.km = k

    def roule(self, k):
        self.km = self.km + k
```

Cours

Pour appeler une méthode `nom_methode(self, param1, param2..)` qui s'applique à un objet `nom_objet`, il faut écrire : `nom_objet.nom_methode(param1, param2 ,…)`.

Le paramètre `self` est toujours le premier paramètre dans la définition d'une méthode, il décrit l'objet sur lequel s'applique la méthode, **il n'apparaît pas dans les arguments de la méthode lors de l'appel**.

Appelons la méthode `roule(self, k)` :

```
>>> voiture_1 = Voiture('Citroen', '2 CV', 152000)
>>> voiture_1.km
152000
>>> voiture_1.roule(15000)
>>> voiture_1.km
167000
```

Ici quand la méthode `voiture_1.roule(15000)` est exécutée, `self` prend le nom de l'objet `voiture_1` et `k` la valeur `15000`.

Notons aussi qu'une méthode peut renvoyer une valeur. Prenons l'exemple de `revision`, une méthode qui renvoie les kilomètres avant la prochaine révision :

```
class Voiture:
    """
    Classe d'objets représentants des voitures de collection

    Attributs:
    marque (str): la nom de la marque de la voiture
    modele (str): le modèle de la voiture
    km (int): les km parcourus par la voiture

    Méthodes:
    roule(km): ajoute des km parcourus
    revision(): renvoie le nombre de km restant avant la prochaine revision des 15000
    """

    def __init__(self,ma, mo , k=0):
        self.marque = ma
        self.modele = mo
        self.km = k

    def roule(self, km):
        self.km = self.km + km

    def revision(self):
        return 15000 - self.km % 15000

>>> voiture_1 = Voiture('Citroen', '2 CV', 152000)
>>> voiture_1.revision()
13000
```

Il existe en Python quelques méthodes particulières. Comme `__init__()`, leur nom est entouré de deux paires de blancs soulignés. Par exemple l'instruction `>>> dir(list)` dans la console permet observer les méthodes des variables de type `list`.

Les paires de blancs soulignés (ou tirets bas) indiquent que ces méthodes ne sont pas appelées directement (elles sont privées) mais plutôt par le biais de fonctions particulières :

- `__init__(self)` est appelée à l'instanciation par `nom_objet = nom_classe()`.
- `__str__(self)` est appelée par la fonction `print()`. La valeur renvoyée par `__str__()` sera affichée quand on fera `print(nom_objet)` ou bien la chaîne de caractère renvoyée par `str(nom_objet)`.

  ```
  def __str__(self):
      return self.modele + ' '+ self.marque
  ```

  ⚠️Noter l'utilisation de `return` et non de `print()` dans la définition de la méthode `__str__(self)`, c'est souvent risque d'erreurs.

  ```
  >>> print(voiture_1)
  2 CV Citroen
  ```

- `__len__(self)` est appelée par `len(nom_objet)`.
- `__add__(self, other)` pour ajouter deux objet avec le signe « `+` »; `__mul__(self, other)` pour les multiplier avec « `*` »; `__lt__(self, other)` pour comparer avec « `<` », etc.

Alias

Lorsqu'un objet est assigné à une variable, par exemple `voiture_1 = Voiture('Citroen', '2 CV')`, la variable est une référence à cet objet, c'est-à-dire son adresse mémoire. Dès lors, deux variables peuvent faire référence au même objet, ce sont des **alias**. On peut accéder ou modifier l'objet par l'une ou l'autre. ⚠️ Cela mène à de nombreuses erreurs de programmation.

```
>>> voiture_2 = Voiture('Peugeot','Dauphine', 75254)
>>> v = voiture_2
```

Les deux variables `voiture_2` et `v` pointent maintenant sur le même objet :

```
>>> voiture_2
<__main__.Voiture object at 0x02E102C8>
>>> v
<__main__.Voiture object at 0x02E102C8>
>>>
```

Et par conséquent, une modification de l'un modifie l'autre :

```
>>> voiture_2.km
75254
>>> v.roule(10000)
>>> voiture_2.km
85254
```

Variable de classe

Nous avons défini une classe comportant des attributs et des méthodes. Les attributs sont déclarés à l'intérieur du constructeur de l'objet et prennent donc une valeur qui est propre à chaque objet instancié de la classe. Deux objets différents appartenant à la même classe ont des valeurs d'attributs qui peuvent être différents. C'est pourquoi les attributs sont aussi appelés des **variables d'instance**.

Dans notre exemple, la classe `Voiture` définit les attributs et les méthodes qui s'appliquent à chaque voiture, c'est-à-dire à chaque instance de cette classe. Mais comment connaître le nombre total d'instances qui ont été créées à partir ce cette classe ? Ce n'est pas une valeur qui est propre à une instance en particulier. Elle concerne plutôt toute la classe.

À l'opposé de ces variables d'instance, il est utile dans certains cas d'avoir des variables dont la valeur est commune à toutes les instances de la même classe. Ces variables sont des **variables de classe**. La valeur d'une variable de classe est partagée par toutes les instances de cette classe. Chacune des instances de la classe peut la lire ou la modifier.

Une variable de classe est déclarée en dehors du constructeur de la classe. Noter qu'à la différence d'une variable d'instance, elle ne commence pas par `self.` puisqu'elle ne s'applique pas à une instance en particulier :

```
class Voiture:
    total_voiture = 0

    def __init__(self,ma, mo, k=0):
        self.marque = ma
        self.modele = mo
        self.km = k
        Voiture.total_voiture += 1
```

puis `NomClasse.nom_variable_de_classe` permet d'utiliser cette variable de classe (en opposition à `nom_objet.nom_attribut` pour une variable d'instance) :

```
>>> Voiture.total_voiture
0
>>> voiture_1 = Voiture('Citroen', '2 CV', 152000)
>>> Voiture.total_voiture
1
>>> voiture_1 = Voiture('Peugeot','Dauphine', 75000)
>>> Voiture.total_voiture
2
>>>
```

Encapsulation

Ajoutons maintenant une nouvelle variable de classe `total_km` qui garde en mémoire le total des kilomètres parcourus par toutes les instances de `Voiture` :

```
class Voiture:
    total_voiture = 0
    total_km = 0

    def __init__(self,ma, mo, k=0):
        self.marque = ma
        self.modele = mo
        self.km = k
        Voiture.total_voiture += 1
        Voiture.total_km += k

    def roule(self, k):
        self.km=self.km + k
        Voiture.total_km += k
```

puis créons deux voitures et faisons rouler une des deux :

```
>>> Voiture.total_km
0
>>> voiture_1 = Voiture('Citroen', '2 CV', 125000)
>>> voiture_2 = Voiture('Peugeot','Dauphine', 75000)
>>> Voiture.total_km
200000
>>> voiture_2.roule(10000)
>>> Voiture.total_km
230000
```

Jusqu'ici tout va bien. Mais que se passe-t-il si on change la valeur de l'attribut `km` d'une instance de `Voiture` directement ?

```
>>> voiture_2.km = 100000
>>> Voiture.total_km
230000
```

L'attribut `km` de `voiture_2` a changé mais pas la valeur de la variable de classe `total_km`, elle n'est plus correcte ! C'est un problème.

Pour éviter ce genre de problème, il faut « protéger » la variable de classe `total_km` pour que sa valeur ne soit pas modifiée directement. C'est comme si cette variable `total_km` était mise à « l'intérieur d'une boîte interne » à l'objet, cachée de « l'exterieur », afin qu'elle ne soit lue et modifiée qu'en utilisant des méthodes qui garantissent que sa valeur reste correcte. C'est le méchanisme d'encapsulation.

Cours

L'**encapsulation** consiste à « enfermer » certains attributs et certaines méthodes à l'intérieur d'un objet pour qu'ils ne soient pas accessibles directement depuis « l'extérieur » de cet objet.

Les attributs et méthodes qui ne sont pas accessibles depuis l'extérieur de l'objet sont des **attributs et méthodes privés**. Ceux qui restent accessibles sont des **attributs et méthodes publics**.

En Python, un **simple blanc souligné**, ou tiret bas, au début d'un nom d'attribut indique que cet attribut est privé, par exemple dans notre exemple `self._km`.

```
class Voiture:
    total_voiture = 0
    total_km = 0

    def __init__(self,ma, mo, k=0):
        self.marque = ma
        self.modele = mo
        self._km = k
        Voiture.total_voiture += 1
        Voiture.total_km += k

    def roule(self, k):
        self._km = self._km + k
        Voiture.total_km += k
    ...
```

Pour respecter le principe de l'encapsulation, il faut éviter de lire ou écrire la valeur de l'attribut `_km` d'une instance de `Voiture` directement depuis « l'exterieur » de l'objet.

Cours

Une classe doit fournir des méthodes (publiques) qui font l'**interface** avec l'extérieur :

- **accesseurs** (ou ***getters*** par convention leur nom commence par **get**) les méthodes permettant d'obtenir la valeur d'un attribut, et
- **mutateurs** (ou ***setters***, par convention leur nom commence par **set**) pour en modifier la valeur.

```
class Voiture:
    total_voiture = 0
    total_km = 0

    def __init__(self, ma, mo, k=0):
        self.marque = ma
        self.modele = mo
        self._km = k
        Voiture.total_voiture += 1
```

```
        Voiture.total_km += k

    def get_km(self):
        return self._km

    def set_km(self, k):
        Voiture.total_km -= self.km        # on soustrait l'ancienne valeur de km
        self._km = k
        Voiture.total_km += k               # on rajoute la nouvelle valeur

    def roule(self, k):
        self._km = self._km + k
        Voiture.total_km += k
```

Noter que ce **simple blanc souligné**, ou tiret bas, au début du nom de l'attribut n'est qu'une convention d'écriture entre programmeurs, elle n'est pas prise en compte par l'interpréteur et il est toujours possible de le lire et de le modifier directement sans passer par son mutateur.

```
>>> voiture_1 = Voiture('Citroen', '2 CV', 152000)
>>> voiture_1._km
152000
>>> voiture_1._km = 160000
>>> voiture_1._km
160000
```

Le problème persiste !

Pour aller plus loin, certains programmeurs utilisent aussi un **double blanc souligné** au début d'un nom d'attribut privé.

```
class Voiture:
    def __init__(self, ma, mo, k=0):
        #...
        self.__km = k
        #...
```

Dans ce cas, l'attribut `__km` ne peut plus être lu par ``voiture_1.__km`` :

```
>>> voiture_1 = Voiture('Citroen', '2 CV', 152000)
>>> voiture_1.__km
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: 'Voiture' object has no attribute '__km'
```

mais il peut encore être modifié avec des effets indésirables :

```
>>> voiture_1.__km = 160000
>>> voiture_1.__km
160000
>>> voiture_1.get_km()
152000
```

Le problème persiste, cette utilisation est contestée[1].

Pour aller encore plus loin (hors programme) une autre manière « Pythonesque » de respecter le principe d'encapsulation est d'utiliser des décorateurs pour transformer les attributs en propriétés (hors programme)[2] .

Héritage et polymorphisme (hors programme)

Cours

L'**héritage** consiste à créer une **nouvelle classe (la classe fille) à partir d'une classe existante** (super classe ou classe mère).

Cela permet de définir de nouveaux attributs et de nouvelles méthodes pour la classe fille, qui s'ajoutent à ceux et celles héritées de la classe mère sans avoir à les réécrire.

Admettons que l'on veuille créer une classe de voiture électrique qui a les propriétés de la classe `Voiture` plus un certain nombre de kWh pour 100 kilomètres.

La définition de la classe fille mentionne la mère. La méthode `__init__` appelle le constructeur de la mère et permet d'ajouter des attributs. On pourrait aussi ajouter des méthodes propres à la classe fille.

```
class VoitureElectrique(Voiture):

    def __init__(self, ma, mo, k=0, kwh=0):
        Voiture.__init__(self, ma, mo, k)     # ou super().__init__()
        self.kwh = kwh
```

Noter la syntaxe `class VoitureElectrique(Voiture)` qui indique le lien mère-fille entre les classe `Voiture` et `VoitureElectrique`. Créons maintenant une instance de `VoitureElectrique` :

```
>>> voiture_3 = VoitureElectrique('TESLA', 'S', kwh=18)
```

Toutes les méthodes de la classe mère s'appliquent à la fille :

```
>>> voiture_3.roule(1000)
>>> voiture_3.get_km()
1000
```

Cours

Le **polymorphisme** permet de modifier le comportement d'une classe fille par rapport à sa classe mère.

Cela permet d'adapter le comportement des objets. Par exemple, créons une classe fille pour des voitures qui ne roulent plus, appelée `Epave` et modifions la méthode `roule()` de cette classe :

```
class Epave(Voiture):

     # inutile de redéfinir __init__ on utilise le constructeur de la classe mère

    def roule(self, k):
        pass
```

La méthode `roule()` d'une instance d'`Epave` n'est plus héritée de celle de la classe mère `Voiture`, elle se comporte différement :

```
>>> voiture_4 = Epave('Trabant','601', 150000)
>>> voiture_4.roule(10000)
>>> voiture_4.get_km()
150000
>>>
```

---

1. Règle du « name mangling » : Voir PEP 8 « Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed. Note: there is some controversy about the use of __names (see below). » ↩
2. Une classe Voiture en utilisant des décorateurs Python :

```
class Voiture:

    total_voiture = 0
    total_km = 0

    def __init__(self, ma, mo, k=0):
        self.marque = ma
        self.modele = mo
        self._km = k
        Voiture.total_voiture += 1
        Voiture.total_km += k

    # propriété km. Permet d'utiliser nom_objet.km
    @property
    def km(self):
        return self._km

    # setter de l'attribut km. Pour utiliser nom_objet.km = ….
    @km.setter
    def km(self, k):
        Voiture.total_km -= self._km
        self._km = k
        Voiture.total_km += k

    def roule(self, k):
        self._km=self._km + k
        Voiture.total_km += k

voiture_1 = Voiture('Citroen','2 CV')
voiture_1.km = 152000
```

↩