Programmation Orientée Objet

Source : Revue Coding, n°16, p.72-79

C'est un paradigme de programmation qui fournit un moyen de structurer les programmes de sorte que les propriétés et les comportements soient regroupés dans des objets individuels.

La POO est une approche pour modéliser des choses concrètes et réelles (comme les voitures). Elle modélise les entités du monde réel en tant qu'objets logiciels auxquels sont associées des données et qui peuvent exécuter certaines fonctions.

Un autre paradigme de programmation courant est la programmation procédurale, qui structure un programme comme une recette en ce sens qu'il fournit un ensemble d'étapes, sous la forme de fonctions et de blocs de code, qui se déroulent de manière séquentielle afin d'accomplir une tâche.

L'essentiel à retenir est que les objets sont au centre de la programmation orientée objet en Python, représentant non seulemnt les données, comme dans la programmation procédurale, mais également dans la structure globale du programme.

Différence entre classes et instances

Les classes sont utilisées pour créer des structures de données définies par l'utilisateur. Les classes définissent des fonctions appelées méthodes, qui identifient les comportements et les actions qu'un objet créé à partir de la classe peut effectuer avec ses données.

Une classe est un modèle pour la façon dont quelque chose doit être défini. Il ne contient en fait aucune donnée.

Une instance est un objet qui est construit à partir d'une classe et contient des données réelles.

In [1]:
```python
class Chien:
    # attributs de classe
    espece = "Canis familiaris"

    # méthode
    def __init__(self, nom, age):
        # attributs d'instances
        self.nom = nom
        self.age = age
```

Les attributs créés dans `__init__` sont des **attributs d'instances**. La valeur d'un attribut d'instance est spécifique à une instance particulière de la classe.

Les attributs de classe sont des attributs qui ont la même valeur pour toutes les instances de classe.

On utilise des attributs de classe pour définir des propriétés qui doivent avoir la même valeur pour chaque instance de classe.

On utilise des attributs d'instance pour les propriétés qui varient d'une instance à l'autre.

Instancier un objet en Python

Créer un nouvel objet à partir d'une classe s'appelle instancier un objet. On peut instancier un nouvel objet **Chien** en tapant le nom de la classe, suivi des paranthèses :

`Chien()`

Chaque nouvelle instance se trouve à une adresse mémoire différente. Chaque instance est donc entièrement nouvelle et unique.

Exemple

In [5]:
```
class Chien :
    pass

a = Chien()
print(a)

b = Chien ()
print(b)

# on compare les 2 variables entre elles
a == b
```

Out[5]:
```
<__main__.Chien object at 0x0000018ED132E550>
<__main__.Chien object at 0x0000018ED0AA80D0>
False
```

Attributs de classe et d'instance

In [6]:
```
class Chien:
    # attributs de classe
    espece = "Canis familiaris"

    # méthode
    def __init__(self, nom, age):
        # attributs d'instances
        self.nom = nom
        self.age = age
```

Pour instancier des objets de cette classe, il faut fournir des valeurs pour le nom et l'âge, sinon Python renvoie une erreur.

Note : Il n'y a pas besoin de fournir des valeurs pour le paramètre `self`. Dans notre cas, il n'y a que 2 paramètres, `nom` et `age`.

In [7]:
```
toutou = Chien("Toutou", 5)
```

Pour accéder aux attributs d'instance, il faut le nom de la variable suivi d'un point, puis du

paramètre de la classe. Idem pour les attributs de classe.

In [55]:
```
print(toutou.nom)
print(toutou.age)
print(toutou.espece)
```

```
Toutou
6
Canis familiaris
```

Les valeurs des attributs peuvent être modifiées dynamiquement :

In [13]:
```
toutou.age = 7
toutou.age
```

Out[13]:
```
7
```

In [14]:
```
toutou.espece = 'Ceci est un chien'
toutou.espece
```

Out[14]:
```
'Ceci est un chien'
```

Les objets personnalisés sont donc modifiables par défaut.

Méthodes d'instance

Ce sont des fonctions définies à l'intérieur d'une classe et qui ne peuvent être appelées qu'à partir d'une instance de cette classe. Le 1er paramètre d'une méthode d'instance est toujours `self`.

In [21]:
```
class Chien:
    espece = "Canis familiaris"

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    # Instance : renvoie une chaîne affichant le nom et l'âge du chien.
    def description(self):
        return f"{self.nom} est âgé de {self.age} ans"

    # Instance : renvoie le nom et le son que fait le chien.
    def aboie(self, son):
        return f"{self.nom} aboie {son}"
```

In [22]:
```
# Instance d'un nouvel objet
toutou = Chien("Toutou", 5)
```

In [23]:
```
# On appelle la méthode
toutou.description()
```

Out[23]:
```
'Toutou est âgé de 5 ans'
```

In [24]:
```
# Idem, en passant un paramètre
toutou.aboie("Waf waf")
```

Out[24]:
```
'Toutou aboie Waf waf'
```

Accéder aux attributs d'instance en str

In [25]:
```
print(toutou)
```

```
<__main__.Chien object at 0x0000018ED0B7B520>
```

Il faut remplacer la méthode `.description()` par `__str__()`

In [31]:
```
class Chien:
    espece = "Canis familiaris"

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    # remplacer .description() par __str__()
    def __str__(self):
        return f"{self.nom} est âgé de {self.age} ans"

    def aboie(self, son):
        return f"{self.nom} aboie {son}"
```

In [32]:
```
toutou = Chien("Toutou", 5)
print(toutou)
```

```
Toutou est âgé de 5 ans
```

La notion d'héritage

L'héritage est le processus par lequel une classe prend les attributs et les méthodes d'une autre. Les classes nouvellements formés sont appelées **classes enfants** et les anciennes **classes parents**.

Les classes enfants héritent de tous les attributs et les méthodes du parent, mais peuvent également spécifier des attributs et des méthodes qui leur sont propres.

Les modifications apportées à la classe parente se propage automatiquement aux classes enfants. Cela se produit tant que l'attribut ou la méthode en cours de modification n'est pas remplacé dans la classe enfant.

Exemple :

On modifie l'aboiement des chiens en fonction de leur race, en créant une classe enfant.

In [47]:
```
class Chien :
    espece = "Canis familiaris"

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def __str__(self):
        return f"{self.nom} est âgé de {self.age} ans"

    def aboie(self, son):
        return f"{self.nom} se met à aboyer : {son}"
```

```
# Classes enfants :
class JackRussellTerrier(Chien):
    pass

class Dachshund(Chien):
    pass

class Bouledogue(Chien):
    pass
```

In [50]:
```
medor = Bouledogue("Medor", 4)

# On peut vérifier la classe
type(medor)
```

Out[50]:
```
__main__.Bouledogue
```

In [40]:
```
# Vérifier si la classe fait partie de la classe 'Chien'
isinstance(medor, Chien)
```

Out[40]:
```
True
```

Personnalisation des classes enfants

Pour remplacer une méthode définie sur la classe parent, il faut définir une méthode avec le même nom sur la classe enfant.

In [53]:
```
# Classes enfants :
class JackRussellTerrier(Chien):
    def aboie(self, son='Arf'):
        return f"{self.nom} aboie {son}"

class Dachshund(Chien):
    pass

class Bouledogue(Chien):
    pass
```

In [54]:
```
toutou = JackRussellTerrier("Toutou", 6)
toutou.aboie()
```

Out[54]:
```
'Toutou aboie Arf'
```

Méthode `super()`

Appelle d'une méthode de la classe parent dans la classe enfant.

In [48]:
```
class JackRussellTerrier(Chien):
    def aboie(self, son='Arf'):
        return super().aboie(son)
```

In [49]:
```
dannythedog = JackRussellTerrier("Danny the Dog", 6)
dannythedog.aboie()
```

Out[49]:
```
'Danny the Dog se met à aboyer : Arf'
```