Langage Python

2020-2021

Hamza

Programmation Orientée Objet

La programmation orientée objet

- **La programmation orientée objet (POO)** consiste à définir des objets logiciels et à les faire interagir entre eux.

- Les objets sont utilisé pour modéliser informatiquement des "***objets***" de la vie courante (voiture, personne, etc.) ou des concepts (date, couleur, etc.).

Notion de classe

- **Une classe** déclare des propriétés communes à un ensemble d'objets.
- Une classe représentera donc une catégorie d'objets.
- Elle apparaît comme un type ou un moule à partir duquel il sera possible de créer des objets.

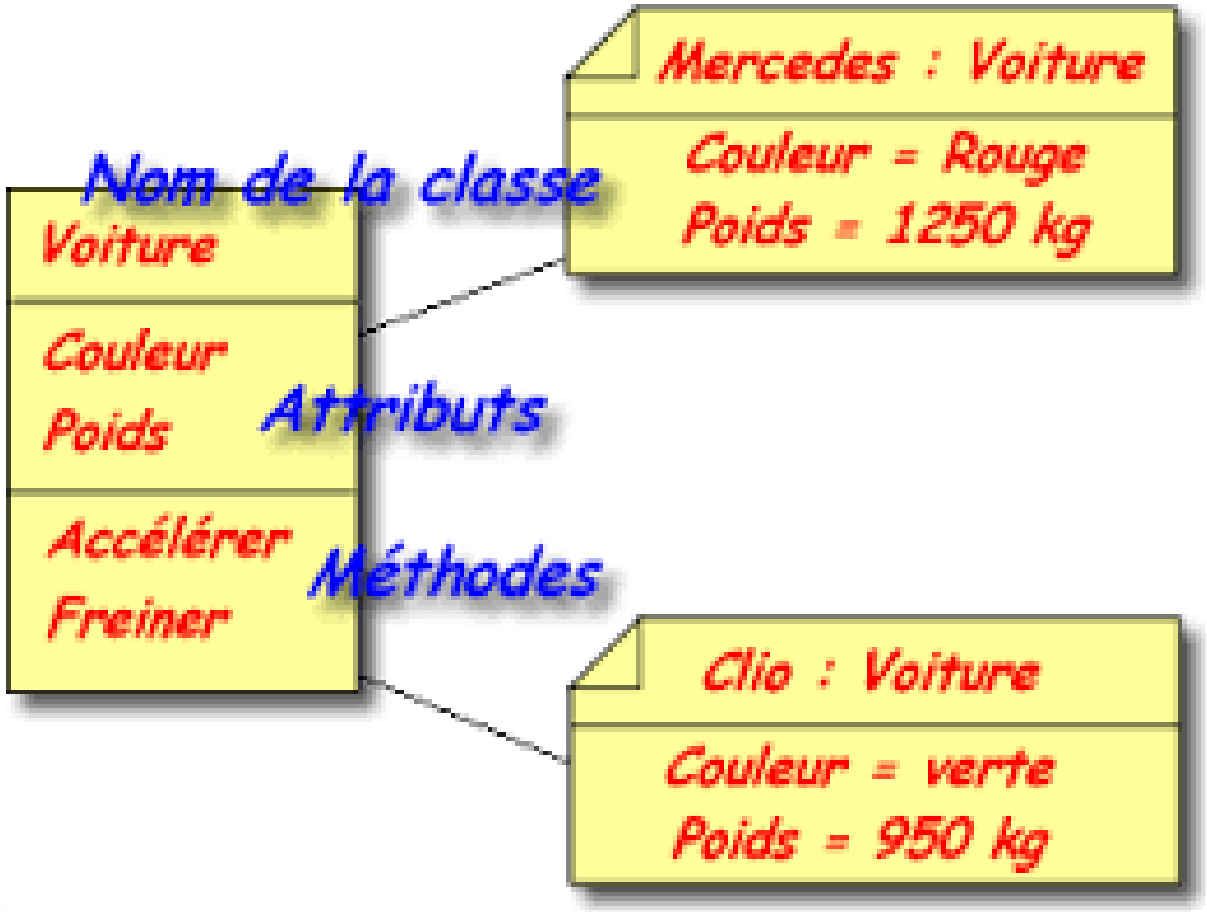
Notion d'objets

- **Un objet** est un conteneur symbolique et autonome qui contient des informations et des mécanismes concernant une entité , manipulés dans un programme.

- Un objet est créé à partir d'une classe. Chaque objet créé à partir de cette classe est une instance de la classe en question.

- Un objet possède une identité qui permet de distinguer un objet d'un autre objet (son nom, une adresse mémoire)

Exemple


Mercedes : Voiture
Couleur = Rouge
Poids = 1250 kg
Nom de la classe
Voiture
Couleur
Poids
Attributs
Accélérer
Freiner
Méthodes
Clio : Voiture
Couleur = verte
Poids = 950 kg

Déclaration de classes en Python

```
class NomDeLaClasse :
    pass
```

- l'instruction ***pass*** sert ici à indiquer à Python que le corps de notre classe est vide
- Il est conseillé en Python de nommer la classe en *CamelCase*, c'est à dire qu'un nom est composé d'une suite de mots dont la première lettre est une capitale.
- On préférera par exemple une classe ***NomDeClasse*** que ***nom_de_classe***.

Instanciation

```
monObjet = NomDeLaClasse()
```

Déclaration des méthodes

```
def maMethode(self,[arguments]):
    // corps de la méthode
```

Une méthode de la classe s'écrit comme une fonction mais avec un premier paramètre self obligatoire, où self représente l'objet sur lequel la méthode sera appliquée.

Autrement dit self est la référence d'instance

Déclaration et Instanciation d'une classes en Python

```
class MaClasse :
    n = 0
    def afficher(self) :
        print("n=",n)
```

```
monObj=MaClasse()

monObj.afficher()
```

Notion d'encapsulation

L'encapsulation est l'idée de protéger les variables contenues dans un objet et de ne proposer que des méthodes pour les manipuler.

- En respectant ce principe, toutes les attributs d'une classe seront donc privées.

L'objet est ainsi vu de l'extérieur comme une «*boîte noire*» possédant certaines propriétés et ayant un comportement spécifié. C'est le comportement d'un objet qui modifiera son état.

Notion de visibilité

La POO permet de préciser le type d'accès des membres (attributs et méthodes) d'un objet. Cette opération s'effectue au sein des classes de ces objets :

- **public** : les membres publics peuvent être utilisés dans et par n'importe quelle partie du programme.
- **private** : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et non par ceux d'une autre classe.
- **protected** : les membres protégés d'une classe sont accessibles par les objets de cette classe et par les objets des sous-classes (héritage).

Notion de visibilité

En Python, tous les attributs (données, méthodes) sont publics !

Une simple convention courante est d'utiliser des noms commençant par

- un caractère de soulignement(tiret bas _) pour signifier attribut protégé.
- deux caractères de soulignement (__) pour signifier attribut privé.

L'initialisateur `__init__`

- Lors de l'instanciation d'un objet, la méthode __init__ est automatiquement invoquée.
- La méthode permet d'effectuer toutes les initialisations nécessaires
- Cette méthode ne contient jamais l'instruction ***return***.

```
class Voiture:
    def __init__(self):
        self.nom = "Ferrari"
```

- ***__init__*** n'est pas un constructeur.
- ***__new__*** est le vrai constructeur, elle est appelée avant la création de l'objet, car son rôle est de créer l'instance et de la retourner.

Accesseurs et mutateurs

Parmi les différentes méthodes que comporte une classe, on a souvent tendance à distinguer :

- Les accesseurs (getters) qui fournissent des informations relatives à l'état d'un objet, c'est-à-dire aux valeurs de certains de ses attributs (généralement privés) sans les modifier ;
- Les mutateurs (setters) qui modifient l'état d'un objet, donc les valeurs de certains de ses attributs.

```
class Voiture:
      def __init__(self, nom):
            self.__nom = nom
      def get_nom(self):
            return self.__nom
      def set_nom(self, nom):
            self.__nom = nom
```

Attribut d'instance & Attribut de classe

Un ***attribut d'instance*** est une variable accrochée à une instance et qui est spécifique à cette instance. Cet attribut n'existe donc pas forcément pour toutes les instances, et d'une instance à l'autre il ne prendra pas forcément la même valeur.

Pour en créer, on a vu qu'il suffisait de les initialiser dans la méthode ***__init__*** en utilisant la syntaxe ***self.nomAttribut***

Un ***attribut de classe*** est un attribut qui sera identique pour chaque instance. Les attributs de classe ne peuvent pas être modifiés ni à l'extérieur d'une classe via une syntaxe ***monObj.attrClasse=valeur,*** ni à l'intérieur via une syntaxe ***self. attrClasse = valeur.***

Puisqu'ils sont destinés à être identiques pour toutes les instances. Les attributs de classe Python ressemblent en quelque sorte aux attributs statiques du C++

Attribut d'instance & Attribut de classe

En général, les variables d'instance stockent des informations relatives à chaque instance alors que les variables de classe servent à stocker les attributs et méthodes communes à toutes les instances de la classe :

```
class Chien:
        type = 'canin' # variable de classe partagée par toutes les instances
      def __init__(self, nom):
          self.nom = nom      # variable d'instance unique à chaque instance

>>> d = Chien('Fido')
>>> e = Chien('Buddy')
>>> d.type     # partagé par tous les chiens
'canin'
>>> e.type     # partagé par tous les chiens
'canin'
>>> d.nom      # unique à d
'Fido'
>>> e.nom      # unique à e
'Buddy'
```

Méthodes spéciales : __str__

- ***__str__*** est une méthode spéciale, comme ***__init__***, qui est censée renvoyer une représentation sous forme de chaîne de caractères d'un objet.
- Par exemple:

```
class Temps:
        def __init__(self, heure = 0, minute = 0, seconde = 0):
            self.heure = heure
            self.minute = minute
            self.seconde = seconde

        def __str__(self):
                return '%.2d:%.2d:%.2d' % (self.heure, self.minute, self.seconde)


>>> temps = Temps(9, 45)
>>> print(temps)
09:45:00
```

Méthodes spéciales : Surcharge d'opérateur

- Surcharger l'opérateur consiste à définir une méthode qui désigne la fonction à 2 opérandes associée à un opérateur, Python permet de surcharger les opérateurs tels que +, =, +=,>,.....
- Fondamentalement, l'appel à un opérateur est identique à l'appel d'une fonction
- Toutes les classes ne sont pas forcément adaptées à la surcharge d'opérateurs.

Exemple:

```
class Temps:
        def __init__(self, heure = 0, minute = 0, seconde = 0):
            self.heure = heure
            self.minute = minute
            self.seconde = seconde

        def __add__(self,T):
          return Temps(self.heure+T.heure, self.minute+T. minute, self.seconde+T.seconde)
```

Méthodes spéciales : Opérations mathématiques

opération	symbole	méthode	symbole unaire	méthode
addition	+	__add__(self,other)	+=	__iadd__(self,other)
soustraction	-	__sub__(self,other)	-=	__isub__(self,other)
multiplication	*	__mul__(self,other)	*=	__imul__(self,other)
division	/	__truediv__(self,other)	/=	__itruediv__(self,other)
élévation à la puissance	**	__pow__(self,other)	**=	__ipow__(self,other)
division entière	//	__floordiv__(self,other)	//=	__ifloordiv__(self,other)
reste de la division entière (modulo)	%	__mod__(self,other)	%=	__imod__(self,other)

Méthodes spéciales : Surcharge d'opérateur

Opérateur	Méthode	Expression
<< Changement bit à gauche	__lshift__(self, other)	a1 << a2
>> Changement de bit à droite	__rshift__(self, other)	a1 >> a2
& Bitwise ET	__and__(self, other)	a1 & a2
^ Bit-bit XOR	__xor__(self, other)	a1 ^ a2
| (Bit à bit OU)	__or__(self, other)	a1 | a2
- Négation (arithmétique)	__neg__(self)	-a1
+ Positif	__pos__(self)	+a1
~ Pas binaire	__invert__(self)	~a1
< Moins que	__lt__(self, other)	a1 < a2
<= Inférieur ou égal à	__le__(self, other)	a1 <= a2
== égal à	__eq__(self, other)	a1 == a2
!= Pas égal à	__ne__(self, other)	a1 != a2
> Supérieur à	__gt__(self, other)	a1 > a2
>= Supérieur ou égal à	__ge__(self, other)	a1 >= a2
[index] Opérateur d'index	__getitem__(self, index)	a1[index]
in opérateur In	__contains__(self, other)	a2 in a1
(*args, ...) Appel	__call__(self, *args, **kwargs)	a1(*args, **kwargs)

Méthodes spéciales :Fonctions

Fonction	Méthode	Expression
Casting à l' int	__int__(self)	int(a1)
Fonction absolue	__abs__(self)	abs(a1)
Casting à str	__str__(self)	str(a1)
Représentation de chaîne	__repr__(self)	repr(a1)
Casting à bool	__nonzero__(self)	bool(a1)
Formatage de chaîne	__format__(self, formatstr)	"Hi {:abc}".format(a1)
Hachage	__hash__(self)	hash(a1)
Longueur	__len__(self)	len(a1)
Renversé	__reversed__(self)	reversed(a1)
Partie entière	__floor__(self)	math.floor(a1)
Partie entière+1	__ceil__(self)	math.ceil(a1)