Programmation orientée objet en python / classes

La programmation orientée objet (POO) permet de créer des entités (objets) que l'on peut manipuler . La programmation orientée objet impose des structures solides et claires. Les objets peuvent interagir entre eux, cela facilite grandement la compréhension du code et sa maintenance. On oppose souvent la programmation objet à la programmation procédurale , la première étant plus “professionnelle” que l'autre car plus fiable et plus propre. Les classes

Une classe regroupe des fonctions et des attributs qui définissent un objet. On appelle par ailleurs les fonctions d'une classe des “ méthodes ”.

Créons une classe Voiture :

class001.py

```
# coding: utf-8

class Voiture:

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

Notre classe Voiture est une sorte d'usine à créer des voitures.

La méthode __init__() est appelée lors de la création d'un objet.

self.nom est une manière de stocker une information dans la classe. On parle d'attribut de classe. Dans notre cas, on stock le nom dans l'attribut nom .

Les objets

Un objet est une instance d'une classe . On peut créer autant d'objets que l'on désire avec une classe .

Créons maintenant notre voiture:

```
>>> ma_voiture = Voiture()
```

Les attributs de class

Les attributs de classe permettent de stocker des informations au niveau de la classe. Elle sont similaires aux variables.

Dans notre exemple:

```
>>> ma_voiture = Voiture()
>>> ma_voiture.nom
'Ferrari'
```

Vous pouvez à tout moment créer un attribut pour votre objet:

```
>>> ma_voiture.modele = "250"
```

Et le lire ainsi:

```
>>> ma_voiture.modele
'250'
```

Les méthodes

Les méthodes sont des fonctions définies dans une classe.

Créons une nouvelle méthode dans notre classe voiture:

class002.py

```
# coding: utf-8

class Voiture:

    def __init__(self):
        self.nom = "Ferrari"

    def donne_moi_le_modele(self):
        return "250"
```

Utilison cette méthode:

```
>>> ma_voiture=Voiture()
>>> ma_voiture.donne_moi_le_modele()
'250'
```

Les propriétés

Quelque soit le langage, pour la programmation orientée objet il est de préférable de passer par des propriétés pour changer les valeurs des attributs. Alors bien que cela ne soit pas obligatoire, il existe une convention de passer par des getter (ou accesseur en francais) et des setter (mutateurs) pour changer la valeur d'un attribut. Cela permet de garder une cohérence pour le programmeur, si je change un attribut souvent cela peut également impacter d'autres attributs et les mutateurs permettent de faire cette modification une fois pour toute.

Un exemple d'utilisation de propriétés:

class003.py

```
# coding: utf-8

class Voiture(object):

    def __init__(self):
        self._roues=4

    def _get_roues(self):
        print "Récupération du nombre de roues"
        return self._roues

    def _set_roues(self, v):
        print "Changement du nombre de roues"
        self._roues  =  v

    roues=property(_get_roues, _set_roues)
```

Quand on changera la valeur du nombre de roues, un message apparaîtra. En soi cela n'apporte rien mais au lieu de faire un simple print , vous pouvez par exemple envoyer un mail, etc.

Testons notre classe:

```
>>> ma_voiture=Voiture()
>>> ma_voiture.roues=5
```

Changement du nombre de roues

```
>>> ma_voiture.roues
```

Récupération du nombre de roues

```
5
```

Il existe une autre syntaxe en passant par des décorateurs:

class004.py

```
class Voiture(object):

    def __init__(self):
        self._roues=4

    @property
    def roues(self):
        print "Récupération du nombre de roues"
```

```
        return self._roues

    @roues.setter
    def roues(self, v):
        print "Changement du nombre de roues"
        self._roues  =  v
```

Le résultat sera le même, mais la lecture du code se trouve amélioré.

La fonction dir

Parfois il est intéressant de decortiquer un objet pour résoudre à un bug ou pour comprendre un script.

La fonction dir vous donne un aperçu des méthodes de l'objet:

```
>>> dir(ma_voiture)
```

['__doc__', '__init__', '__module__', 'donne_moi_le_modele', 'nom'] L'attribut spécial __dict__

Cet attribut spécial vous donne les valeurs des attributs de l'instance:

```
>>> ma_voiture.__dict__
{'nom': 'Ferrari'}
```

L'héritage de class

L'héritage est un concept très utile. Cela permet de créer de nouvelles classes mais avec une base existante.

Gardons l'exemple de la voiture et créons une classe VoitureSport :

class005.py

```
class Voiture:

    roues = 4
    moteur = 1

    def __init__(self):
        self.nom = "A déterminer"

class VoitureSport(Voiture):

    def __init__(self):
```

```
        self.nom = "Ferrari"
```

On a indiqué que VoitureSport a hérité de classe Voiture , elle recupère donc toutes ses méthodes et ses attributs.

On peut toujours instancier la classe Voiture si on le désire:

```
>>> ma_voiture=Voiture()
>>> ma_voiture.nom
'A déterminer'
>>> ma_voiture.roues
4
```

Instancions maintenant la classe VoitureSport :

```
>>> ma_voiture_sport=VoitureSport()
>>> ma_voiture_sport.nom
'Ferrari'
>>> ma_voiture_sport.roues
4
```

On remarque tout d'abord que l'attribut roues a bien été hérité. Ensuite on remarque que la méthode init a écrasé la méthode de la classe Voiture . On parle alors de surcharge de méthode.

Polymorphisme / surcharge de méthode

Comme nous l'avons vu plus haut si une classe hérite d'une autre classe, elle hérite les méthodes de son parent .

Exemple:

class006.py

```
# coding: utf-8

class Voiture:

    roues = 4
    moteur = 1

    def __init__(self):
        self.nom = "A déterminer"

    def allumer(self):
        print "La voiture démarre"

class VoitureSport(Voiture):
```

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

    ma_voiture_sport = VoitureSport()
    ma_voiture_sport.allumer()
```

Le résultat:

```
  La voiture démarre
```

Il est cependant possible d' écraser la méthode de la classe parente en la redéfinissant. On parle alors de surcharger une méthode .

class007.py

```
# coding: utf-8

class Voiture:

    roues = 4
    moteur = 1

    def __init__(self):
        self.nom = "A déterminer"

    def allumer(self):
        print "La voiture démarre"

class VoitureSport(Voiture):

    def __init__(self):
        self.nom = "Ferrari"

    def allumer(self):
        print "La voiture de sport démarre"

    ma_voiture_sport = VoitureSport()
    ma_voiture_sport.allumer()
```

Le résultat:

```
  La voiture de sport démarre
```

Enfin dernier point intéressant: il est possible d'appeler la méthode du parent puis de faire la spécificité de la méthode. On peut d'ailleurs appeler n'importe quelle autre méthode.

class008.py

```
# coding: utf-8

class Voiture:

    roues = 4
    moteur = 1

    def __init__(self):
        self.nom = "A déterminer"

    def allumer(self):
        print "La voiture démarre"

class VoitureSport(Voiture):

    def __init__(self):
        self.nom = "Ferrari"

    def allumer(self):
        Voiture.allumer(self)
        print "La voiture de sport démarre"

    ma_voiture_sport = VoitureSport()
    ma_voiture_sport.allumer()
```

Le résultat:

```
La voiture démarre
La voiture de sport démarre
```

Les classes Voiture et VoitureSport possédent donc chacune une méthode de même nom mais ces méthodes n'éffectuent pas les mêmes tâches. On parle dans ce cas de polymorphisme . Conventions

Prenez l'habitude de nommer votre classe uniquement avec des caractères alphanumériques et commençant par une majuscule. Et à l'inverse l'instance peut être nommée sans majuscule.

```
voiture_sport = VoitureSport()
```