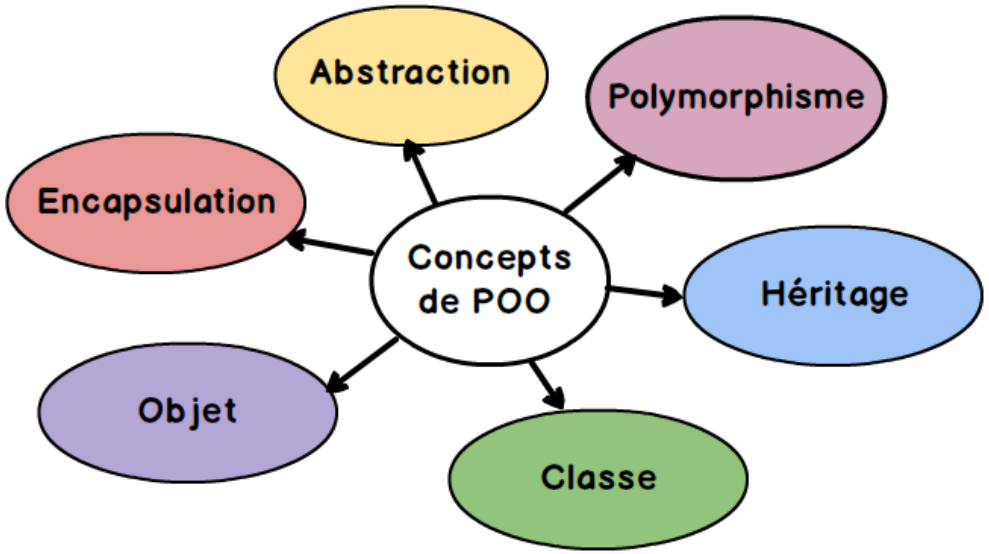
Chapitre 3: POO en Python

Principes de la POO



- L'encapsulation masque les détails d'implémentation d'une classe à d'autres objets.
- L'héritage est un moyen de former de nouvelles classes en utilisant des classes déjà définies.
- Le polymorphisme est le processus d'utilisation d'un opérateur ou d'une fonction de différentes manières pour différentes entrées de données.
- L'abstraction simplifie la réalité complexe en modélisant des classes appropriées au problème.

Classes et objets en Python

Créer une classe

Pour créer une classe, utilisez le mot-clé **class**:

Exemple:

Créez une classe nommée Voiture, avec une propriété nommée ‘vitesse’:

```
class Voiture:
  vitesse = 100
```

Créer un objet

Maintenant, nous pouvons utiliser la classe nommée Voiture pour créer des objets.

Exemple:

Créez un objet nommé ‘renault’ et affichez la valeur de la vitesse:

```
renault = Voiture()
print(renault.vitesse)
```

Classes et objets en Python

Les attributs de classe

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

```
renault= Voiture()
print(renault.vitesse)
```

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

```
renault.model = "Megane"
```

Et le lire ainsi:

```
print(renault.model)
```

Classes et objets en Python

La fonction __init__()

Les exemples vus auparavant sont des classes et des objets dans leur forme la plus simple et ne sont pas vraiment utiles dans des vrais applications.
Utilisez la fonction **__init__()** pour affecter des valeurs aux propriétés de l'objet ou à d'autres opérations nécessaires à la création de l'objet.

```
class Voiture:
  def __init__(self, model, vitesse):
    self.model = model
    self.vitesse = vitesse

renault = Voiture("Mégane", 100)
print(renault.model)
print(renault.vitesse)
```

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

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

Classes et objets en Python

Méthodes d'objet

Les objets peuvent également contenir des méthodes. Les méthodes dans les objets sont des fonctions qui appartiennent à l'objet.
Créons une méthode dans la classe Voiture:

```
class Voiture:
    def __init__(self, model, vitesse):
        self.model = model
        self.vitesse = vitesse
    def accelerer(self):
        # ajoute 10 miles par heure à la vitesse actuelle
        self.vitesse = self.vitesse + 10;
renault = Voiture("Mégane", 100)
renault.accelerer()
print("La vitesse actuelle: " + str(renault.vitesse))
```

Classes et objets en Python

Modifier les propriétés d'un objet

Vous pouvez modifier les propriétés d'un objet comme celui-ci:

```
renault.vitesse = 180
```

Supprimer les propriétés d'un objet

Vous pouvez supprimer des propriétés sur des objets à l'aide du mot clé del, l'exemple suivant supprime la propriété 'vitesse' de l'objet renault:

```
del renault.vitesse
```

L'instruction passe

la définition d'une classe ne peut pas être vide, mais si pour une raison quelconque vous avez une définition de classe sans contenu, insérez l'instruction pass pour éviter d'obtenir une erreur.

```
class Voiture:
  pass
```

Classes et objets en Python

Les propriétés

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 accéder et modifier 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.

Classes et objets en Python

Les propriétés

```
class Voiture:
    def __init__(self, model, vitesse):
        self._model = model
        self._vitesse = vitesse
    def _get_vitesse(self):
         print ("Récupération de la vitesse")
         return self._vitesse
    def _set_vitesse(self,v):
         print ("Changement de la vitesse")
         self._vitesse=v
    vitesse = property(_get_vitesse, _set_vitesse)

renault = Voiture("Mégane", 100)
renault.vitesse = 140
print(renault.vitesse)
```

Quand on changera la vitesse, un message apparaîtra. En soi cela n'apporte rien mais au lieu de faire un simple **print**

Classes et objets en Python

Les propriétés

Il existe une autre syntaxe en passant par des décorateurs:

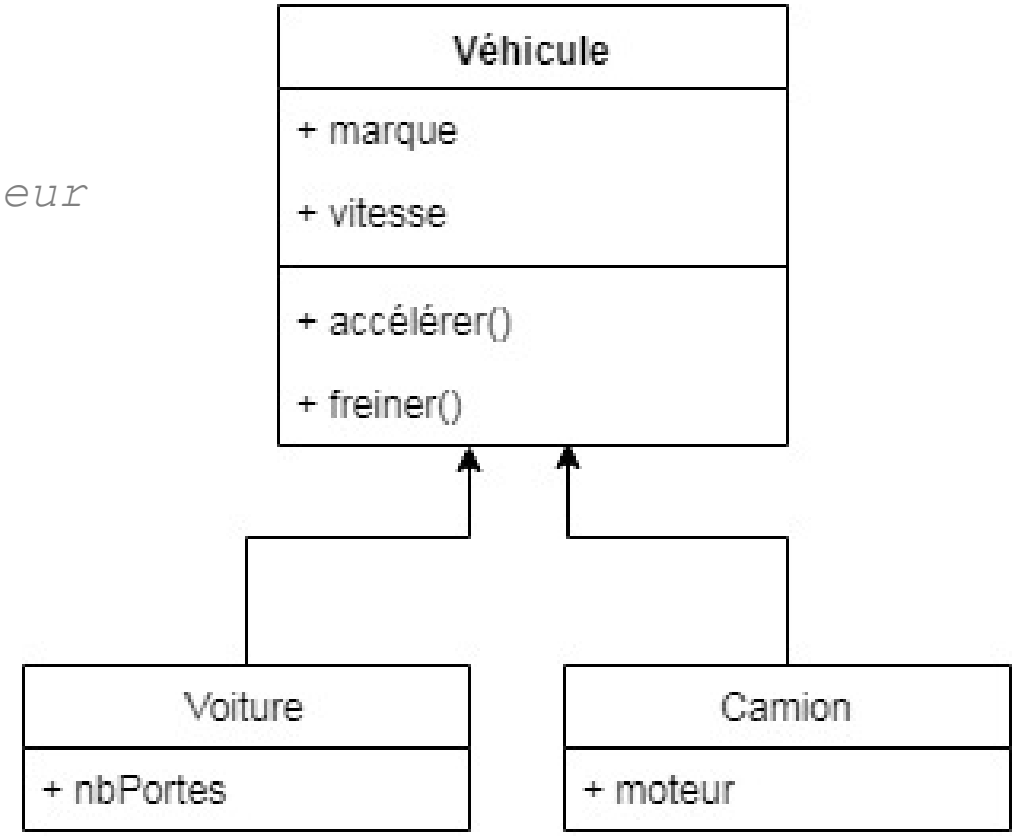
```
class Voiture:
    def __init__(self, model, vitesse):
        self._model = model
        self._vitesse = vitesse
    @property
    def vitesse(self):
         print ("Récupération de la vitesse")
         return self._vitesse
    @vitesse.setter
    def vitesse(self,v):
         print ("Changement de la vitesse")
         self._vitesse=v

renault = Voiture("Mégane", 100)
renault.vitesse = 140
print(renault.vitesse)
```

Les classes et l'héritage en Python

Exemple d'héritage simple

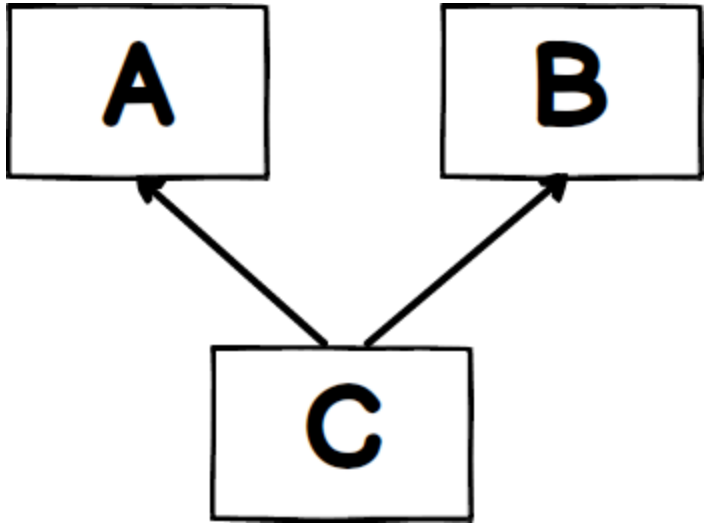
```
class Vehicule(object):
    def __init__(self, marque, vitesse): #Constructeur
        self.marque = marque
        self.vitesse = vitesse
    def getMarque(self): # Pour récupérer la marque
        return self.marque
        # Pour vérifier si cette vehicule est une
voiture
    def estUneVoiture(self):
        return False
# Sous classe
class Voiture(Vehicule):
    def estUneVoiture(self):
        return True
# code de test
vh = Vehicule("Volvo", 100)  # Un objet de vehicule
print(vh.getMarque(), vh.estUneVoiture())
vt = Voiture("Mercedes", 300)  # Un objet de voiture
print(vt.getMarque(), vt.estUneVoiture())
```



On remarque tout d'abord que les attributs vitesse et model ont bien été hérité. Ensuite on remarque que la méthode **estUneVoiture** a écrasé la méthode de la classe **Vehicule**. On parle alors de surcharge de méthode.

Les classes et l'héritage en Python

Exemple d'héritage multiple



Contrairement à Java, Python prend en charge l'héritage multiple. Nous spécifions toutes les classes parentes sous forme de liste séparée par des virgules entre parenthèses.

```
class A(object):
    def __init__(self):
        self.str1 = "Hello"
        print("A")
class B(object):
    def __init__(self):
        self.str2 = "World"
        print("B")
```

```
class C(A, B):
    def __init__(self):
        # Appel le constructeur de la classe A et B
        A.__init__(self)
        B.__init__(self)
        print("C")
    def afficherMsg(self):
        print(self.str1, self.str2)
c = C()
c.afficherMsg()
```

Polymorphisme / surcharge de méthode

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

Exemple

```
class Vehicule:
    def __init__(self): #Constructeur
        self.marque = "a déterminer"
        self.vitesse = 100
    def demarrer(self):
        print("la voiture démarre")

# Sous classe
class Voiture(Vehicule):
     def __init__(self): #Constructeur
        self.marque = "Renault"
        self.vitesse = 120

# code de test
vt = Voiture ()  # Un objet de Voiture
vt. demarrer()
vt = Voiture ()  # Un objet de Vehicule
vt. demarrer()
```

Polymorphisme / surcharge de méthode

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

Exemple

```
class Vehicule:
    def __init__(self): #Constructeur
        self.marque = "a déterminer"
        self.vitesse = 100
    def demarrer(self):
        print("la vehicule démarre")

# Sous classe
class Voiture(Vehicule):
     def __init__(self): #Constructeur
        self.marque = "Renault"
        self.vitesse = 120
    def demarrer(self):
        print("la Voiture démarre")
# code de test
vt = Voiture ()  # Un objet de Voiture
vt. demarrer()
vt = Voiture ()  # Un objet de Vehicule
vt. demarrer()
```

Polymorphisme / surcharge de méthode

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.

Exemple

```
class Vehicule:
    def __init__(self): #Constructeur
        self.marque = "a déterminer"
        self.vitesse = 100
    def demarrer(self):
        print("la vehicule démarre")
# Sous classe
class Voiture(Vehicule):
     def __init__(self): #Constructeur
        self.marque = "Renault"
        self.vitesse = 120
    def demarrer(self):
        Vehicule. demarrer(self)
        print("la Voiture démarre")
# code de test
vt = Voiture ()  # Un objet de Voiture
vt. demarrer()
vh = Vehicule()  # Un objet de Vehicule
vh. demarrer()
```

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

Exercice

1.Créer une **classe Calcul** ayant un **constructeur par défaut** (sans paramètres) permettant d'effectuer différents calculs sur les nombres entiers.
2.Créer au sein de la classe Calcul une **méthode** nommée **Factorielle()** qui permet de calculer le factorielle d'un entier. Tester la méthode en faisant une instanciation sur la classe.
3.Créer au sein de la classe Calcul une **méthode** nommée **Somme()** permettant de calculer la **somme des n premiers entiers: 1 + 2 + 3 + .. + n.** Tester la méthode.
4.Créer au sein de la classe Calcul une **méthode** nommée **testPrim()** permettant de tester la **primalité d'un entier** donné. Tester la méthode.
5.Créer une **méthode tableMult()** qui crée et affiche la table de multiplication d'un entier donné.
6.Créer une **méthode listDiv()** qui récupère tous **les diviseurs d'un entier** donné sur une **liste Ldiv**.