Programmation Orientée Objet en Python

#3 Polymorphisme

par David Albert

2023

Table des matières

01 Héritage
Héritage. Classe abstraites. Interfaces.

02 Polymorphisme
Définition. Spécialisation. Surcharge de méthodes.

03 Héritage multiple
Héritage multiple. Ordre d'héritage.

04 Méthodes de classes
Attribut / méthodes de classes. Décorateur ***@classmethod***. Mot-clé **cls**.

01

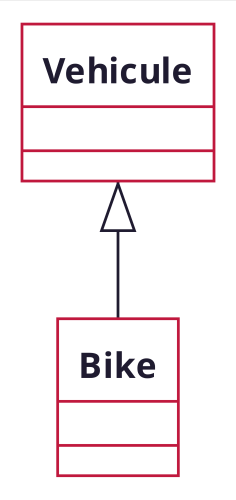
Héritage

Héritage

En POO, l'**héritage** est le concept qui permet de créer une nouvelle classe à partir d'une classe existante.

Syntaxe UML

La classe `Bike` hérite de la classe `Vehicule`.
La classe **fille** est `Bike`.
La classe **parente** est `Vehicule`.



Syntaxe python

```python
class Vehicule: # ici on définit la classe mère
    def __init__(self, wheels, brand):
        self._brand = brand
        self._wheels = wheels

    def accelerate(self):
        print("Go !")

class Bike(Vehicule): # class fille
    def __init__(self):
        super().__init__(2, "Canyon")
```

En héritant de `Vehicule`, la classe `Bike` hérite de ses méthodes.

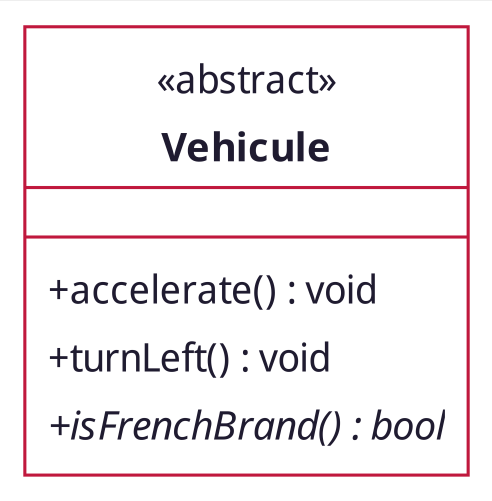
```python
b = Bike()
b.accelerate()
# Go !
```

Classe abstraite

Définition

Une **classe abstraite** est une classe qui comprend **au moins** une méthode **non implémentée**.

Syntaxe UML



> Les méthodes abstraites sont écrites en *italique*. A la main, on souligne.

Intérêt

- Implémenter certains opérationes communes à un groupe d'objets malgré que le concept soit encore *abstrait*

Syntaxe python

```python
class Vehicule:
    def accelerate(self):
        print("Go !")

    def turnLeft(self):
        print("Go left !")

    def isFrenchBrand(self):
        raise NotImplementedError("The method is abstract")
```

Interface

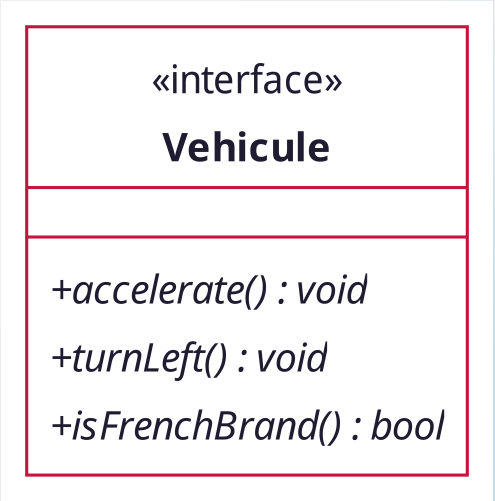
Définition

Une **interface** est une classe abstraite particulière. Elle ne contient **aucun attributs** et ses méthodes ne sont **pas implémentées**.

Intérêts

- Définir les opérations sans préciser leur implémentation
- Préciser les conditions et les effets de l'invocation des opérations

Syntaxe UML



Syntaxe python

```python
class Vehicule:
    def accelerate(self):
        raise NotImplementedError("The method is abstract")

    def turnLeft(self):
        raise NotImplementedError("The method is abstract")

    def isFrenchBrand(self):
        raise NotImplementedError("The method is abstract")
```

Les méthodes abstraites renvoient une erreur.

! Les **classes abstraites** et les **interfaces** ne seront **jamais instanciées** directement.

02

Polymorphisme

Polymorphisme

En POO, le **polymorphisme** est le concept qui permet de **modifier le comportement** d'une classe fille par rapport à sa classe mère.

Cela permet d'utiliser l'héritage comme:

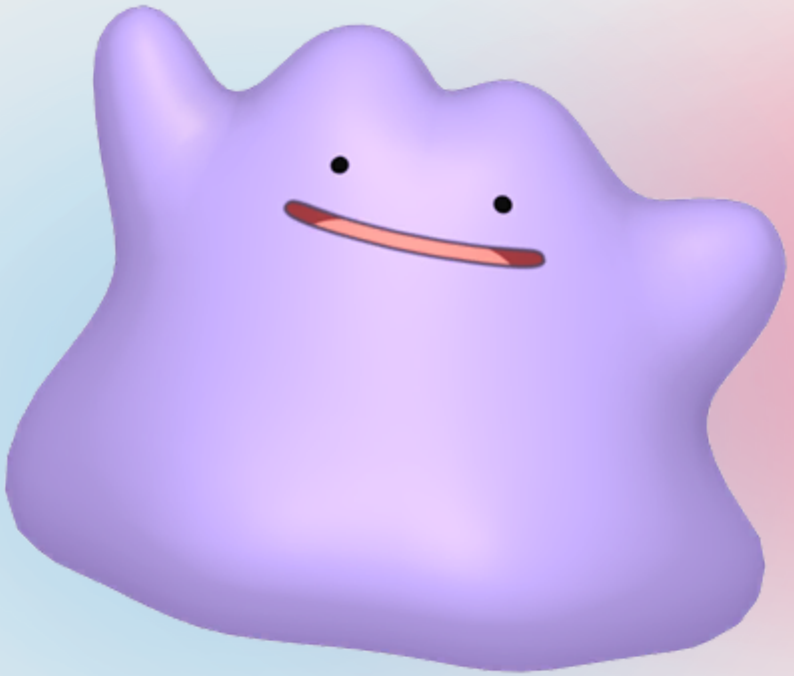
- mécanisme de **spécialisation** d'un concept.
- mécanisme d'**extension du système**.

Bonne pratique

1. On définit une **interface commune** à une famille d'objets (la classe de base).
2. On écrit les **détails d'implémentations** des classes spécialisées.

Polymorphisme

"qui peut prendre plusieurs formes"

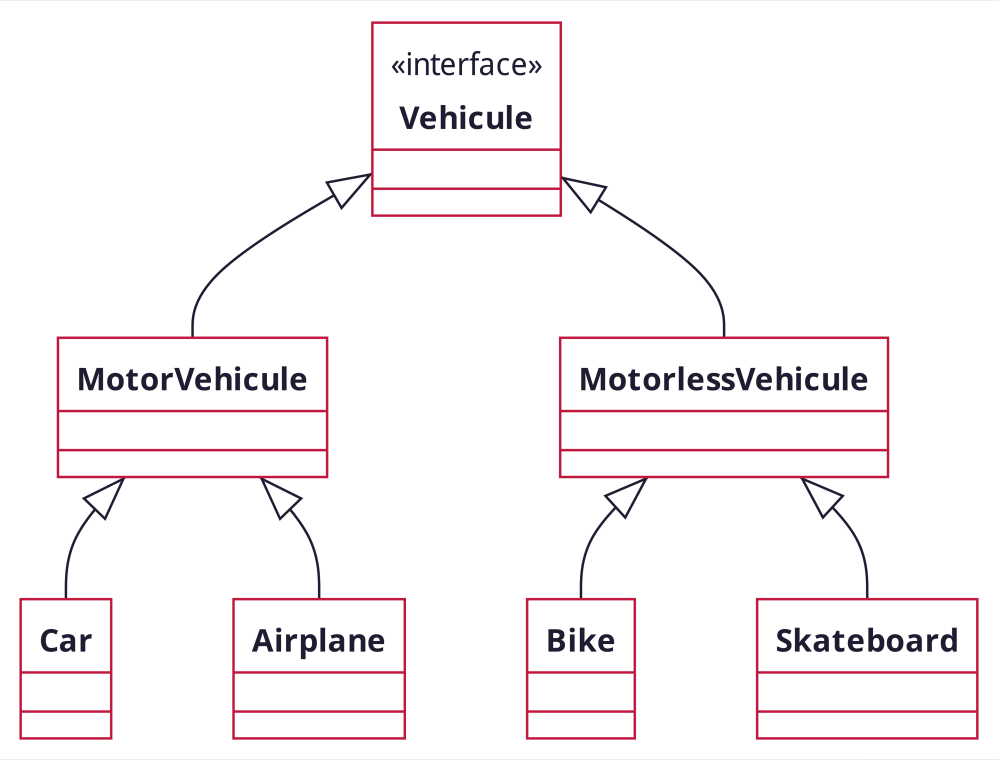


Surcharger une méthode

Mot-clé super

TO DO

Polymorphisme en pratique (1)



On surcharge la méthode `accelerate`.

```python
class Vehicule:
    def accelerate(self):
        raise NotImplementedError("The method is abstract")

class MotorlessVehicule(Vehicule):
    def accelerate(self):
        print("Go cleanly !")

class Bike(MotorlessVehicule):
    def accelerate(self):
        print("Go cleanly by bike!")

class Bike(MotorlessVehicule):
    def accelerate(self):
        print("Go cleanly by skate!")
```

La méthode `accelerate` n'est pas la même pour un vélo et un skateboard.

Polymorphisme en pratique (2)

Nous pouvons appeler la méthode `accelerate` d'un objet sans nous soucier son type intrinsèque.

```
vehicules : list[Vehicule] = []
vehicules.append(Car())
vehicules.append(Skateboard())

for v in vehicules:
    v.accelerate()

# OUTPUT
# Go cleanly by bike!
# Go cleanly by skate!
```

03

Héritage multiple

Héritage multiple

TO DO

Ordre d'héritage

Afin de pouvoir déboguer lors d'erreurs avec l'héritage multiple, il est possible de connaître l'ordre d'héritage. Pour cela, on utilise la méthode `__mro__`.

```
class A():
    pass

class B():
    pass

class C(A, B):
    pass

if __name__ == '__main__':
    # Attention, __mro__ est un attribut de classe.
    # Il doit donc etre recupere depuis la classe
    print(C.__mro__)
```

L'exécution de ce code renvoit

```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <type 'object'>)
```

Order d'héritage

Exercice

```
class A():
    pass

class B():
    pass

class C(A, B):
    pass

class D(B,A):
    pass

class E(D,A):
    pass

class F(C, D, B):
    pass
```

Question 1 : Qu'affiche ce programme ?

```
print(C.__mro__)
```

Question 2 : Qu'affiche ce programme ?

```
print(D.__mro__)
```

Question 3 : Qu'affiche ce programme ?

```
print(E.__mro__)
```

Question 4 : Qu'affiche ce programme ?

```
print(G.__mro__)
```

Order d'héritage

Exercice

```
class A():
    pass

class B():
    pass

class C(A, B):
    pass

class D(B,A):
    pass

class E(D,A):
    pass

class F(E, D):
    pass

class G(F, B):
    pass
```

Réponse 1 :

```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
```

Réponse 2 :

```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

Réponse 3 :

```
(<class '__main__.E'>, <class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>,
<class 'object'>)
```

Réponse 4 :

```
(<class '__main__.G'>, <class '__main__.F'>, <class '__main__.E'>, <class '__main__.D'>,
<class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

04

Méthodes de classes

Définition

Avant propos

Jusqu'alors nous avons utilisé des **méthodes d'instances**. Celles-ci sont propres à un objet et manipulent les données (**= attributs d'instances**) de ce dernier.

Méthodes de classes

Maintenant, nous allons voir comment utiliser des **méthodes de classes**. Celles-ci manipulent des données communes à toutes les instances d'une même classe (**= les attributs de classes**). Les méthodes de classes sont définies grâce au décorateur `@classmethod` et prennent en 1er argument le paramètre `cls` (une référence vers la classe).

Exemples d'usages

- Stocker des constantes de classe
- Garder un compteur du nombre d'instances
- Créer un constructeur alternatif
- Profiling (nombre de passage et temps dans passé dans chaque fonction)

Exemple

```python
class Counter:
    count = 0                   # attribut de classe

    def __init__(self, name):
        self.name = name        # attribut d'instance

    @classmethod
    def add(cls, num):          # méthode de classe
        cls.count += num


if __name__ == '__main__':
    c1 = Counter("Counter #1")
    c2 = Counter("Counter #2")

    print(Counter.count, c1.count, c2.count)
    # output : 0 0 0

    Counter.add(5)
    print(Counter.count, c1.count, c2.count)
    # output : 5 5 5

    c1.add(5)
    print(Counter.count, c1.count, c2.count)
    # output : 10 10 10

    c2.add(-10)
    print(Counter.count, c1.count, c2.count)
    # output : 0 0 0
```