5 : Héritage simple

L'**héritage** est un outil très important en programmation orientée objet.

Il permet de construire des hiérarchies de classes, de la plus générale à la plus spécialisée.

C'est un concept essentiel à la construction de librairies (d'algèbre linéaire, par exemple).

5.1 : Principe de l'héritage

Il s'agit de créer des classes qui sont **des cas particuliers** de classes précédemment construites, et qui héritent de leurs propriétés : **attributs**, **méthodes**, etc.

On parle de **classe mère ou parent** pour la classe la plus générale et de classe **fille ou enfant** pour la classe la plus spécifique.

Une classe mère peut avoir plusieurs classe enfant et vice versa.

Exemple:

- La classe des nombres complexes (classe mère) et celle des numbres complexes de module 1 (classe fille). Les objets de la seconde sont des cas particuliers des objets de la première.
- La classe véhicule (classe mère) et la classe voiture (classe fille).

Chaque objet de la classe fille est un objet de la classe mère mais pas l'inverse.

Pour les objets mathématiques, une classe fille est souvent un sous-ensemble ou un sous-espace de la classe mère.

- L'héritage peut nous aider à représenter des objets présentant des différences et des similitudes dans leur fonctionnement. La classe fille est une spécialisation de la classe mère.

- L'héritage est également un moyen de réutiliser facilement le code existant. Si nous avons déjà une classe, nous pouvons créer une sous-classe qui hérite des méthodes de la première, et modifier seulement ce qui est différent.

Dans un premier temps, on va voir :

- comment construire une classe fille,
- comment la classe fille à accès aux données de la classe mère,
- comment ajouter et modifier des comportements dans la classe fille.

5.2 : Classe fille et héritage

In [3]:

```
class Véhicule:
    def __init__(self):
        print('Initialisation du véhicule')
        self.__vitesse = 0
        self.est_allume = False
        print('vitesse:', self.__vitesse)

    def démarrer(self):
        self.est_allume = True
        print('Le véhicule démarre')

    def get_vitesse(self):
        return self.__vitesse

    def set_vitesse(self, v):
        self.__vitesse = v
```

In [4]:

```
class VéhiculeTerrestre (Véhicule):
    def rouler(self):
        if self.est_allume:
            print('Le véhicule roule')
        else:
            print('Impossible, le véhicule est éteint')
        print('vitesse:', self.__vitesse)

class VéhiculeAérien (Véhicule):
    def décoller(self):
        if self.est_allume:
            print('Le véhicule décolle')
        else:
            print('Impossible, le véhicule est éteint')
        print('vitesse:', self.__vitesse)
```

In [5]:

```
v1 = Véhicule()
v2 = VéhiculeTerrestre()
v3 = VéhiculeAérien()
```

```
Initialisation du véhicule
vitesse: 0
Initialisation du véhicule
vitesse: 0
Initialisation du véhicule
vitesse: 0
```

Remarques :

- Pour faire hériter une classe d'une classe mère, on écrit `classe_fille(classe_mère)` au moment de la déclaration.
- On a deux appels de `__init()__` de `Véhicule` . Python, n'ayant pas de `__init()__` dans la classe fille, appelle celui de la classe mère. Il s'agit d'héritage.
- Les attributs de la **classe mère** sont automatiquement des attributs de la **classe fille**. Un véhicule terrestre a donc les attributs: `__vitesse` et `est_allumé` .

On regardera plus en détails comment les méthodes, le constructeur s'hérite en classe fille et classe mère.

5.3 : Accès à la classe parente

On souhaite maintenant regarder comment accéder aux **attributs hérités** de la classe mère:

In [6]:

```
v = VéhiculeTerrestre()
print("v est-il allumé ? ", v.est_allume)
```

```
Initialisation du véhicule
vitesse: 0
v est-il allumé ?  False
```

5.3.1 : Attributs privés et publics

In [7]:
```
print("vitesse de v : ", v.__vitesse)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/tmp/ipykernel_152199/2477179060.py in <module>
----> 1 print("vitesse de v : ", v.__vitesse)

AttributeError: 'VéhiculeTerrestre' object has no attribute '__vitesse'
```

Pas de problème dans le premier cas, mais le second cas produit une erreur. **Pourquoi ?**

- `est_allume` est un attribut public de la classe mère
- `__vitesse` est attribut privé de la classe mère

Ce résultat est logique. En cas d'encapsulation, on peut pas avoir accès un attribut privé en dehors de la classe. Il faut passer par des accesseurs et mutateurs.

In [8]:

```
v.démarrer()
v.rouler()
```

```
Le véhicule démarre
Le véhicule roule

-----------------------------------------------------------------------
-
AttributeError                            Traceback (most recent call las
t)
/tmp/ipykernel_152199/2345461210.py in <module>
      1 v.démarrer()
----> 2 v.rouler()

/tmp/ipykernel_152199/3116105842.py in rouler(self)
      5         else:
      6             print('Impossible, le véhicule est éteint')
----> 7         print('vitesse:', self.__vitesse)
      8
      9 class VéhiculeAérien (Véhicule):

AttributeError: 'VéhiculeTerrestre' object has no attribute '_VéhiculeTerr
estre__vitesse'
```

Ici, le résultat est plus étonnant. En effet, précédemment, on pouvait manipuler une variable privée dans une méthode d'une classe.

Ici, `__vitesse` est une attribut de la classe véhicule. Par héritage il est aussi un attribut de `VéhiculeTerrestre` mais pas directement.

Par conséquent, l'encapsulation s'applique aussi dans la classe fille. Un attribut privé issu de la classe mère n'est pas accessible directement dans la classe fille.

Pour s'en sortir :

On peut utiliser les accesseurs/mutateurs de la classe mère qui sont hérités dans la classe fille. Les méthodes étant toujours publiques, il n'y aura pas de problème.

In [9]:

```
class VéhiculeTerrestre2 (Véhicule):
    def rouler(self):
        if self.est_allume:
            print('Le véhicule roule')
        else:
            print('Impossible, le véhicule est éteint')
        print('Vitesse:', self.get_vitesse())

v = VéhiculeTerrestre2()
```

```
Initialisation du véhicule
vitesse: 0
```

In [10]:

```
v.démarrer()
v.rouler()
```

```
Le véhicule démarre
Le véhicule roule
Vitesse: 0
```

5.3.2 : Attributs protégés

En programmation objet il existe un troisième type de visibilité après **publique** et **privée** : il s'agit de la visibilité **protégée**.

L'idée est que les attributs protégés seront privés pour les classes ou fonctions extérieures, mais publics pour les classes filles.

En pratique, le `Python` ne dispose pas de la visibilité **protégé**, mais propose une approche par convention pour différencier ce qui fait partie de l'implémentation interne de ce qui constitue l'interface publique d'un objet.

On notera tout élément protégé d'une classe, méthode ou attribut, par le préfixe `_` .

5.4 : Enrichissement et surcharge

Une classe **fille** peut :

- enrichir la classe mère (la spécialiser) en ajoutant des méthodes et des attributs ;
- modifier des méthodes, opérateurs et constructeurs pour la surcharger.

In [11]:

```
class VéhiculeTerrestre3 (Véhicule):
    def rouler(self):
        if self.est_allume:
            print('Le véhicule roule')
        else:
            print('Impossible, le véhicule est éteint')
        print('Vitesse:', self.get_vitesse())

    def set_nb_roues(self, n):
        self.nb_roues = n

    def get_nb_roues(self):
        return self.nb_roues
```

In [17]:

```
v = VéhiculeTerrestre3()
v.set_nb_roues(3)
print("Vitesse: ", v.get_vitesse(), " ;  nombre de roues: ", v.get_nb_roues())
```

```
Initialisation du véhicule
vitesse: 0
Vitesse:  0  ;  nombre de roues:  3
```

On a ajouté deux méthodes:

- une qui crée un nouvel attribut `nb_roues` ,
- un accesseur associé.

On affiche ensuite les deux attributs, `__vitesse` et `nb_roues` , et tout se passe comme prévu.

On voit qu'on peut **spécialiser la classe véhicule** avec de nouvelles méthodes et de nouveaux attributs.

Maintenant, que se passe-t-il si on souhaite introduire un nouvel attribut dès le constructeur ?

Solution naturelle:

In [13]:

```
class VéhiculeTerrestre4(Véhicule):

    def __init__(self):
        print('Initialisation du véhicule terrestre')
        self.nb_roues = 0

    def rouler(self):
        if self.est_allume:
            print('Le véhicule roule')
        else:
            print('Impossible, le véhicule est éteint')
        print('Vitesse:', self.get_vitesse())

    def donner_nb_roues(self, n):
        self.nb_roues = n

    def get_nb_roues(self):
        return self.nb_roues
```

In [14]:

```
v = VéhiculeTerrestre4()
v.get_vitesse()
```

```
Initialisation du véhicule terrestre

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/tmp/ipykernel_152199/1644738296.py in <module>
      1 v = VéhiculeTerrestre4()
----> 2 v.get_vitesse()

/tmp/ipykernel_152199/504275213.py in get_vitesse(self)
     11
     12     def get_vitesse(self):
---> 13         return self.__vitesse
     14
     15     def set_vitesse(self, v):

AttributeError: 'VéhiculeTerrestre4' object has no attribute '_Véhicule__vitesse'
```

Que se passe-t-il ?

L'accesseur de l'attribut `__vitesse` ne marche plus. Avant il marchait. Que s'est-il passé ?

On voit dans l'affichage qu'on ne passe pas le `__init__()` de la classe mère mais uniquement celui de la classe fille.

Par conséquent l'attribut `__vitesse` n'est pas créé (car il est normalement créé dans le `__init__()` de la classe mère).

Solution : une fonction *built-in* nommée `super()` permet de régler le problème.

In [15]:

```
class VéhiculeTerrestre5(Véhicule):

    def __init__(self, n):
        print('Initialisation du véhicule terrestre')
        super().__init__()
        self.nb_roue = n

    def rouler(self):
        if self.est_allumé:
            print('Le véhicule roule')
        else:
            print('Impossible, le véhicule est éteint')
        print('Vitesse:', self.get_vitesse())

    def démarrer(self):
        super().démarrer()
        print("Démarrage du véhicule terrestre")

    def get_nb_roues(self):
        return self.nb_roues
```

In [18]:

```
v = VéhiculeTerrestre5(3)
v.get_vitesse()
```

```
Initialisation du véhicule terrestre
Initialisation du véhicule
vitesse: 0
```

Out[18]:

```
0
```

On voit que :

- Avec `super()` on appelle le `__init__()` de la classe mère puis on ajoute/initialise l'attribut spécifique à la classe fille. On crée les attributs associés à la mère, puis ceux associés à la classe fille.
- On peut aussi modifier la signature du `__init__()` .

In [19]:
```
v.démarrer()
```

```
Le véhicule démarre
Démarrage du véhicule terrestre
```

On voit a travers cet exemple qu'on peut re-définir une fonction qui était déjà définie dans la classe mère.

On peut re-définir/surcharger : les méthodes, le `__init__()` , les opérateurs, etc.

5.5 : Exemple mathématique

On propose un exemple complet d'héritage en mathématiques : les complexes et les complexes de module 1.

In [20]:

```
class complexe:
    def __init__(self, x, y):
        self.reel = x
        self.imag = y

    def module(self):
        return np.sqrt(self.reel**2 + self.imag**2)

    def __str__(self):
        return "partie réelle : " + str(self.reel) + " ; partie imaginaire : " + str(sel
```

```
    def __lt__(self, other):
        mini = other
        if self.module() < other.module():
            mini = self
        return mini

    def __add__(self, other):
        res = complexe(0, 0)
        res.reel = self.reel + other.reel
        res.imag = self.imag + other.imag
        return res

    def __mul__(self, other):
        res = complexe(0, 0)
        res.reel = self.reel * other.reel - self.imag * other.imag
        res.imag = self.imag * other.reel + self.reel * other.imag
        return res
```

Réécriture des complexes :

$$x + iy = re^{i\theta}$$

Les complexes de module 1 sont donnés par $r = 1$, et donc définis par un unique paramètre θ, qui représente l'argument.

In [21]:

```
class complexe_un(complexe):

    def __init__(self, angle):
        super().__init__(np.cos(angle), np.sin(angle))

    def __angle(self):
        r = super().module()
        angle = 0.0
        if self.imag > 0.0:
            angle = np.arccos(self.reel / r)
        else:
            angle = -np.arccos(self.reel / r)
        return angle

    def __mul__(self, other):
        angle1 = self.__angle()
        angle2 = other.__angle()
        res = complexe_un(angle1 + angle2)
        return res
```

La multiplication de nombre complexe de module est un complexe de module un, mais ce n'est pas vrai pour l'addition !

On redéfinit donc la multiplication, pas l'addition.

In [31]:

```
z1 = complexe(np.cos(20), np.sin(20))
z2 = complexe(np.cos(30), np.sin(30))

print("multiplication (complexe) : ", type(z1), " ", type(z2), " ", type(z1 + z2))

z3 = complexe_un(20)
z4 = complexe_un(30)

print("addition  (complexe_un) : ", type(z3), " ", type(z4), " ", type(z3 + z4))

print("multiplication (complexe_un) : ", type(z3), " ", type(z4), " ", type(z3 * z4))
```

```
multiplication (complexe) :  <class '__main__.complexe'>   <class '__main_
_.complexe'>   <class '__main__.complexe'>
addition  (complexe_un) :  <class '__main__.complexe_un'>   <class '__main
__.complexe_un'>   <class '__main__.complexe'>
multiplication (complexe_un) :  <class '__main__.complexe_un'>   <class '_
_main__.complexe_un'>   <class '__main__.complexe_un'>
```

En redéfinissant la multiplication on assure au niveau du type et pas au niveau du résultat que le résultat est un **complexe de module un**.

Si on fait pas cela , on passe par le `*` classique, puis on teste le module : on est alors sensibles aux erreurs d'arrondi.

5.6 : TD

5.6.1 : Exercice 1 - Modélisation de formes géométriques

On s'intéresse la modélisation de formes géométriques centrées en zéro.

1) Proposer deux hiérarchies de formes géométriques que l'on pourrait modéliser par de l'héritage.

2) Coder ces hiérarchies en modifiant les constructeurs et en utilisant ceux des classes mère.

3) Écrire une fonction `aire` dans la mère et la surcharger à chaque fois.

4) Coder accesseurs et mutateurs dans la classe mère et donner des exemples d"utilisation dans la classe fille.

5.6.2 : Exercice 2 - Matrices

En physique ou en biologie, on utilise beaucoup de matrice creuses. On va coder une hiérarchie de classes pour limiter le temps de calcul (la limitation du stockage sera traitée ultérieurement).

On se donne cinq types de matrices : dense, triangulaire inférieure et supérieure, tri-diagonale, et diagonale.

1) Comment organiser l'héritage ?

2) Dans quelle classe coder les accesseurs et mutateurs ?

3) Quelles opérations peut-t-on surcharger, et comment, pour gagner du temps de calcul ?