

7 : Classes abstraites

La notion de **classe abstraite** est utilisée lors de l'héritage pour obliger les classes filles à :

- implémenter certaines méthodes (dites **méthodes abstraites**)
- avoir certains comportements,
- et donc à respecter une **interface**.

7.1 : Principe des classes abstraites

Une **classe abstraite** est **une classe qui ne permet pas d'instancier des objets.**

Elle ne peut servir que de modèle/d'interface pour les classes filles.

Exemple : Dans les cas des véhicules, on peut considérer

- les classes `Véhicule`, `VéhiculeTerrestre` et `VéhiculeAérien` comme **abstraites**,
- les classes `Voiture`, `Moto`, `Avion`, `Camion`, `Hydravion` comme **concrètes**.

En effet, on ne veut pas *vraiment* construire un `VéhiculeTerrestre`, ce serait trop abstrait. On veut plutôt construire un `Camion` ou une `Voiture`.

`VéhiculeTerrestre` va être utilisée pour **définir les attributs et les méthodes que devront contenir toutes les classes filles.**

Une classe abstraite est constituée de méthodes et potentiellement d'attributs.

- Des méthodes possédant une implémentation, qui peut être utilisée ou redéfinie dans les classes filles (c'est ce qu'on a fait jusqu'à présent).
- Des méthodes simplement déclarées – nom et signature – et ne possédant aucune implémentation (avec le mot-clé `pass`). **Ces méthodes devront obligatoirement être implémentées dans les classes filles.**
- Certaines méthodes peuvent avoir une implémentation mais être marquées comme abstraites, et doivent donc être implémentées par les classes filles.

Les méthodes sans implémentation ou marquées abstraites s'appellent des **méthodes abstraites** ou **méthodes virtuelles**.

7.2 : Règles de construction

Les classes abstraites ne sont pas fournies dans le Python de base. On utilise pour cela la librairie `abc` (**Abstract Base Classes**).

```
In [1]: import abc
        class ClasseAbstraite(abc.ABC):
            pass
```

```
In [2]: from abc import ABC
        class ClasseAbstraite(ABC):
            pass
```

Pour construire une classe abstraite, on la fait hériter de `abc.ABC`.

7.2.1 : Déclaration et utilisation de classes abstraites

In [3]:

```
class VéhiculeTerrestre (abc.ABC):  
    def __init__(self, nb_roues):  
        self.vitesse = 0  
        self.est_allumé = False  
        self.nb_roues = nb_roues  
        self.bon_état_roues = True  
  
    def get_vitesse(self):  
        print('la vitesse est', self.vitesse)  
  
    @abc.abstractmethod  
    def contrôle_technique(self):  
        pass  
  
    @abc.abstractmethod  
    def démarrer(self):  
        self.est_allume = True  
        print('Le véhicule terrestre démarre')
```

Remarques:

- Le décorateur `@abc.abstractmethod` permet de rendre les méthodes `démarrer` et `contrôle_technique` abstraites.
- La méthode `get_vitesse`, quant à elle, ne sera pas abstraite.
- La méthode `démarrer` est abstraite avec une implémentation, `contrôle_technique` est juste déclarée.

In [4]:

```
v = VéhiculeTerrestre()
```

```
-----  
-  
TypeError                                Traceback (most recent call las  
t)  
/tmp/ipykernel_43246/3396994234.py in <module>  
----> 1 v = VéhiculeTerrestre()  
  
TypeError: Can't instantiate abstract class VéhiculeTerrestre with abstrac  
t methods contrôle_technique, démarrer
```

On remarque qu'on ne peut pas déclarer des objets de cette classe.

L'erreur nous dit que les méthodes `démarrer` et `contrôle_technique` sont abstraites, et donc qu'on ne peut pas déclarer l'objet sans déclarer ces méthodes.

In [9]:

```
class Voiture(VéhiculeTerrestre):  
    def __init__(self, marque):  
        super().__init__(4)  
        self.marque = marque  
  
    def contrôle_technique(self):  
        print('vérification de la batterie')  
        print('vérification des 4 roues')  
        self.bon_état_roues = True  
  
    def démarrer(self):  
        super().démarrer()  
        print('démarre en utilisant le clé de contact')
```

In [10]:

```
v = Voiture("Audi")
```

Remarques :

- Le constructeur d'une classe concrète peut utiliser celui d'une classe mère abstraite.
- On voit qu'on peut re-définir `démarrer` et définir `contrôle_technique`.

In [11]:

```
class Voiture(VéhiculeTerrestre):  
    def __init__(self, marque):  
        super().__init__(4)  
        self.marque = marque  
  
    def démarrer(self):  
        super().démarrer()  
        print('en utilisant le clé de contact')  
  
v = Voiture("Audi")
```

```
-----  
-  
TypeError                                Traceback (most recent call las  
t)  
/tmp/ipykernel_43246/2049512546.py in <module>  
      8         print('en utilisant le clé de contact')  
      9  
----> 10 v = Voiture("Audi")  
  
TypeError: Can't instantiate abstract class Voiture with abstract method c  
ontrôle_technique
```

On n'a pas défini `contrôle_technique` donc la déclaration ne fonctionne pas : les **méthodes abstraites doivent donc absolument être re-définies dans les classes filles**.

On aurait eu une erreur similaire si on n'avait pas re-défini `démarrer`.

Règles:

- En Python, une méthode abstraite doit être obligatoirement de visibilité publique ou protégée, afin que les classes filles puissent en hériter.
- Une classe dérivée d'une classe abstraite n'est pas obligée de redéfinir toutes les méthodes abstraites. **Cependant, si tel est le cas, elle devient elle-même abstraite et ne peut être instanciée.**

En gros, une **classe concrète doit re-définir toutes les méthodes abstraites.**

In [9]:

```
class Véhicule(abc.ABC):
    def __init__(self):
        self.vitesse = 0
        self.est_allume = False

    @abc.abstractmethod
    def démarrer(self):
        self.est_allume = True
        print('Le véhicule terrestre démarre')
```

In [10]:

```
class VéhiculeTerrestre(Véhicule):
    def __init__(self, nb_roues):
        self.nb_roues = nb_roues
        self.bon_état_roues = True

    @abc.abstractmethod
    def contrôle_technique(self):
        pass

v = VéhiculeTerrestre()
```

```
-----
-
TypeError                                Traceback (most recent call las
t)
/tmp/ipykernel_27702/2987492671.py in <module>
      8         pass
      9
--> 10 v = VéhiculeTerrestre()
```

TypeError: Can't instantiate abstract class VéhiculeTerrestre with abstrac
t methods contrôle_technique, démarrer

On voit que l'on n'a pas déclaré que la classe était abstraite, et pourtant elle l'est ! (voir remarque précédente)

7.2.2 : Attributs abstraits

On peut aussi vouloir créer des attributs abstraits, c'est-à-dire des attributs qui seront communs à toutes les instances et qui devront forcément avoir une valeur dans les classes filles.

Par exemple, on peut forcer un `Véhicule` à avoir un nom.

En Python, on crée un attribut abstrait comme on créerait une méthode abstraite ; on rajoute juste un décorateur en plus du `@abc.abstractmethod` : `@property`.

In [29]:

```
class Véhicule(abc.ABC):
    @property
    @abc.abstractmethod
    def nom(self):
        pass

class Voiture(Véhicule):
    nom = "Peugeot"

v = Voiture()
print("v.nom = ", v.nom)
```

```
v.nom = Peugeot
```

7.3 : Interfaces

Une **interface** est une déclaration de comportements que doit posséder les classes l'implémentant.

En Python (contrairement au Java par exemple), on passe par les classes abstraites pour définir une interface.

Pour définir ce qui ressemble le plus à une interface, on va déclarer une classe abstraite respectant les règles suivantes :

- aucun constructeur ne sera défini ou implémenté ;
- toutes les méthodes définies seront abstraites ;
- aucune des méthodes définies ne possédera une implémentation par défaut.

In [12]:

```
class VéhiculeTerrestre (abc.ABC):  
  
    @abc.abstractmethod  
    def accélérer(self):  
        pass  
  
    @abc.abstractmethod  
    def freiner(self):  
        pass  
  
class Voiture (VéhiculeTerrestre):  
    def accélérer(self):  
        print("j'appuie sur la pédale d'accélération")  
  
    def freiner(self):  
        print("j'appuie sur la pédale de frein")  
  
class Moto (VéhiculeTerrestre):  
    def accélérer(self):  
        print("je tourne la poignée droite")  
  
    def freiner(self):  
        print("je serre le levier de frein")
```

Remarque:

- `VéhiculeTerrestre` est une interface, là où `Voiture` et `Moto` sont des classes concrètes.
- Les deux méthodes abstraites de `VéhiculeTerrestre` doivent forcément être définies dans les classes `Voiture` et `Moto`.

7.4 : TD

7.4.1 : Géométrie

On propose de construire une hiérarchie de classes sur les objets géométriques.

- 1) On propose une interface `ObjetGéométrique2D` pour représenter un objet géométrique 2D général. Donner plusieurs méthodes abstraites possibles. Implémenter la classe.
- 2) Implémenter une classe `Carré` héritant de `ObjetGéométrique2D` et implémenter les méthodes abstraites héritées ainsi que le constructeur.
- 3) On propose une interface `TransformationGéométrique2D`. Donner une méthode abstraite possible.
- 4) Implémenter une classe `Translation` héritant de `TransformationGéométrique2D` et implémenter la méthode abstraite héritée ainsi que le constructeur.
- 5) Illustrer la translation sur un `Carré`.

7.4.2 : Jeu de rôle

On souhaite proposer un code pour construire un jeu de rôle.

1) On propose de créer **classe abstraite** `Personnage`. Elle contiendra comme **attributs abstraits** le nombre maximal de points de vie (`maxPV`), le nombre actuel de PV (`PV`), l'attaque (`Att`), la défense (`Def`) et la magie (`Mag`). Elle contiendra comme **méthodes abstraites** `attaque`, `sort_offensif` et `sort_soin`.

2) On propose deux classes `Mage` et `Guerrier` qui héritent de `Personnage`. Dans ces deux classes, on initialise les objets avec comme points initiaux `10 PV`, `2 Att`, `2 Def` et `1 Mag`. On pourra donner des bonus selon la classe.

3) On va maintenant implémenter les trois méthodes abstraites précédentes. Les trois prennent un personnage **J** en paramètre, et certaines prennent un adversaire **A**.

- **Attaque** : on lance un dé pour le joueur (**d1**) et un deuxième dé bonus (**d2**) ; on retranche aux PV de l'adversaire :

$$\begin{cases} \max(0, \text{Att}(J) * d1 + 0.5 * \text{Att}(J) * d2 - \text{Def}(A)) & \text{pour un guerr} \\ \max(0, \text{Att}(J) * d1 - \text{Def}(A)) & \text{pour un mage} \end{cases}$$

- **sort_offensif** : on lance un dé pour le joueur (**d1**) et un deuxième dé bonus (**d2**) ; on retranche aux PV de l'adversaire :

$$\begin{cases} \max(0, \text{Mag}(J) * d1 - \text{Def}(A)) & \text{pour un guerr} \\ \max(0, \text{Mag}(J) * d1 + 0.5 * \text{Att}(J) * d2 - \text{Def}(A)) & \text{pour un mage} \end{cases}$$

- **sort_soin** : on lance un dé pour le joueur (**d**) ; ses nouveaux PV sont alors :

$$\min(\text{maxPV}(J), \text{PV}(J) + \text{Mag}(J) * d)$$
