La programmation orientée objet

Sommaire :

Chapitre 1
Les classes

Une classe : c'est quoi ?

Pour créer un objet, il faut déjà définir à quoi il va ressembler. Pour cela, nous allons utiliser **les classes**. Le principe est assez similaire aux fonctions : on les définit une seule fois, avec différents paramètres et on peut ensuite les appeler plusieurs fois avec des paramètres différents. Elles permettent d'organiser notre code et d'éviter les répétitions.

Exemple : On va imaginer que vous êtes propriétaire d'une concession automobile. Les voitures peuvent avoir des caractéristiques communes mais qui peuvent prendre des valeurs différentes (toutes les voitures ont une couleur mais pas forcément la même).

- Quand on va créer une classe pour notre voiture, on va donc créer ce qu'on appelle des **attributs**. Dans notre exemple, ces attributs seront la vitesse, la couleur et la marque de la voiture.

- A partir de cette classe, on va pouvoir créer différents objets qu'on appelle des **instances**. Chaque instance aura les même attributs (vitesse, couleur, marque) mais avec des valeurs différentes
- On peut également ajouter à notre classe des fonctions qui vont nous permettre d'agir sur nos instances. On appelle ça une **méthode** (on connait déjà quelques méthode : `append()` , `keys()` ...). Dans le cas de notre voiture, on pourra définir plusieurs méthodes comme `avancer` , `faire_plein` ou encore `auto_destruction` .

Comment créer une classe ?

1. Pour créer une classe, on utilise l'instruction `class` suivi du nom de notre classe et de " `:` ".
2. On ajoute ensuite les attributs de la classe (ce sont ni plus ni moins que des variables).

Pour afficher un attribut, on fait un `print()` du nom de la classe suivi d'un point et du nom de l'attribut (ex : `Voiture.marque`).

In [ ]:
```
class Voiture: # On créer notre classe 'voiture'
    marque = "Lamborghini" # On créer un attribut 'marque'
    couleur = "rouge" # On créer un attribut 'couleur'

print(Voiture.marque) # On affiche la valeur de l'attribut 'marque'
print(Voiture.couleur) # On affiche la valeur de l'attribut 'couleur'
```

Comment créer des instances ?

Maintenant qu'on a vu comment créer une classe, on va pouvoir créer des instances de cette classe. Ce qui veut dire qu'on va partir de notre classe pour créer des objets qui auront les mêmes attributs que notre classe de base.

On créer une instance de la même manière qu'on crée une variable : on donne un nom a notre instance suivie du signe `=` et du nom de notre classe avec des parenthèses vides (pour l'instant).

In [ ]:
```
class Voiture:
    marque = "Lamborghini"

voiture_01 = Voiture() # On créer une instance qu'on appelle 'voiture_01'
voiture_02 = Voiture() # Même chose avec 'voiture_02'

print(voiture_01.marque) # La marque de 'voiture_01' est 'Lamborghini'
print(voiture_02.marque) # La marque de 'voiture_02' est 'Lamborghini'
```

Attributs de classe VS attributs d'instance

Attributs de classe :
Admettons qu'on veut modifier la marque de toutes les voitures. Dans ce cas, on va modifier l'attribut de notre classe directement en modifiant la marque de ma classe

`Voiture`. Pour modifier un attribut, on appelle l'attribut (comme pour l'afficher) et on lui assigne une nouvelle valeur avec le signe `=` suivie de la valeur choisie (ex : `Voiture.marque = "Porsche"`).

In [ ]:
```
Voiture.marque = "Porsche" # L'attribut 'marque' de la classe 'Voiture' a mainte

print(voiture_01.marque)
print(voiture_02.marque)
```

Attributs d'instance :
Admettons maintenant qu'on souhaite modifier la marque indépendamment pour chaque voiture. Pour ce faire, on ne va pas cette fois modifier l'attribut `marque` de la classe `Voiture` mais on va directement modifier l'attribut de l'intance choisie.

In [ ]:
```
voiture_01.marque = "Peugeot" # L'attribut 'marque' de l'instance 'voiture_01' a
voiture_02.marque = "Volkswagen" # L'attribut 'marque' de l'instance 'voiture_02

print(Voiture.marque) # 'Porsche'
print(voiture_01.marque) # 'Peugeot'
print(voiture_02.marque) # 'Volkswagen'
```

Comment initialiser une instance ?

Nous allons maintenant découvrir une méthode appellée `init` qui va permettre directement lors de la création d'une instance, de passer des paramètres à ces instance pour les initialiser et ainsi les défini un peu différemment les unes des autres. Cela nous permetterais de définir la marque de notre voiture directement lors de la création de notre instance.

Pour définir cette méthode, on la créer de la même manière qu'une fonction. On la défini avec le mot `def` et on va l'appeler `__init__` (avec 2 tirets du bas de chaque coté du mot `init`). Ce mot est réservé en Python il permet d'appeler la fonction ayant ce nom automatiquement lorsqu'on créer une instance à partir d'une classe.

```
class Voiture:
    def __init__():
        marque = "Lamborhini" # Code exécuté lors de la création d'une
instance
```

Pour pouvoir passer des paramètres lors de la création d'une instance, on peut ajouter des arguments à la fonction `__init__` qu'on définira dans cette fonction. Pour pouvoir ajouter des arguments dans notre classe, on doit ajouter un argument obligatoire qu'on appelle communément `self`. Cet argument correspond à l'intance qu'on crée. Pour créer des variables dans notres classe (des attributs), il faudra maintenant écrire le mot `self` suivit d'un point et du nom de l'attribut (ex : `self.marque = "Lamborhini"`).

```
class Voiture:
    def __init__(self, marque): # On ajoute l'argument 'marque'
        self.marque = marque # On définit la variable 'marque' qui a
pour valeur l'argument passé dans la fonction
```

Lorsqu'on créer une instance de la classe `Voiture`, on peut maintenant l'écire de cette manière : `voiture_01 = Voiture("Lamborghini")`.

In [ ]:
```python
class Voiture:
    def __init__(self, marque): # On ajoute un argument 'marque'
        self.marque = marque # On créer l'attribut 'marque' qui a pour valeur l'

voiture_01 = Voiture("Lamborghini")
voiture_02 = Voiture("Porsche")

print(voiture_01.marque)
print(voiture_02.marque)
```

Maintenant qu'on connais un peu mieux comment fonctionnent les classe, on peut faire des choses un peu plus poussées comme ajouter un compteur de voiture crées. Pour ce faire, on créer une variable `voitures_crees` qu'on initalise à `0`. Ensuite, dans le `__init__`, on incrémente la variable `voitures_crees` de `1`. Comme la fonction `__init__` s'exécute dès qu'on créer une instance de la classe, chaque création d'instance (donc de voiture) ajoutera `1` au compteur.

In [ ]:
```python
class Voiture:
    voitures_crees = 0
    def __init__(self, marque):
        Voiture.voitures_crees += 1
        self.marque = marque

voiture_01 = Voiture("Lamborghini")
voiture_02 = Voiture("Porsche")
print(Voiture.voitures_crees)
```

C'est quoi concrètement « `self` » ?

`self` est un argument qu'on peut ajouter dans les méthodes (fonctions contenues dans une classe) et qui est substitué par les instance (c'est pas grave de ne pas tout comprendre ne vous inquiétez pas). Par exemple, lorsqu'on exécute `voiture_01.afficher_marque()`, c'est comme si on faisait `Voiture.afficher_marque(voiture_01)`. D'où l'intéret d'ajouter self comme argument de la fonction `afficher_marque` car sinon Python nous retournerait une erreur disant qu'un argument n'est pas défini.

In [ ]:
```python
class Voiture:
    def __init__(self, marque):
        self.marque = marque

    def afficher_marque(self):
        print(f"La voiture est une {self.marque}")

voiture_01 = Voiture("Lamborghini")
voiture_01.afficher_marque() # Execute la méthode 'afficher_marque' pour l'insta
Voiture.afficher_marque(voiture_01) # Fait exactement la même chose que la ligne
```

Exercice 1

```python
class Voiture:
    pneus = 4
```

```
    def __init__(self, marque):
        self.marque = marque
    def afficher_marque(self):
        print(f"La voiture est une {self.marque}")

lamborghini = Voiture("Lamborghini")
```

A partir du code ci-dessus, répondez au 5 questions suivantes :

1 - Quel est le nom de la classe ?
Réponse :

2 - Quel est le nom de l'attribut de classe ?
Réponse :

3 - Quel est le nom de l'attribut d'instance ?
Réponse :

4 - Quel est le nom de la deuxième méthode de la classe ?
Réponse :

5 - Quel est le nom de l'instance de la classe ?
Réponse :

Chapitre 2
Les classes de données

Une classe de données : c'est quoi ?

Avec les classes en python, il y a beaucoup de code qui est répété lorsqu'on initialise une instance. Pour contrer ce problème, depuis la version 3.7 de Python, un module appelé `dataclasses` a été implémenté. Il permet d'avoir accès à un décorateur et à des fonctions qui vont être générées automatiquement, notamment les méthodes spéciales comme `__init__` et `__repr__` qui sont des fonctions que l'on retrouve dans la plupart des classes.

Pour la méthode `__init__` :

Plutot que de définir la fonction `__init__` de cette façon dans une classe :

In [ ]:
```
class Utilisateur:
    def __init__(self, prenom:str, nom:str):
        self.prenom = prenom
        self.nom = nom
```

On peut importer le module `dataclasses`, ajouter le décorateur `dataclass` avant la création de notre classe `Utilisateur` pour ensuite créer les attributs de notre classe de la manière suivante :

In [ ]:
```
from dataclasses import dataclass

@dataclass
```

```
class Utilisateur:
    prenom: str
    nom: str = "" # La valeur par défaut de l'attribut 'nom' est une chaine de c
```

On peut remarquer que définir la fonction `__init__` avec le module `dataclasses` est bien plus lisible et bien plus court qu'avec la méthode classique.

Pour la méthode `__repr__` :

Lorsqu'on a défini notre classe de données avec le décorateur `@dataclass`, la création de la méthode `__repr__` se fait automatiquement :

In [ ]:
```
utilisateur_01 = Utilisateur("Jean")
print(repr(utilisateur_01))
```

Différencier attributs d'instance et attributs de classe

Pour les attributs d'instance, on a vu qu'on pouvais les écire avec le nom de l'attribut suivi de `:` et du type (`str` , `int` , `list` ...) et qu'on pouvait lui assigner une valeur par défaut en ajoutant `=` suivi de la valeur.

Lorsqu'on veut créer un attribut de classe de données, il faudra au préalable importer `ClassVar` contenu dans le module `typing` . Pour créer un attribut de classe, on écrira le nom de l'attribut suivi de `: ClassVar[]` avec entre les crochets, le type de l'attribut (`int` , `str` , `list` ...). De même, pour spécifier une valeur par défaut, on ajoute un `=` suivi de la valeur par défaut.

Par exemple :

In [ ]:
```
from typing import ClassVar

@dataclass
class Utilisateur:
    prenom: str
    nom: str = ""
    compteur: ClassVar[int] = 0
```

La méthode `__post_init__`

Lorsqu'on défini une méthode `__post_init__` dans une classe de données, cette méthode sera appellée automatiquement après la méthode `__init__` .

Par exemple :

In [ ]:
```
@dataclass
class Utilisateur:
    prenom: str
    nom: str = ""
    compteur: ClassVar[int] = 0

    def __post_init__(self):
        self.nom_complet = f"{self.prenom} {self.nom}"
```

```
utilisateur_01 = Utilisateur("Jean", "Bernier")
print(utilisateur_01.nom_complet)
```

Chapitre 3
Les différentes méthodes

Les méthodes de classe

Les méthodes de classes, au lieux d'appartenir à la classe, ce sont des méthodes qui au lieux d'appartenir aux instances, vont appartenir directement à la classe. Elles ont pour avantage de pouvoir des des "instances par défault" de notre classe.

Pour définir une méthode comme une méthode de classe on va utiliser le décorateur `@classmethod`. Cette méthode aura pour paramètre un argument appellé conventionnellement `cls` qui va représenter notre classe.

Dans l'exemple suivant, lorsqu'on veut créer une Voiture qui as les caractéristiques d'une Lamborghini, au lieux de devoir ajouter les paramètres un à un dans une instance, on créer cette instance directement avec la méthode de classe `lamborghini`.

In [ ]:
```
class Voiture:
    def __init__(self, marque, vitesse, prix):
        self.marque = marque
        self.vitesse = vitesse
        self.prix = prix

    @classmethod
    def lamborghini(cls):
        return cls(marque="Lamborghini", vitesse=250, prix=200000)

    @classmethod
    def porsche(cls):
        return cls(marque="Lamborghini", vitesse=250, prix=200000)

# Au lieux de devoir écrire :
lambo = Voiture(marque="Lamborghini", vitesse=250, prix=200000)
# On écrit :
lambo = Voiture.lamborghini()

# Au lieux de devoir écrire :
porsche = Voiture(marque="Lamborghini", vitesse=250, prix=200000)
# On écrit :
porsche = Voiture.porsche()
```

Les méthodes statiques

Les méthodes statiques sont des méthodes qui ne possèdent aucun arguments (même pas `self` ni `cls`). Pour créer des méthodes statiques, on va devoir ajouter le décorateur `@staticmethod` pour spécifier au programme que c'est une méthode statique.

In [ ]:
```python
class Voiture:
    voiture_crees = 0
    def __init__(self, marque, vitesse, prix):
        Voiture.voiture_crees += 1
        self.marque = marque
        self.vitesse = vitesse
        self.prix = prix

    @classmethod
    def lamborghini(cls):
        return cls(marque="Lamborghini", vitesse=250, prix=200000)

    @classmethod
    def porsche(cls):
        return cls(marque="Lamborghini", vitesse=250, prix=200000)

    @staticmethod
    def nombre_voitures():
        print(f"Vous avez {Voiture.voiture_crees} voiture(s) dans votre garage."

lambo = Voiture.lamborghini()
porsche = Voiture.porsche()
Voiture.nombre_voitures()
```

La méthode `__str__`

Lorsqu'on essaye d'afficher un objet avec un `print`, on obtient généralement peu d'informations (ex : `<__main__.Voiture object at 0x0000012F1EED19D0>`). Pour avoir un peu plus d'informations, on peut définir la fonction `__str__` qui va nous permettre, lorsqu'on affiche un objet, de pouvoir afficher les informations qu'on veut.

Par exemple :

In [ ]:
```python
class Voiture:
    def __init__(self, marque, vitesse):
        self.marque = marque
        self.vitesse = vitesse

    def __str__(self):
        return f"La voiture de marque {self.marque} a une vitesse maximale de {s

porsche = Voiture("Porsche", 200)
print(porsche)
```

Chapitre 4
Notion d'héritage

L'héritage : c'est quoi ?

L'héritage *(concept avancé de la programmation orientée objet)* va nous permettre d'éviter de répéter inutilement des lignes de code.

Dans l'exemple suivant, on a une classe `Utilisateur` qui permet de créer un utilisateur avec un nom et un prénom. On peut également afficher (pour chaque utilisateur) `projets` qui est une liste contenant des noms de projets (dont on veut que ceux qui commencent par `pr_` soient des projets protégés).

In [ ]:
```python
projets = ["pr_GameOfThrones", "HarryPotter", "pr_Avengers"]

class Utilisateur:
    def __init__(self, nom, prenom):
        self.nom = nom
        self.prenom = prenom

    def __str__(self):
        return f"Utilisateur : {self.nom} {self.prenom}"

    def afficher_projets(self):
        for projet in projets:
            print(projet)

paul = Utilisateur("Paul", "Dupuis")
```

Admettons que Paul Dupuis vient d'entrer dans un entreprise. On lui crée donc un profile utilisateur pour qu'il puisse accèder aux projet de l'entreprise. Or, on voudrait le restreindre pour qu'il puisse n'avoir accès qu'à certain projets (on verra ça avec dans la section sur la surcharge)

On créer donc l'utilisateur Paul Dupuis avec la classe `Utilisateur` mais pour que Paul n'ait pas accès aux projets qui sont protégés, il va faloir créer une nouvelle classe (ex : `Junior`) qui va hériter de `Utilisateur`. On appellera `Utilisateur` la **classe parent** et `Junior` la **classe fille** *(ou classe enfant)* et on dira que **`Junior` hérite de la classe `Utilisateur`**.

On va donc créer la classe `Junior` et on va spécifier, dans les parenthèses, la classe dont `Junior` va hériter (ex : `class junior(Utilisateur)`). Une fois l'héritage effectué, on peut récupérer les attributs de la classe parent (`Utilisateur`) dans la classe enfant (`Junior`). On va donc, dans la fonction `__init__` de la classe `Junior`, appeler la fonction parent pour récupérer ses attributs.

In [ ]:
```python
class Junior(Utilisateur): # 'Junior' hérite de la classe 'Utilisateur'
    def __init__(self, nom, prenom): # On définit les mêmes arguments que dans '
        Utilisateur.__init__(self, nom, prenom) # On récupère les attributs de '

paul = Junior("Paul", "Dupuis")
print(paul)
paul.afficher_projets()
```

La fonction `super()`

Lorsqu'une classe enfant hérite d'une classe enfant, on a vu précedemment qu'il fallait appeller la classe parente (ex : `Utilisateur.__init__(self, nom, prenom)`). Avec cette méthode, on est obligé d'indiquer le nom de la classe. Le problème étant que si on

utilise plusieurs fois des méthodes de la classe parente à l'intérieur de notre classe enfant, et que par la suite on souhaite modifier le nom de cette classe : il faudra donc changer à tous les endroits où on utilise la classe parente (c'est plus dure à expliquer qu'à comprendre je vous rassure). C'est là qu'intervient la fonction `super()` puisqu'elle nous permet d'appeler directement les méthodes de la classe parente sans avoir besoin d'indiquer son nom. Attention, avec la fonction `super()`, il est inutile d'ajouter le self dans les parenthèse du `__init__` qui suit.

Dans l'exemple précédent, ça aurait donnée ça :

In [ ]:
```
class Junior(Utilisateur):
    def __init__(self, nom, prenom):
        super().__init__(nom, prenom) # On remplace 'Utilisateur' par 'super()'
```

La surcharge

On a vu précédemment que la classe `Junior` héritait de la méthode `afficher_projets` dans la classe parente. Or, on veut que lorsqu'un membre junior de l'entreprise (ex : Paul) ne puisse pas avoir accès aux projets protégés. Pour ce faire, on va surcharger la fonction `afficher_projets`, c'est à dire qu'on va re-définir cette fonction dans la classe `Junior`. Lorsqu'on appelera cette fonction, c'est la fonction "la plus proche" (qui se trouve dans la classe la plus près) qui sera appellée.

Si on veut appeller la fonction `afficher_projets` contenue dans la classe parente, on peut utiliser la fonction vue précédemment : `super().afficher_projet()`.

In [ ]:
```
class Junior(Utilisateur):
    def __init__(self, nom, prenom):
        super().__init__(nom, prenom)

    def afficher_projets(self):
        for projet in projets:
            if not projet.startswith("pr_"):
                print(projet)

paul = Junior("Paul", "Dupuis")
paul.afficher_projets()
```

Le polymorphisme

Le polymorphisme est un concept associé aux classes qui indique qu'on doit pouvoir utiliser des méthodes de la même façon sur tous les objets d'une même entité.

Dans l'exemple suivant, on utilise la même méthode (`avance`) pour deux entité différentes (`a` et `v`). C'est le même principe avec la méthode `len` qui peut être à la fois utilisé sur une liste ou sur une chaine de caractères.

In [ ]:
```
class Vehicule:
    def avance(self):
        print("Le véhicule démarre")
```

```python
class Voiture(Vehicule): # Hérite de 'Vehicule'
    def avance(self):
        super().avance()
        print("La voiture roule")

class Avion (Vehicule): # Hérite de 'Vehicule'
    def avance(self):
        super().avance()
        print("L'avion vol")

v = Voiture()
a = Avion()
a.avance()
v.avance()
```