Définition

La programmation orientée objet ou POO est un **paradigme** de programmation consistant en une **abstraction** des éléments de notre programme en **objets**, possédant des **propriétés** et des fonctions (appelées **méthodes** dans un contexte POO).

Pour cela, nous créons des **classes**, qui serviront de **modèle** pour nos objets qui, une fois **initialisées**, vont partager des propriétés et méthodes en commun.

Création d'une classe

Nous allons d'abord créer une classe simple, avec des propriétés prédéfinies, et sans méthode.

```
class User:
 nom    = "Lama"
 prenom = "Serges"

print(User.prenom) # L'opérateur . est l'opérateur d'appartenance
```

Le constructeur

Le constructeur est la **méthode** qui sera **systématiquement appelée** lorsqu'on **instanciera** une classe, c'est à dire qu'on créé un objet à partir d'une classe. La classe sert ici de **patron**, elle proposera un **squelette** qui sera **commun à tous les objets issus de cette classe**.

```
class User:
 def __init__(self, nom: str, prenom: str):
  self.nom    = nom
  self.prenom = prenom

user1 = User("Lama", "Serges") # Appel de la méthode __init__
user2 = User("Robin", "Hotton") # Appel de la méthode __init__

print(user1.nom)
print(user2.nom)
```

> `__init__(self, propriete_1, propriete_2, ...) ->` La méthode constructeur qui est appelée à chaque fois qu'on instancie la classe.

> `self ->` Défini dans la classe, mais représente l'objet en cours lorsqu'on appelle le constructeur.

Le constructeur permet de mettre en paramètre la **valeur des propriétés** que prendra l'objet, permettant ainsi de construire **une multitude d'objets** avec des propriétés différentes, mais suivant un même patron, car tous des instances de la même classe.

Les méthodes

Les **méthodes** sont des fonctions définies dans la classe, et qui seront partagées par tous les objets issus de cette classe.

```
class User:
    def __init__(self, nom: str, prenom: str):
        self.nom = nom
        self.prenom = prenom

 # Méthode appelable depuis tous les objets
    def get_full_name(self):
        print(f"Utilisateur {self.nom.upper()} {self.prenom.capitalize()}")

user1 = User("Doe", "john")
user2 = User("Hotton", "Robin")

print(user1.get_full_name())  # ici on doit print car la methode renvoie
print(user2.get_full_name())  # une chaine de caractère.
```

Les propriétés statiques

Les propriétés et méthodes d'une classe sont partagées entre tous les objets-instances de cette classe. Quand nous avons besoin des **propriétés de la classe, et non pas des objets** issus de cette classe, on utilise des propriétés **statiques**.

Par exemple, grâce à une propriété statique, on va pouvoir compter le nombre d'instance d'une classe.

```
class User:
    nombre_instance = 0  # Propriété statique de la classe

    def __init__(self, nom: str, prenom: str):
        User.nombre_instance += 1  # nombre_instance appartient à User, pas à self
        self.nom = nom
        self.prenom = prenom

    def get_full_name(self):
        print(f"Utilisateur {self.nom.upper()} {self.prenom.capitalize()}")

user1 = User("Doe", "john")  # User.nombre_instance += 1
```

```
user2 = User("Hotton", "Robin")  # User.nombre_instance += 1

print(User.nombre_instance)
```

Les méthodes statiques

Tout comme les propriétés, **les méthodes peuvent être statiques** grâce au décorateur `@staticmethod`.

```
class User:
 nombre_instance = 0
 def __init__(self, nom: str, prenom: str):
  User.nombre_instance += 1
  self.nom    = nom
  self.prenom = prenom


 def get_full_name(self) -> None:
  print(f"{self.nom.upper()} {self.prenom.capitalize()}")

 # Méthode de classe (statique), self n'est pas en paramètre
 # cls représente la classe (nom libre mais cls est une convention)
 @staticmethod
 def get_instance_count():
  print(f"Le nombre d'instance est {User.nombre_instance}")


john = User("Doe", "john")
john.get_full_name()  # On appelle la méthode d'objet depuis l'objet
User.get_instance_count() # On appelle la méthode statique depuis la classe

robin = User("Hotton", "Robin")
robin.get_full_name()
User.get_instance_count()
```

Les méthodes de classe

Les méthodes de classe ou `@classmethod`, bien qu'ayant des propriétés similaires aux méthodes statiques, ont un fonctionnement différent et ne sont pas utilisées de la même manière.

En général, nous utiliseront des `@staticmethod` pour la **gestion des propriétés statiques** de la classe, et nous utiliseront les `@classmethod` pour créer des **"usines" d'objet, ou "factories"**.

```python
class User:
 nombre_instance = 0
 def __init__(self, nom: str, prenom: str):
  User.nombre_instance += 1

  self.nom    = nom
  self.prenom = prenom

 def get_full_name(self):
  print(f"{self.nom.upper()} {self.prenom.capitalize()}")

 @staticmethod
 def get_instance_count():
  print(f"Le nombre d'instance est {User.nombre_instance}")

 # Contrairement à une méthode statique, la classe doit être en paramètre
 @classmethod
 def create_zizou(cls):
  return cls("Zidane", "Zinédine")


zizou = User.create_zizou()
zizou2 = User("Zidane", "Zinédine")

print(zizou.__dict__)
print(zizou2.__dict__)
```

Les méthodes magiques

Les méthodes magiques sont implémentées directement dans Python et sont accessibles depuis n'importe quelle classe. Il en existe une pléthore et nous allons en voir quelques une ici.

> `__str__(self) ->` Régis le comportement de n'importe quel objet lorsqu'il sera print, ou converti en chaîne de caractères.

```python
class User:
 def __init__(self, nom: str, prenom: str):
  self.nom    = nom
  self.prenom = prenom

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

johnny = User("Halliday", "johnny")
print(johnny)
```

> `__eq__(self, other) ->` Régis le comportement à adopter lorsqu'on essaye de vérifier si un objet de la classe est égal (`==`) à un autre objet de cette même classe.

```python
class Youtuber:
 def __init__(self, nom_de_chaine: str, nb_abonnes: int):
  self.nom_de_chaine = nom_de_chaine
  self.nb_abonnes    = nb_abonnes

 # 2 youtubers ayant le même nombre d'abonnés sont considérés comme égaux
 def __eq__(self, other):
  return self.nb_abonnes == other.nb_abonnes

squeezie = Youtuber("Squeezie", 10000000)
zerator  = Youtuber("Zerator", 10_000_000)
tuor   = Youtuber("TheTuor59", 2)

print(squeezie == zerator)
# print(squeezie == zerator == killer) # False
# print(squeezie == zerator and squeezie == killer and zerator == killer)
print(zerator == tuor)
```

> Liste des méthodes magique