🔄 Le Polymorphisme (POO) en Python

📚 Définition

Le **Polymorphisme** (du grec *poly* signifiant "plusieurs" et *morphē* signifiant "forme") est le principe qui permet à des objets de classes différentes de **répondre différemment au même message (ou appel de méthode)**.

En Python, le polymorphisme est géré de manière très intuitive et se manifeste principalement de deux façons :

1. Le Polymorphisme par Surcharge de Méthode (Method Overriding)

C'est la forme que nous avons déjà légèrement touchée avec l'Héritage. Elle se produit lorsqu'une **sous-classe (enfant)** fournit sa propre implémentation d'une méthode qui est déjà définie dans sa **super-classe (parent)**.

- **Règle :** Le nom de la méthode et ses paramètres doivent être les mêmes que ceux de la méthode du parent.
- **But :** Spécialiser le comportement hérité pour qu'il soit spécifique à la sous-classe.

Exemple de la moto : Dans l'exercice précédent, la classe Moto a **surchargé** la méthode demarrer() de la classe Vehicule pour ajouter un message spécifique ("Le moteur vrombit fort !").

2. Le Polymorphisme par Interfaces (Duck Typing)

C'est le mécanisme le plus puissant et le plus courant du polymorphisme en Python. Le "Duck Typing" (typage canard) est basé sur le proverbe :

"Si ça marche comme un canard et que ça cancane comme un canard, c'est un canard."

En POO Python, cela signifie que Python ne se soucie pas du **type** d'un objet (s'il est une Voiture ou une Moto), mais uniquement s'il possède la **méthode** que vous essayez d'appeler.

Démonstration du Duck Typing

Considérez la fonction suivante :

Python

```
def faire_demarrer(objet_vehicule):
    """
    Cette fonction appelle la méthode 'demarrer()' sur n'importe quel objet.
    """
    objet_vehicule.demarrer()
```

Si nous appliquons cette fonction à différents objets, la même ligne de code (objet_vehicule.demarrer()) produira un résultat différent en fonction de la classe de l'objet :

Python

```
# En utilisant les classes de la solution précédente

ma_voiture = Voiture("BMW", "Série 3", 4)
ma_moto = Moto("Harley", "Fat Bob")

print("--- 🦆 Test de Duck Typing ---")

# La voiture appelle sa version HÉRITÉE de demarrer()
faire_demarrer(ma_voiture)

# La moto appelle sa version SURCHARGÉE de demarrer()
faire_demarrer(ma_moto)
```

Explication :

1. Python voit l'appel objet_vehicule.demarrer().
2. Il ne se demande pas : "Est-ce un objet de type Voiture ou Moto ?"

3. Il se demande plutôt : **"Est-ce que cet objet possède une méthode demarrer() ?"**
4. Puisque les deux objets possèdent cette méthode (soit par héritage, soit par surcharge), la fonction les traite de la même manière, mais l'exécution interne de demarrer() est **polymorphe** (elle prend plusieurs formes).

🛠️ Exercice Pratique sur le Polymorphisme

Utilisons le concept de Duck Typing pour solidifier la compréhension.

Créez deux classes complètement indépendantes qui n'héritent pas l'une de l'autre, mais qui partagent une méthode de même nom.

1. **Classe Oiseau :**
    - Méthode voler() qui affiche : "L'oiseau s'envole en battant des ailes."
2. **Classe Avion :**
    - Méthode voler() qui affiche : "L'avion décolle et prend de l'altitude."

Ensuite, créez une fonction faire_voler(chose) qui prend un objet en argument et appelle sa méthode voler().

Testez la fonction avec une instance de chaque classe.