Introduction à la notion d'héritage avec un exemple simple

/home/phil/Documents/LMG/STS_A_2025-2027/PYTHON_STS2/C_classe_(heritage)/02_heritage_introduction

1. Exemple de classes Animal, Chien et Chat

Commençons par créer une classe Animal (avec un A majuscule) qui possède un nom et une manière de faire du bruit.

```
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def faire_du_bruit(self):
        return f"{self.nom} fait un bruit."
```

Nous allons maintenant créer une classe Chien. Le **chien *est* un animal** ; il semble naturel que la classe Chien hérite des propriétés et des comportements définis par la classe Animal. En informatique pour que cet héritage naturel se passe bien il faudra respecter une syntaxe. Pour que la classe Chien hérite des attributs et des comportements de la classe Animal , il faut et il suffit de passer la classe Animal en paramètre de la classe Chien.

```
class Chien(Animal):
    pass

mon_toutou = Chien("Medor")
print(mon_toutou.faire_du_bruit()) # Affiche: Medor fait un bruit.
```

On remarque qu'il n'a pas été nécessaire d'écrire de méthode *__init__ ()* pour la classe Chien. La méthode *__init__ ( )* de la classe mère (*Animal*) a été utilisée par défaut. Conséquence :

```
mon_toutou.nom = " Medor "
```

Même mécanisme pour la méthode *faire_du_bruit()* . En l'absence de méthode *faire_du_bruit()* dans la classe Chien, c'est la méthode *faire_du_bruit()* de la classe Animal qui est utilisée.

💡Il peut être intéressant de préciser le comportement d'un chien qui fait du bruit. Pour cela redéfinissons la méthode *faire_du_bruit()* dans la classe *Chien*. (***redéfinition*** = ***override*** en anglais)

```
class Chien(Animal):
    def faire_du_bruit(self):
        return f"{self.nom} aboie."

mon_toutou = Chien("Medor")
print(mon_toutou.faire_du_bruit()) # Affiche: Medor aboie.
```

Parfois il n'est pas nécessaire de redéfinir mais simplement d'étendre le comportement de la classe mère. Si nous souhaitons définir un chat par son nom et sa couleur ; nous aurons besoin de deux paramètres (nom, couleur). Pour cela nous allons étendre le constructeur de la classe mère.

```
# Chat hérite de Animal
class Chat(Animal):
    def __init__(self, nom, couleur):
        super().__init__(nom) # exécute __init__ () de Animal (=la classe mère)
        self.couleur = couleur  # ajoute un nouvel attribut spécifique au chat

mon_chat = Chat("Titi", "Gris")
print(mon_chat.couleur)          # Affiche : Gris
```

Ci-dessus, le constructeur de la classe Chat appelle le constructeur de sa classe mère et ajoute ensuite un attribut couleur.

💡 Notez l'appel au constructeur de la classe mère : **`super().__init__ (<param>)`**

1) Ajouter maintenant une méthode à la classe Chat pour obtenir le résultat suivant :

```
print(mon_chat.faire_du_bruit())

Titi miaule.
```

2) Modifier ensuite le programme pour obtenir le résultat suivant :

```
mon_toutou = Chien("Medor")
print(mon_toutou.faire_du_bruit())
print(mon_toutou.mordre())

Medor aboie.
Medor mord.
```

💡 Pour retrouver la classe à laquelle appartient un objet, utiliser la fonction *type()*

```
print(type(mon_toutou))

<class '__main__.Chien'>
```

```
animaux = [Chien("Medor"), Chat("Titi", "Gris")]
for a in animaux:
    print(a.faire_du_bruit())
# Medor aboie.
# Titi miaule.
```

💡 La même fonction appelée pour différents objets conduit à des résultats différents. On parle de **polymorphisme**. C'est un des gros atouts de la Programmation Orientée Objet (POO)

2. Application : tous à la Ferme ! tous_a_la_ferme.py

Consigne : Créer une classe *Vache* qui hérite de *Animal*, possède un attribut *production_lait* (en litres/jour) et une méthode *faire_du_bruit()* qui renvoie *"<nom> meugle."*.
Ajouter une méthode *traire()* qui renvoie *"<nom> produit X litres de lait par jour."*
Ecrire les instructions de test nécessaires à la validation de l'ensemble.

3. Application : tous au Zoo ! tous_au_zoo.py

Consigne : Créer 4 classes héritant de *Animal* (*Chien*, *Chat*, *Vache*, *Canard*). Chaque animal a son propre cri. Écrire ensuite une fonction *concert(animaux)* qui prend une liste et affiche le cri de chaque animal. Ajouter ensuite un attribut *regime* (*"carnivore"* / *"herbivore"* / *"omnivore"*) et une méthode *manger(aliment)* qui renvoie *"<nom> mange <aliment>."* si l'aliment est compatible, sinon *"<nom> refuse <aliment>."*
Ecrire les instructions de test nécessaires à la validation de l'ensemble.

4. Application : Véhicules vehicules.py

Consigne : Créer une classe *Vehicule* avec un attribut *marque* et une méthode *se_deplacer()* qui renvoie *"<marque> se déplace."*. Faire hériter *Voiture* (attribut *nb_portes*), *Velo* (attribut *nb_vitesses*) et *Camion* (attribut *charge_max*). Chacune de ces classes redéfinira *se_deplacer()*.
Ajouter une méthode *klaxonner()* qui affiche un son différent selon le véhicule.
Ecrire les instructions de test nécessaires à la validation de l'ensemble.

5. Défi : Trouver les erreurs dans le code ci-dessous

⚠️ **Seule une lecture attentive est autorisée pour trouver les 3 erreurs.**

```
class Animal:
    def __init__(self, nom):
        self.nom = nom

class Chien(Animal):
    def __init__(self, nom, race):
        super().__init__(self, nom)
        self.race = race

    def faire_du_bruit(self):
        return f"{nom} aboie."

mon_chien = Chien("Rex")
print(mon_chien.faire_du_bruit())
```

