Ecriture des __str__ et __repr__ du fichier Animal_TODO.py

🎯 Objectif. être capable de :

- Écrire `__str__` pour qu'un objet s'affiche proprement à l'utilisateur.
- Écrire `__repr__` pour qu'un objet s'affiche précisément au développeur.
- Étendre une méthode de la classe mère avec `super()` quand c'est pertinent.
- Réécrire une méthode quand le comportement change vraiment.

Table des matières

1. 📚 Rappel 1 — `__str__` ou `__repr__` : quelle différence ?....1
2. 📚 Rappel 2 — Comment écrire ces méthodes....2
   a. Pour `__str__`....2
   b. Pour `__repr__`....2
3. Ecriture guidée pas à pas....2
   a. TODO 1 — `Animal.__str__`....2
   b. TODO 2 — `Animal.__repr__`....2
   c. TODO 3 — `Chien.__str__` (ÉTENDRE)....3
   d. TODO 4 — `Chien.__repr__` (RÉÉCRIRE)....3
   e. TODO 5 — `Chat.__str__`....3
   f. TODO 6 — `Chat.__repr__`....4
4. Expériences à faire une fois les TODO complétés....4
5. Grille d'auto-évaluation....4

1. 📚 Rappel 1 — `__str__` ou `__repr__` : quelle différence ?

Méthode	Répond à la question	À destination de	Exemples d'utilisation
__str__	Comment présenter l'objet à un humain ?	Utilisateur	print(obj), str(obj), f"{obj}"
__repr__	Comment décrire l'objet et sa structure.	Développeur	repr(obj)`, console, `print(liste)`

Règle d'or :

- `__str__` = joli pour l'utilisateur.
- `__repr__` = précis et réexécutable pour le développeur. On doit pouvoir recréer l'objet à l'identique à partir de l'information retournée ; d'où l'mportance des guillemets pour encadrer les chaînes de caractères (string).

À retenir aussi :

- Si `__str__` n'existe pas, Python utilise `__repr__` par défaut.
- Dans une **liste** (`print([obj1, obj2])`), c'est toujours `__repr__` qui s'affiche.

2. 📚 Rappel 2 — Comment écrire ces méthodes

a. Pour `__str__`

```
def __str__(self):
    return f"une phrase lisible avec {self.nom}"
```

- Une seule ligne de `return`.
- Une **f-string** en français, lisible par un utilisateur.
- Pas de `print` à l'intérieur.

b. Pour `__repr__`

```
def __repr__(self):
    return f"{self.__class__.__name__}(nom={self.nom!r})"
```

- `self.__class__.__name__` → donne le nom de la classe courante (`"Animal"`, `"Chien"`, etc.).
- `{self.nom!r}` → affiche la valeur avec ses guillemets (indispensable pour du code Python valide).
- **Le résultat doit ressembler à un appel de constructeur** : `Animal(nom='Rex')`.

3. Ecriture guidée pas à pas

a. TODO 1 — `Animal.__str__`

Ce qu'on veut obtenir : `Animal nommé Rex`

Le problème se résout en répondant à trois questions :

1. Quel mot fixe ? → `"Animal nommé "`
2. Quelle valeur variable ? → `self.nom`
3. Comment les coller ensemble ? → une f-string.

Compléter : `return f".....{self.nom}"`

Vérification :

```
print(Animal("Rex"))   # doit afficher  Animal nommé Rex
```

b. TODO 2 — `Animal.__repr__`

Ce qu'on veut obtenir : `Animal(nom='Rex')`

Le problème se résout en répondant à trois questions :

1. Comment obtenir le nom de la classe ? → `self.__class__.__name__`
2. Comment afficher `'Rex'` **avec** les guillemets ? → `{self.nom!r}`
3. Comment assembler les deux ? → une f-string.

Résultat : `return f"{self.__class__.__name__}(nom={self.nom!r})"`

Vérification :
```
print(repr(Animal("Rex")))   # doit afficher  Animal(nom='Rex')
```

c. TODO 3 — `Chien.__str__` (ÉTENDRE)

Ce qu'on veut obtenir : `Animal nommé Médor, race Berger allemand`

⚠️ **Piège à éviter :**

Ne pas réécrire `"Animal nommé "` à la main. Réutilise-le via `super().__str__()`.

Pourquoi ?

Si demain `Animal.__str__` change (par exemple en `"Animal : "`), la version de `Chien` doit suivre automatiquement. Si tu recopies, ton code sera obsolète sans que tu t'en rendes compte.

A compléter : `return f"{.....}, race {self.race}"`

Vérification :
```
c = Chien("Médor", "Berger allemand")
print(c)  # doit afficher Animal nommé Médor, race Berger allemand
```

d. TODO 4 — `Chien.__repr__` (RÉÉCRIRE)

Ce qu'on veut obtenir : `Chien(nom='Médor', race='Berger allemand')`

⚠️ **Piège à éviter :** Ne pas réutiliser `super().__repr__()`. Cela donnerait :
```
Animal(nom='Médor'), race Berger allemand
```

→ ce n'est pas du code Python valide pour créer cet objet. Ce n'est pas ce que l'on veut.

Pourquoi réécrire ? : `__repr__` doit refléter la classe réellement instanciée ; c'est à dire celle qui a été utilisée pour créer l'objet ; et avec tous ses attributs. Ici : `nom` *et* `race`.

Résultat : `return f"Chien(nom={self.nom!r}, race={self.race!r})"`

Vérification :
```
c = Chien("Médor", "Berger allemand")
print(repr(c))   # doit afficher  Chien(nom='Médor', race='Berger allemand')
```

e. TODO 5 — `Chat.__str__`

Ce qu'on veut obtenir : `Animal nommé Titi, couleur Gris`

Même logique que TODO 3 : étendre `super().__str__()` avec l'attribut propre au chat (`couleur`).

A compléter : `return f"{.....}, couleur {self.couleur}"`

f. TODO 6 — `Chat.__repr__`

Ce qu'on veut obtenir : `Chat(nom='Titi', couleur='Gris')`

Même logique que TODO 4 : réécrire entièrement.

A compléter : `return f"..."`

4. Expériences à faire une fois les TODO complétés

Ces expériences aident à comprendre pourquoi on a fait ces choix.

Expérience 1 - Supprimer (ou mettre en commentaires) `Chien.__str__`

Que se passe-t-il quand on fait `print(medor)` ?
→ On obtient `Animal nommé Médor` (sans la race). D'où l'intérêt d'**étendre** la méthode.

Expérience 2 — Supprimer `Chien.__repr__`

Que se passe-t-il quand on fait `print(repr(medor))` ?
→ On obtient `Chien(nom='Médor')` (sans la race). D'où l'intérêt de **réécrire**.

Expérience 3 — Supprimer `Animal.__str__`

Que se passe-t-il quand on fait `print(medor)` ?
→ Python utilise `__repr__` par défaut. On voit que `__repr__` sert de **filet de sécurité**.

Expérience 4 — Remplacer `{self.nom!r}` par `{self.nom}` dans `Animal.__repr__`

Que se passe-t-il ?
→ On obtient `Animal(nom=Rex)` : plus de guillemets, donc le code Python n'est plus valide.

5. Grille d'auto-évaluation

Coche chaque case quand vous êtes sûr(e) de vous.

☐ Je sais écrire `__str__` dans la classe mère.
☐ Je sais utiliser `self.__class__.__name__` dans `__repr__`.
☐ Je sais utiliser `!r` pour afficher avec guillemets.
☐ Je sais ÉTENDRE `__str__` dans une classe fille avec `super()`.
☐ Je sais pourquoi on NE réutilise PAS `super().__repr__()` dans une classe fille.
☐ Je sais pourquoi `print([obj1, obj2])` affiche le `__repr__`.
☐ Mon code s'exécute sans erreur et tous les « Attendu » sont respectés.

5 cases ou plus → l'essentiel est maitrisé.
moins de 4 cases →Il est nécessaire de reprendre les rappels 1 et 2, puis refaire les TODO 1 et 2.