TP — Programmation Orientée Objet en Python

Thème : Création d'un mini-jeu de rôle (RPG)

Durée : 3 heures

Objectifs pédagogiques

À l'issue de ce TP, vous serez capable de :

- Définir une classe Python avec attributs et méthodes
- Écrire et utiliser un constructeur (`__init__`)
- Manipuler l'encapsulation de base (attributs, propriétés)
- Mettre en œuvre l'héritage entre classes
- Redéfinir (override) des méthodes et utiliser `super()`
- Utiliser le polymorphisme pour manipuler des objets de classes différentes de façon uniforme
- Implémenter quelques méthodes spéciales (`__str__` , `__repr__` , `__eq__`)

Contexte

Vous allez développer, étape par étape, un petit système de combat pour un jeu de rôle. Le TP est découpé en 4 parties de difficulté croissante. Chaque partie doit être testée avant de passer à la suivante.

Créez un seul fichier `rpg.py` que vous complèterez au fil du TP, avec un bloc de tests à la fin (sous `if __name__ == "__main__":`).

.

.

.

.

Partie 1 — La classe de base `Personnage` (45 min)

1.1 — Constructeur et attributs

Créez une classe `Personnage` avec :

- Un constructeur prenant `nom`, `points_de_vie` (par défaut 100) et `force` (par défaut 10)
- Ces trois valeurs doivent être stockées comme attributs d'instance
- Un attribut `est_vivant` initialisé à `True`

1.2 — Méthodes

Ajoutez les méthodes suivantes :

Méthode	Rôle
`attaquer(cible)`	Réduit les PV de `cible` de la valeur de `self.force`, puis affiche un message du type `"Aragorn attaque Gobelin et inflige 10 dégâts !"`
`subir_degats(degats)`	Diminue `points_de_vie`. Si `points_de_vie <= 0`, fixe `points_de_vie` à 0 et `est_vivant` à `False`
`est_en_vie()`	Retourne `True` / `False` selon l'attribut `est_vivant`
`se_reposer(points)`	Augmente les PV sans dépasser un maximum de 100

1.3 — Méthode spéciale

Ajoutez `__str__` pour qu'`print(personnage)` affiche par exemple :

```
Aragorn — PV: 100/100 — Force: 10
```

Test attendu

```python
p1 = Personnage("Aragorn", points_de_vie=80, force=15)
p2 = Personnage("Gobelin")
p1.attaquer(p2)
print(p2)
print(p2.est_en_vie())
```

Partie 2 — L'héritage : des classes spécialisées (1h)

Créez trois classes héritant de `Personnage` : `Guerrier`, `Mage` et `Voleur`.

2.1 — `Guerrier`

- Constructeur identique à `Personnage`, plus un attribut `armure` (par défaut 5)
- Redéfinit `subir_degats(degats)` : l'armure réduit les dégâts reçus (`degats - armure`, minimum 1 dégât), puis appelle la méthode de la classe parente via `super()` avec les dégâts recalculés

2.2 — `Mage`

- Ajoute un attribut `points_de_mana` (par défaut 50)
- Ajoute une méthode `lancer_sort(cible)` qui coûte 20 mana et inflige `force * 2` dégâts à `cible`. Si le mana est insuffisant, afficher un message et ne rien faire
- Redéfinit `se_reposer(points)` pour qu'elle régénère aussi 10 points de mana en plus des PV (en réutilisant `super().se_reposer(points)`)

2.3 — `Voleur`

- Ajoute un attribut `chance_critique` (probabilité entre 0 et 1, par défaut 0.2)
- Redéfinit `attaquer(cible)` : avec une probabilité `chance_critique` (utilisez le module `random`), les dégâts infligés sont doublés ("coup critique !"). Réutilisez la logique de la classe parente autant que possible avec `super()`

Question de réflexion (à répondre en commentaire dans le code)

Pourquoi est-il préférable d'appeler `super().attaquer(cible)` plutôt que de dupliquer le code d'affichage et de calcul des dégâts dans `Voleur.attaquer` ?

.

.

.

.

.

Partie 3 — Polymorphisme et gestion de groupe (45 min)

3.1 — Une fonction polymorphe

Écrivez une fonction `combat(equipe1, equipe2)` qui prend deux listes de `Personnage` (ou sous-classes) et simule un combat au tour par tour très simple :

- Tant que les deux équipes ont au moins un membre vivant :
  - Chaque personnage vivant de `equipe1` attaque un personnage vivant aléatoire de `equipe2`, puis inversement
- Affiche à la fin quelle équipe a gagné

Cette fonction ne doit **jamais** tester le type exact des personnages (pas de `if type(p) == Guerrier`) : elle doit fonctionner uniquement grâce au polymorphisme (chaque objet sait comment il attaque et comment il encaisse des dégâts).

3.2 — Classe `Equipe`

Créez une classe `Equipe` qui encapsule une liste de personnages, avec :

- Un constructeur prenant un `nom` et une liste de personnages
- Une méthode `membres_vivants()` retournant la liste des personnages encore en vie
- Une méthode `est_defaite()` retournant `True` si plus aucun membre n'est vivant
- `__str__` qui liste les membres et leur état

Adaptez `combat()` pour qu'elle accepte deux objets `Equipe`.

Partie 4 — Pour aller plus loin (bonus, 30 min)

Choisissez **au moins une** des extensions suivantes :

1. **Méthode spéciale** `__eq__` : deux personnages sont considérés égaux s'ils ont le même nom et la même classe.
2. **Classe abstraite** : utilisez le module `abc` pour faire de `Personnage` une classe abstraite avec une méthode abstraite `capacite_speciale()`, que chaque sous-classe doit implémenter (`lancer_sort` pour le Mage, etc.).
3. **Système d'inventaire** : ajoutez une classe `Objet` (nom, bonus_force) et une méthode `equiper(objet)` sur `Personnage` qui augmente `force`.

4. **Journal de combat** : créez une classe `JournalDeCombat` qui enregistre chaque action (`attaquer`, `subir_degats`, etc.) sous forme de liste de chaînes de caractères, et affichez-la à la fin du combat.

Barème indicatif

Partie	Points
Partie 1 — classe de base	6
Partie 2 — héritage	7
Partie 3 — polymorphisme	5
Partie 4 — bonus	2
Qualité du code (nommage, docstrings, PEP8)	2

Consignes de rendu

- Un seul fichier `rpg.py`
- Chaque classe et méthode doit avoir une courte docstring
- Un bloc de test clair à la fin du fichier, sous `if __name__ == "__main__":`