OOP 3

On va plonger dans la séparation des responsabilités en créant un mini-jeu.

Introduction

Une application est un peu comme un restaurant. Au restaurant, vous n'interagissez jamais avec le cuisinier, alors que c'est pourtant lui qui prépare votre plat. Techniquement, vous pourriez manger directement dans sa cuisine debout au milieu des casseroles, ça marcherait très bien, mais ce serait désagréable pour tout le monde, et très vite ingérable.

Vous, vous interagissez avec un serveur, qui prend votre commande et s'occupe d'intéragir avec la cuisine afin de vous apporter votre plat. Ainsi, vous n'avez jamais à mettre les pieds en cuisine.

Il y a une séparation des responsabilités : le serveur s'occupe de vous, et le cuisinier s'occupe des plats.

L'intérêt, c'est que le cuisinier peut se concentrer sur son travail sans s'inquiéter d'un client qui veut une autre corbeille de pain ou trouve que la clim est trop forte et que ça lui casse la voix. Inversement, un serveur n'a même pas à savoir cuisiner !

Un autre bénéfice est qu'un serveur peut être remplacé par un autre serveur, ça n'a aucun impact sur le cuisinier : l'ensemble est modulaire, et on peut facilement remplacer des éléments.

Dans une application, il y a une interface utilisateur, la UI ("you-aïe"), qui a pour préoccupation d'afficher des choses à l'utilisateur, de récupérer ses inputs, et de lui re-présenter des choses en fonction de ces inputs (c'est le serveur qui s'occupe de vous).
Tout ce qu'on appelle la logique métier (la cuisine), qui consiste à manipuler des données, faire des calculs, doit être distinctement séparé de la UI, dont ce n'est pas le rôle. Ainsi, on pourra par la suite facilement changer de UI, sans impacter les fonctionnalités de l'application.

Présentation du projet

Nous allons programmer un tic-tac-toe, ce jeu très simple où deux joueurs tentent d'aligner respectivement trois X ou trois O dans une grille de 3x3.

Pour que l'exercice ne soit pas trop difficile, un squelette de code est déjà étayé. Dans les fichiers fournis, vous trouverez donc :

- `main.py`, contenant un main (le programme principal à executer)
- `game.py`, contenant:
  - la classe TicTacToeGame, contenant la logique du jeu, qu'il va falloir compléter
  - la classe CellSymbol, qui sert d'enum pour représenter un X / un O / une case vide
- `ui.py`, contenant la classe UserInterface, qu'il va falloir compléter

Echauffement

- Exécuter le programme pour s'assurer qu'il ne fait pas grand chose de formidable : normalement, vous pouvez taper le nom des joueurs, et voir une grille vide s'afficher dans la console à chaque tour. Et c'est déjà mieux que crasher, figurez-vous.

> Note: ne taper aucun nom pour un joueur équivaut à taper la string "Player 1" ou "Player 2" par défaut. Pour le moment, ça ne change rien, mais vous trouverez vite ça pratique pour vos tests sur la durée.

- Lisez le code de TicTacToeGame :
  - Déterminez ce qu'il se passe dans le constructeur.
  - Notez quelles sont les propriétés disponibles. Ces propriétés font partie de ce qui est exposé vers l'extérieur, c'est-à-dire que UserInterface pourra y faire appel.
  - De même, notez les méthodes disponibles qui pourront être appelées de l'extérieur par UserInterface.

> Remarque: une convention en Python est qu'une méthode commençant par un "_" n'est pas censée être appelée depuis l'exterieur, seulement en interne par la classe qui la déclare.

- Lisez le code de UserInterface :
  - Déterminez ce que fait le constructeur.
  - Déterminez ce qu'il se passe dans la méthode `show()`.

> Note : si vous ne connaissez pas le mot-clé `while`, `while` execute une boucle **tant que** la condition qui lui est associée est vraie. Un `while True:` va donc tourner pour toujours, sauf si le mot clé `break` est appelé à l'intérieur. `break` stoppe immédiatement n'importe quelle boucle pour en sortir.

> Note : le mot clé `continue` permet de passer immédiatement à l'itération suivante d'une boucle en cours. Cette information est purement indicative, il n'est pas nécessaire d'utiliser de `continue` dans cet exercice, ni de toucher à ceux qui sont en place.

> Note : `try` et `except` permettent de gérer un `raise` qui aurait potentiellement lieu dans le bloc try. Il n'est pas nécessaire de s'intéresser à ça pour cet exercice.

Quelques snippets de code sont disponibles tout à la fin, si vous en avez besoin.

1 - Initialisation des joueurs

Dans TicTacToeGame:

- Repenser à ce qu'il se passe dans le constructeur (vous n'avez pas à le modifier), puis implémenter `initialize()` pour que le jeu stocke réellement le nom des joueurs, et qu'il sache qui est le joueur courant.
  On considère que le joueur 1 commence la partie en premier.
- Implémenter `_get_player_symbol()`, pour renvoyer l'une des constantes de la class CellSymbol en fonction du joueur en paramètre. Vous pouvez considérer que le joueur 1 utilise les X, et le joueur 2 les O.
- Implémenter `swap_player()` pour que le jeu intervertisse le joueur courant avec l'autre joueur.

2 - Branchement de la UI

Maintenant que notre TicTacToeGame sait au moins se souvenir du prénom de ceux qui veulent jouer, faisons appel à son savoir-faire depuis l'extérieur.

Dans la méthode `show()` de UserInterface:

- Repérer où est appelée la méthode `initialize()` de l'objet TicTacToeGame, pour avoir l'exemple d'une de ses méthodes qu'on appelle depuis la UI.
- Implémenter le `# TODO: change player` pour faire véritablement le changement de joueur.
- Executer le main, et vérifier que le joueur change bien après chaque coup.

3 - Jouer un coup

Le jeu est déjà incroyable en l'état, mais accrochez-vous, on peut faire encore mieux.

Dans la class TicTacToeGame :

- Implémenter `play_move()`. Le paramètre de la fonction est un int, l'index de la grille où placer le symbole du joueur courant. Vous n'avez rien à modifier en dehors de cette fonction, utilisez ce dont vous disposez déjà.

Dans la class UserInterface :

- Implémenter le `# TODO: play the move`
- Executer le main, et vérifier qu'on peut désormais jouer des coups.
- Jouez toujours 4 pour les deux joueurs. Jouez 999, puis -1. Modifiez `play_move()` pour gérer ce qui vous semble problématique.

4 - Match nul

- Dans TicTacToeGame, implémenter `is_filled()` pour renvoyer si oui ou non le plateau est entièrement rempli.
- Dans UserInterface.show(), implémenter le `# TODO: check if the game can still be played` pour sortir de la boucle lorsque le plateau est rempli.
- Executer le main et remplir le plateau pour vérifier que le jeu s'arrête.
- Dans UserInterface, implémenter `display_game_over()` pour indiquer à l'utilisateur qu'il y a match nul.

5 - Accès à la victoire

Il est temps de déterminer un vainqueur à ce jeu fantastique quasiment au même niveau que Fortnite.

- Dans TicTacToeGame, implémenter la méthode `is_won_by()`.

Au lieu d'essayer d'écrire tous les cas d'un coup, procédez par étapes.
Vous pouvez commencer par renvoyer True si juste la première ligne est occupée par le symbole du joueur, et vérifier que ce cas-là fonctionne correctement.
Ensuite, vous pouvez faire en sorte de vérifier la même chose pour toutes les lignes horizontales.
Ensuite, pareil pour les colonnes, puis les diagonales.
Si vous en avez envie, vous pouvez faire des sous-fonctions pour différent cas.
Notez qu'il y a plusieurs façons possibles d'implémenter cette méthode, à vous de trouver celle qui vous convient.

- Dans UserInterface, implémenter le `# TODO: check if the game is won by the current player` et sortir de la boucle si c'est le cas.
- Dans UserInterface, modifier `display_game_over()` pour indiquer qui a gagné, s'il y a un gagnant.
- Executer le main :
    - Jouer 0, 8, 1, 7, 2, et vérifier que le joueur 1 l'emporte (ligne du haut)
    - Jouer 1, 0, 2, 3, 4, 6 et vérifier que le joueur 2 l'emporte (colonne de gauche)

- Jouer 2, 0, 8, 4, 5 et vérifier que le joueur 1 l'emporte (colonne de droite)
- Jouer 4, 0, 2, 1, 6 et vérifier que le joueur 1 l'emporte (diagonale /)
- Jouer 2, 0, 3, 4, 5, 8 et vérifier que le joueur 2 l'emporte (diagonale \ )
- Jouer 1, 0, puis rentrer 'quit' et vérifier que le programme s'arrête sur un match nul
- Jouer 0, 0, et vérifier que le programme s'arrête avec une erreur indiquant qu'il est interdit de recouvrir un symbole déjà placé
- Jouer, 4, 1, 'oups' et vérifier que le programme affiche un message en continuant normalement

Bravo : le jeu peut maintenant sortir sur PS5. Le vrai gagnant c'est toi.

Annexe: snippets

Pour printer les nombres de 0 inclu à 9 inclu:

```
for i in range(0, 10):   # range(a, b) includes a but excludes b
    print(i)
```

Pour setter le premier élément d'une liste à "hey":

```
my_list = ["hi", "hello", "sup"]
my_list[0] = "hey"
print(my_list)  # -> ["hey", "hello", "sup"]
```

Pour déclencher une erreur qui stoppe tout le programme:

```
if something_is_wrong:
    raise ValueError("Something went wrong!")
```

Pour sortir d'une boucle:

```
while True:
    print("I'm in the loop")
    if i_want_to_get_out_of_the_loop:
        break

print("I broke out of the loop!")
```