Atelier 3 - KINXIP11 - Info1.Projet

Le cœur du jeu

Fraude

Le travail est individuel. Les programmes seront tous testés à l'aide de l'outil MOSS. S'il s'avère que des programmes sont identiques (ou très similaires), vous serez sanctionnés au delà du simple 0 à l'atelier. Nous comparerons également les programmes entre les ateliers 2, 3 et 4 afin de vérifier que vous n'avez pas copié le programme d'un pair sans faire le travail.

Sous Moodle, nous avons mis des capsules permettant de vous fournir des connaissances nécessaires pour ce projet. Lien : **Capsules**

1 Objectif

Programmer les différents déplacements de la règle du jeu, une fonction qui vérifie la fin de partie, une fonction qui programme le tour de jeu d'un joueur (mais pas une partie) et une fonction générale pour les tests unitaires.

Vous rendrez **un unique fichier .py** contenant les fonctions demandées et le code principal qui les appelle. Le programme doit être en python 3 et compréhensible par vos pairs.

2 Détails

Votre programme devra permettre de choisir d'exécuter le tour de jeu d'un joueur sur une configuration proposée, afficher le résultat de l'appel de fin de partie du 1 ou 2 exemples et d'appeler une fonction générale de tests.

Le programme doit reprendre les fonctionnalités demandées en atelier 2.

Il est possible de "s'inspirer" des ateliers 2 corrigés lors de l'évaluation par les pairs, **sans copier** voir **Fraude** ci-dessus.

3 Questions

3.1 Les déplacements et autres actions

Le travail de décomposition en fonctions (et de tests unitaires) est essentiel à cette étape.

Pour chaque type de déplacement donné, vous devez définir une ou plusieurs fonctions qui permettent d'effectuer ce déplacement et ses effets en fonction de la grille courante, du joueur courant, d'une case de départ et d'une case d'arrivée. Typiquement, une fonction `deplacement_X` (où X est le type de déplacement) devra :

— vérifier la validité du déplacement vis à vis de la règle (distance entre les cases, direction, cases de départ/intermédiaires/arrivée remplies ou vides...),

— si le déplacement est valide, effectuer le déplacement et appliquer les effets : mettre à jour la grille en ayant déplacé et/ou supprimé des pions.

Et enfin il est conseillé que cette fonction renvoie un booléen indiquant si le déplacement a pu être appliqué. Ce booléen pourra être utilisé lors du tour du jeu en cas de saisie invalide vis à vis de la règle (et donc de re-proposer un déplacement au joueur).

Attention : les saisies ne se font pas dans la fonction de déplacement.

Pour vérifier la validité des déplacements, vous aurez besoin de plusieurs petites fonctions. Voici quelques exemples :

— tester si le contenu d'une case est celui attendu. Utile pour vérifier s'il y a le pion du joueur dans la case de départ, ou s'il y a bien un pion dans la case intermédiaire entre le départ et l'arrivée ou encore si le pion est de la bonne couleur...
— tester la distance entre deux cases. Utile pour vérifier si la case de départ et celle d'arrivée sont à la bonne distance.
— tester la direction entre deux cases. Utile pour vérifier si la case de départ et celle d'arrivée sont bien orthogonales par exemple.
— ...

Pour ces fonctions de vérifications, des tests unitaires seront codés et mis dans une fonction générale et unique qui reprend tous les tests afin de faciliter le travail de l'évaluateur.

Remarque : La proposition d'enchaînement de déplacement ne se programme pas dans la fonction de déplacement.

3.2 Le tour de jeu d'un joueur

Vous devez définir une ou plusieurs fonctions qui permettent à un joueur d'effectuer un tour de jeu en fonction de la grille courante et du joueur courant. Attention : le tour d'un joueur n'est pas une partie !.

Grossièrement, un tour de jeu pour un joueur ressemble à :

— saisir le type de déplacement
— tant que le déplacement n'est pas valide :
  — saisir les coordonnées
  — appliquer le déplacement
— si le déplacement peut s'appliquer et qu'il peut être enchaîné, dans le même tour, le joueur doit pouvoir choisir de refaire ce type de déplacement. Tant que le joueur veut enchaîner :
  — tant que le déplacement n'est pas valide :
    — saisir les coordonnées
    — appliquer le déplacement
— passer au joueur suivant

Vous pouvez développer un autre algorithme pour le tour de jeu, celui-ci n'est qu'une possibilité.

3.3 La fin de partie

Vous devez définir une ou plusieurs fonctions qui vérifient si une partie est finie en fonction de l'état de la grille et du joueur courant. Cette fonction sera utilisée pour l'atelier 4.

N'oubliez pas d'ajouter une fonction de test pour toutes les fonctions que vous développez pour la fin de partie et de faire en sorte qu'elle soit appelée dans la fonction générale de tests.

3.4 Code principal

Votre code principal devra permettre de choisir de jouer un tour de jeu en configuration de début, milieu ou fin, ou de lancer le fonction générale de tests.

Pour les tours jeu d'un joueur, quelle que soit la configuration, le code doit permettre de choisir parmi tous les déplacements de la règle et de faire les enchaînements autorisés par la règle.

3.5 Tests unitaires

Vous aurez un grand nombre de tests à faire, commencez par les définir (test driven development) au fur et à mesure que vous aurez besoin de définir des fonctions. La fonction générale de tests sera appelée dans votre code principal (voir Tests Unitaires sous **Capsules**).

4 Evaluation

Voici une liste non exhaustive des critères sur lesquels vous serez évalués :

- Respect des consignes de dépôt (0,6/7) : anonymat, un seul fichier, pas d'erreur d'exécution, pas de code de la suite, pas uniquement atelier 2...
- Déplacements/Actions (1,9/7) :
  - correction fonctionnelle : le code doit répondre correctement aux fonctionnalités, le déplacement est correctement appliqué à la grille
  - la validité vis à vis des règles est bien vérifiées (distance, direction, cases vides/pleines...)
  - les erreurs de saisies du joueur sont gérées (pas d'interruption du programme) et le joueur peut resaisir tant qu'il se trompe. Vous pouvez indiquer au relecteur si votre code est incomplet, afin qu'il ne perde pas son temps à tester toutes vos fonctions inutilement.
- Tour de jeu (1,5/7) :
  - Il existe au moins une fonction qui permet de faire un tour de jeu
  - Respect des règles, pas d'erreur
  - Le dialogue avec le joueur est clair et pratique
- Fin de partie (1,2/7) : décomposition en fonctions, respect des règles, pas d'erreur...
- Tests (0,8/7) : la fonction de test générale est structurée et comprend des tests pour la majorité des fonctions
- Clean code (1,5/7) :
  - le nom des variables est explicite et respecte les conventions
  - le programme est composé de fonctions courtes et explicites
  - le programme est commenté pour améliorer la lisibilité et compréhension
  - il y a un code principal qui s'exécute au lancement
- ...

5 Conseils

5.1 Tests unitaires

En utilisant la fonction de tests, pensez à tester "unitairement" les fonctions que vous prévoyez, de manière à pouvoir les intégrer et utiliser en confiance par la suite. Une fois la fonction testée,

gardez les appels aux tests dédiés en commentaires, pour pouvoir les réactiver si besoin.

5.2 Itérations successives

D'une manière générale, et suivant votre aisance à programmer, il est conseillé de travailler par "itération successive". Par exemple d'abord écrire des fonctions de déplacement qui ne vérifient pas la validité vis à vis de la règle (on fait confiance aux joueurs) afin de pouvoir programmer le tour du jeu d'un joueur. Ainsi, dans la fonction de déplacement vous pouvez utiliser des fonctions auxiliaires de validité renvoyant toujours `True` que vous compléterez plus tard. Bien sûr n'oubliez pas de les compléter.

5.3 Décomposition

Pensez à bien décomposer en fonctions et à bien nommer notamment parce qu'un certain nombre de fonctions seront utiles pour la partie jeu ordinateur contre joueur.

5.4 Anticiper l'évolution

Pensez lorsque vous écrivez votre programme aux possibilités d'évolutions : déplacement supplémentaire, ou règle d'application d'un déplacement modifiée...

6 Rappels :

— pas d'objet
— pas de récursivité
— pas de try/except
— pas de bibliothèque non standard