Travail pratique # 3
Jeu de dames !

Jean-Francis Roy

Date de remise : Vendredi 21 mars 2014 à 23h00
Pondération de la note finale : 12%

1 Objectifs

Ce travail vous permettra de vous familiariser avec les structures de données principales de Python comme les listes et les dictionnaires, ainsi que la modélisation de programmes informatiques, via le paradigme orienté objet. Bien que vous ne verrez ce paradigme qu'après la semaine de lecture, cet énoncé et le code source joint présentent les éléments nécessaires à la réalisation de ce TP. Vous pouvez (devriez) donc le commencer dès maintenant !

2 Organisation du travail en équipe

Nous privilégions le travail en équipe car ça fait partie de votre formation académique et ça vous permettra de partager la charge du travail. Ceci dit, si certains veulent faire le TP individuellement, libre à eux de le faire. Aussi, pour ceux qui n'arrivent pas à trouver un coéquipier, je les invite à utiliser le forum du cours.

Chaque coéquipier doit contribuer à part égale au développement de ce TP. Laisser son coéquipier faire tout le travail (peu importe les raisons) est inacceptable : vous passerez à côté des objectifs de ce cours. De la même manière, il ne faut pas non plus trop en faire : il faut apprendre à travailler en équipe !

3 Problème à résoudre

Vous devez réaliser un jeu de dames simplifié, où deux joueurs s'affrontent. La modélisation dans le paradigme orienté objet est déjà réalisée pour vous : vous n'avez qu'à programmer le contenu des différentes fonctions, déjà documentées.

3.1 Le jeu de dames simplifié

La version simplifiée du jeu de dames que vous devez réaliser est un mélange du jeu de dames international et de la version anglaise *Checkers*. En voici les règles.

- Un *damier* de 8 cases par 8 contient 24 *pièces* (12 blanches, 12 noires). La position initiale est illustrée dans la figure 1 ;
- Le joueur avec les pièces blanches commence en premier ;
- Une pièce de départ est appelée un *pion*, et peut se déplacer **en diagonale** vers l'avant (vers le haut pour les blancs, vers le bas pour les noirs). Une case doit être libre pour pouvoir s'y déplacer ;
- Lorsqu'un pion atteint le côté opposé du plateau, il devient une *dame*. Cette action se nomme la *promotion*. Une dame a la particularité qu'elle peut aussi se déplacer vers l'arrière (toujours en diagonale) ;
- Une *prise* est l'action de "manger" une pièce adverse. Elle est effectuée en sautant par dessus la pièce adverse, toujours en diagonale, **vers l'avant ou l'arrière**. On ne peut sauter par dessus qu'une pièce adverse à la fois : il faut donc que la case d'arrivée soit libre ;
- Après une prise, le joueur courant peut effectuer une (ou plusieurs) prises supplémentaires en utilisant la même pièce ;
- Lors du tour d'un joueur, si celui-ci peut prendre une pièce ennemie, il doit absolument le faire (on ne peut pas déplacer une pièce s'il était possible d'effectuer une prise) ;
- On déduit des deux dernières règles que lorsqu'un joueur commence son tour et prend une pièce adverse, s'il peut continuer son tour en continuant de prendre des pièces adverses avec la même pièce, il **doit** le faire.

3.2 Le paradigme orienté objet

Nous avons vu en classe au module 5 que pour développer des programmes plus simples de maintenance, d'amélioration et d'extension, une bonne pratique est de séparer le programme en *modules*. Un paradigme de programmation offrant une solution à ce besoin est le *paradigme orienté objet*, dont nous allons présenter quelques composantes de base.

Un *objet* est une structure de données contenant des *attributs* (qui sont simplement des variables, aussi appelées *variables membres*), et qui répond à un ensemble de messages via des *méthodes* (qui sont simplement des fonctions qui agissent sur les variables membres). Imaginons par exemple un objet "pièce" du jeu de dames, qui aurait comme attributs une

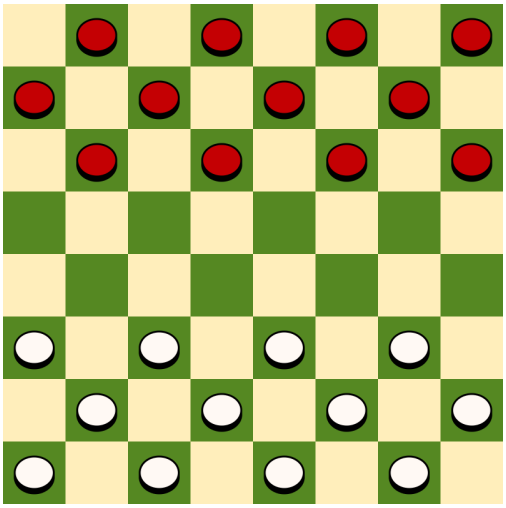

FIG. 1 : Position initiale du jeu de dames. Imaginez que les pièces rouges sont noires ! (source : Wikipédia)

couleur (blanc ou noir) et un type de pièce (pion ou dame). Un exemple de méthode serait "promouvoir()", une fonction modifiant le type de pièce pour transformer le pion en dame. Une autre méthode pourrait être "est_noir()", qui retourne vrai si la pièce est noire, et faux autrement.

Une *classe* est la définition d'un objet. Elle contient la définition du contenu de l'objet (attributs et méthodes), ainsi que le code des méthodes. Une classe peut être vue comme un "moule" à objets : on l'utilise pour créer de nouveaux objets.

Une *instance* est une instantiation d'une classe. Il s'agit donc d'un objet ayant des valeurs d'attributs qui lui sont propres. On peut créer un nombre quelconque d'instances d'une même classe, et chaque instance contiendra un espace mémoire qui lui est propre (des valeurs de variables membres qui lui sont propres). Par exemple, un damier contient au départ 24 pièces : on a donc au départ 24 instances de la classe pièce.

Vous trouverez joint à cet énoncé un exemple de classe en Python : la classe `Piece` qui est déjà programmée pour vous. Vous remarquerez que chaque méthode est une fonction qui contient un paramètre spécial nommé `self`, qui permet d'accéder aux attributs et méthodes de l'objet. Voici également un petit exemple de code orienté objet en Python. Il s'agit du même programme que vous avez fait dans le TD7, mais en version orientée objet.

```
class Personne :
    def __init__(self, nom, age, taille) :
        self.nom = nom
        self.age = age
        self.taille = taille

    def est_majeur(self) :
        return self.age >= 18

def consulter(dico) :
    while True :
        nom = input("Entrez le nom (ou <enter> pour terminer) : ")
        if nom == "" :
            break
        if nom in dico :                # le nom est—il répertorié ?
            item = dico[nom]            # consultation proprement dite
            age, taille = item.age, item.taille
            print("Nom : {} — âge : {} ans — taille : {} m.".format(
                  nom, age, taille))
        else :
            print("*** nom inconnu ! ***")

def remplir(dico) :
    while True :
        nom = input("Entrez le nom (ou <enter> pour terminer) : ")
        if nom == "" :
            break
        age = int(input("Entrez l'âge (nombre entier !) : "))
        taille = float(input("Entrez la taille (en mètres) : "))
        dico[nom] = Personne(nom, age, taille)
    return dico
```

3.3 La modélisation de la solution

La solution est modélisée en 3 classes : une classe `Piece`, donnée en exemple plus haut, une classe `Damier` modélisant le damier du jeu, et la classe `Partie`, modélisant une partie de dames entre deux joueurs. Un fichier par classe est fourni, et ceux-ci sont à compléter. La modélisation étant déjà faite pour vous, nous avons inclus en commentaires une énorme quantité d'information expliquant ce qu'il y a à faire pour chaque méthode. Quelques méthodes sont déjà programmées pour vous, et le reste est à faire. Pour programmer le code d'une méthode, vous n'avez qu'à remplacer l'instruction `pass` qui ne fait rien, par votre code.

Les section suivantes donnent un aperçu du de la décomposition du travail. Vous trouverez dans les fichiers joints plus d'indications pour chaque méthode à programmer.

3.3.1 La classe `Piece`, déjà programmée pour vous à titre d'exemple

```
class Piece :
    """
    Classe modélisant une pièce d'un jeu de dames.
    """
    def __init__(self, couleur, type_de_piece) :
        """
        La méthode spéciale __init__ d'une classe est appelée lorsqu'on
        instancie un nouvel objet. Elle peut prendre des paramètres
        supplémentaires (ici, "couleur" et "type_de_piece"), qui sont
        les paramètres nécessaires lorsqu'on crée un nouvel objet. Le
        mot clé "self" permet de stocker des informations dans
        l'instance de l'objet. Chaque instance a son propre espace
        mémoire et peut donc contenir des valeurs différentes dans ses
        variables membres.

         :param couleur : couleur de la pièce ("blanc", "noir").
         :type couleur : string.
         :param type_de_piece : type de pièce ("pion", "dame").
         :type type_de_piece : string.
        """
        pass

    def est_pion(self) :
        """
        Retourne si la pièce est un pion.

         :return : True si la pièce est un pion, False autrement.
        """
        pass

    def est_dame(self) :
        """
        Retourne si la pièce est une dame.

         :return : True si la pièce est une dame, False autrement.
        """
        pass

    def est_blanc(self) :
        """
```

```
        Retourne si la pièce est de couleur blanche.

        :return : True si la pièce est de couleur blanche, False
                  autrement.
        """
        pass

    def est_noir(self) :
        """
        Retourne si la pièce est de couleur noire.

        :return : True si la pièce est de couleur noire, False autrement.
        """
        pass

    def promouvoir(self) :
        """
        Cette méthode permet de "promouvoir" une pièce, c'est à dire la
        transformer en dame.
        """
        pass
```

3.3.2 La classe `Damier`

```
class Damier :
    """
    Classe représentant le damier d'un jeu de dames.
    """

    def __init__(self) :
        """
        Méthode spéciale initialisant un nouveau damier.
        """
        # Dictionnaire de cases. La clé est une position
        # (ligne, colonne), et la valeur une instance de la
        # classe Piece.
        self.cases = {}

    def get_piece(self, position) :
        """
        Récupère une pièce dans le damier.

        :param position : La position où récupérer la pièce.
```

```
        :type position : Tuple de coordonnées matricielles
                         (ligne, colonne).
        :return : La pièce à cette position s'il y en a une, None
                  autrement.
        """
        pass

    def position_valide(self, position) :
        """
        Vérifie si une position est valide (chaque coordonnée doit être
        dans les bornes).

        :param position : Un couple (ligne, colonne).
        :type position : tuple de deux éléments
        :return : True si la position est valide, False autrement
        """
        pass

    def lister_deplacements_possibles_a_partir_de_position(self,
                                        position, doit_prendre=False) :
        """
        Cette méthode retourne une liste de positions qui sont
        accessibles par une pièce qui est placée sur la position reçue
        en paramètres. Un paramètre "doit_prendre" indique si la liste
        des positions retournée ne doit contenir que les positions
        résultant de la prise d'une autre pièce.

        :param position : La position de départ du déplacement.
        :type position : Tuple de deux éléments (ligne, colonne)
        :param doit_prendre : Indique si oui ou non on force la liste
                              de positions à ne contenir que les
                              déplacements résultants de la prise d'une
                              pièce adverse.
        :type doit_prendre : Booléen.
        :return : Une liste de positions où il est possible de se
                  déplacer depuis la position "position".
        """
        pass

    def lister_deplacements_possibles_de_couleur(self, couleur,
                                                 doit_prendre=False) :
        """
        Fonction retournant la liste des positions (déplacements)
        possibles des pièces d'une certaine couleur. Encore une fois,
        un paramètre permet d'indiquer si on ne désire que les
```

```
        positions résultant de la prise d'une pièce adverse.

        :param couleur : La couleur ("blanc", "noir") des pièces dont on
                         considère le déplacement.
        :type couleur : string
        :param doit_prendre : Indique si oui ou non on force la liste de
                              positions à ne contenir que les
                              déplacements résultants de la prise d'une
                              pièce adverse.
        :return : Une liste de positions où les pièces de couleur
                  "couleur" peuvent de se déplacer.
        """
        pass

    def deplacer(self, position_source, position_cible) :
        """
        Effectue un déplacement sur le damier. Si le déplacement est
        valide, on doit mettre à jour le dictionnaire self.cases, en
        déplaçant la pièce à sa nouvelle position.

        :param position_source : La position source du déplacement.
        :type position_source : Tuple (ligne, colonne).
        :param position_cible : La position cible du déplacement.
        :type position_cible : Tuple (ligne, colonne).
        :return : "ok" si le déplacement a été effectué sans prise,
                  "prise" si une pièce adverse a été prise, et
                  "erreur" autrement.
        """
        pass

    def convertir_en_chaine(self) :
        """
        Retourne une chaîne de caractères où chaque case est écrite
        sur une ligne distincte. Chaque ligne contient l'information
        suivante  : ligne,colonne,couleur,type

        Cette méthode pourrait par la suite être réutilisée pour
        sauvegarder un damier dans un fichier.

        :return : La chaîne de caractères.
        """
        pass

    def charger_dune_chaine(self, chaine) :
        """
```

```
        Remplit le damier à partir d'une chaîne de caractères
        comportant l'information d'une pièce sur chaque ligne. Chaque
        ligne contient l'information suivante :
        ligne,colonne,couleur,type

        :param chaine : La chaîne de caractères.
        :type chaine : string
        """
        pass
```

3.3.3 La classe `Partie`

```
class Partie :
    def __init__(self) :
        """
        Méthode d'initialisation d'une partie. On initialise 4 membres :
        — damier : contient le damier de la partie, celui—ci contenant
          le dictionnaire de pièces.
        — couleur_joueur_courant : le joueur à qui c'est le tour de
          jouer.
        — doit_prendre : un booléen représentant si le joueur actif
          doit absoluement effectuer une prise de pièce.
        — position_source_forcee : Une position avec laquelle le joueur
          actif doit absoluement jouer. Le seul moment où cette
          position est utilisée est après une prise : si le joueur peut
          encore prendre d'autres pièces adverses, il doit absolument
          le faire. Ce membre contient None si aucune position n'est
          forcée.
        """
        pass

    def valider_position_source(self, position_source) :
        """
        Vérifie la validité de la position source.

        :param position_source : La position source à valider.
        :return : Un couple où le premier élément représente la validité
                  de la position (True ou False), et le euxième élément
                  est un éventuel message d'erreur.
        """
        pass

    def valider_position_cible(self, position_source, position_cible) :
```

```
        """
        Vérifie si oui ou non la position cible est valide, en fonction
        de la position source.

        :return : Un couple où le premier élément représente la validité
                  de la position (True ou False), et le deuxième élément
                  est un éventuel message d'erreur.
        """
        pass

    def demander_positions_deplacement(self) :
        """
        Demande à l'utilisateur les positions sources et cible, et
        valide ces positions.

        :return : Un couple de deux positions (source et cible). Chaque
                  position est un couple (ligne, colonne).
        """
        pass

    def tour(self) :
        """
        Cette méthode simule le tour d'un joueur.
        """
        pass

    def jouer(self) :
        """
        Démarre une partie. Tant que le joueur courant a des
        déplacements possibles (utilisez la méthode appriopriée !),
        un nouveau tour est joué.

        :return : La couleur ("noir", "blanc") du joueur gagnant.
        """
        pass

    def sauvegarder(self, nom_fichier) :
        """
        Sauvegarde une partie dans un fichier.

        :param nom_fichier : Le nom du fichier où sauvegarder.
        :type nom_fichier : string.
        """
        pass
```

```
    def charger(self, nom_fichier) :
        """
        Charge une partie dans à partir d'un fichier.

         :param nom_fichier : Le nom du fichier à charger.
         :type nom_fichier : string.
        """
        pass
```

4 Ce que vous devez rendre

Vous devez rendre un fichier `.zip` comportant uniquement les fichiers suivants :

- Le fichier `__init__.py`, sans modification ;
- Le fichier `piece.py` sans modification ;
- Le fichier `damier.py`, complété ;
- Le fichier `partie.py`, complété ;
- Le fichier de barême, complété avec vos noms.

Le nom du `.zip` doit respecter le format suivant : TP3_Matricule.zip où Matricule est le numéro matricule de l'un des membres de l'équipe.

Note : afin de simplifier la correction, veuillez remettre ces fichiers directement dans une archive zip, et non dans un dossier ou sous-dossier.

La remise de ce TP doit se faire via le lien "remise de travaux" du site Web qui vous redirige vers votre guichet étudiant (aucune remise par courriel n'est acceptée). Il est important de respecter la date et l'heure de la remise, car tout travail remis en retard sera attribué la note 0. Veuillez aussi noter que Pixel n'autorise aucun retard et assurez-vous après la remise d'avoir remis le bon fichier rempli et que Pixel vous a bien confirmé la remise par courriel. Tout manquement à ces vérifications pourrait entraîner une note de 0 à ce travail.

5 Remarques

Qualité du code Certains étudiants ont remis aux travaux pratiques 1 et 2 du code qui n'est pas exécutable. Une raison commune est que l'indentation des fichiers n'est pas correcte. Un programme Python ne peut pas être exécuté si l'indentation est incorrecte : ce n'est pas un programme valide ! Il est inacceptable de remettre un programme qui ne peut pas être exécuté : c'est un signe que vous n'avez jamais testé votre programmation, ou que vous avez copié-collé du code d'un autre étudiant, ce qui est du plagiat.

Plagiat : Tel que décrit dans le plan de cours, le plagiat est interdit. Une politique stricte de tolérance zéro est appliquée en tout temps et sous toutes circonstances. Tous les cas seront référés à la direction du Département.

6 Les points importants qui seront vérifiés lors de la correction

- Le fonctionnement de votre implantation ;
- Le respect des prototypes des méthodes de classe ;
- La qualité du code remis ;
- Le respect des biens livrables.

7 Évaluation

Le travail est noté sur 100 points. Chaque méthode implantée vaudra un certain nombre de points qui sera déterminé en fonction de la difficulté qu'elle représente, puis une partie des points sera attribuée à la qualité du code. Un barème plus détaillé est disponible en pièce jointe à ce document.

Bon travail !