

Aide jalon 02

mmc

marc-michel dot corsini at u-bordeaux dot fr

Rev. 2 : 17 Fevrier 2022

1 Créer le fichier `players.py`

Je ne pensais pas que ce soit un problème. J'ai déjà signalé que votre zone de jeu était le répertoire Code vous ne **devez** pas aller ailleurs pour travailler, ce qui évitera pas mal de « ça ne marche pas ». Pour recopier un fichier, il y a plein de manière de procéder

- Depuis un terminal, vous utilisez la commande `copy` (sous windows) ou `cp` sous Mac ou Linux
- Avec la souris, vous utilisez le menu contextuel « copy »
- Avec votre outil de programmation (idle, notepad++, spyder3, vscode, ...). Vous ouvrez le fichier est vous utiliser la commande « enregistrer sous » ou « save as »

Bref, ...

2 Généralités

On vous demande de créer des joueurs numériques, afin de pouvoir jouer à un jeu. Ces joueurs numériques sont des classes qui vont dériver (hériter) d'une classe déjà construite `Player` permettant de garantir que toutes les fonctionnalités nécessaires soient présentes.

Pour indiquer qu'une classe dérive B d'une autre classe A, on utilise, en **python** la syntaxe

```
class B(A):
```

L'intérêt de ce processus d'héritage, est de minimiser le travail du programmeur. Il va se contenter de spécifier les « comportements » (c'est-à-dire les méthodes ou attributs) différents ou supplémentaires de la classe B par rapport à sa classe parente A.

Pour les différents types de joueurs, il va falloir définir la méthode `decision` dont le but est de renvoyer l'action que souhaite faire le joueur en fonction de l'état du jeu.

- `self` à l'intérieur de la classe va permettre d'accéder à toutes les informations disponibles pour un joueur

- `self.game` à l'intérieur de la classe va permettre d'accéder à toutes les informations disponibles sur le jeu auquel le joueur joue.

On vous demande de démarrer votre code de la méthode `decision` par la séquence d'instructions :

```
def decision(self, state):
    self.game.state = state
    if self.game.turn != self.who_am_i:
        print("not my turn to play")
        return None
    # a moi de jouer
```

la méthode `decision` reçoit donc une information extérieure (le paramètre `state`). La première instruction stocke la valeur dans `self.game.state` c'est à dire dans l'attribut `state`, de l'attribut `game` du joueur. Dit autrement, le joueur dispose de sa propre version du jeu, sur laquelle il va pouvoir travailler.

La seconde instruction compare la valeur de `self.game.turn` (qui correspond à savoir à qui est le tour de jeu) avec la valeur de `self.who_am_i` (qui correspond au rôle du joueur) – dit autrement `self.game.turn` dit « c'est à 'X' » de jouer, tandis que `self.who_am_i` dit « je suis le joueur 'X' (ou 'O') ». Si les deux valeurs ne correspondent pas, le joueur numérique renvoie `None` indiquant qu'il n'a pas à prendre de décision puisque ce n'est pas son tour de jeu.

Si les deux valeurs `self.game.turn` et `self.who_am_i` sont identiques il faut renvoyer l'action que le joueur numérique souhaite faire.

Les actions d'un jeu, sont contenues dans l'attribut `actions` du jeu, et donc la méthode `decision` de **tous les joueurs numériques** doit renvoyer un élément appartenant à cet attribut.

3 Rappels

Quelques points importants

1. On ne crée pas une nouvelle classe tant que la première n'est pas complètement faite et validée par les tests fournis. L'ordre des classes est voulu, pour vous permettre d'aller du simple au complexe, vouloir faire `MinMax` alors que vous n'avez pas réussi à faire `Human` c'est comme vouloir faire le marathon de New York alors que vous n'avez pas le droit de quitter le territoire national.
2. Prenez l'habitude de tester vos codes, pas uniquement avec les testcodes fournis ou les tests de `main_tests` mais directement dans le shell. Pour cela vous pouvez vous inspirer des testcodes, les étapes sont simples :
 - importer la classe du jeu `from ... import`
 - créer le jeu
 - créer le joueur en lui donnant une chaîne de caractères ou un entier qui correspondra à son nom, et le jeu

- indiquer si le joueur est le premier ou le second joueur

```
>>> from hexapawn import Hexapawn
>>> jeu = Hexapawn(4,4) # version simple
>>> moi = Human('tyty', jeu) # un joueur interface pour hexapawn
>>> moi.who_am_i = jeu.opponent # je joue en second, jeu.turn pour commencer
```

Vous pouvez alors afficher le jeu directement ou afficher le jeu du joueur, de même que vous pouvez consulter les actions possibles ou l'état du jeu ou si la partie est terminée

```
>>> print(jeu)
>>> print(jeu.actions)
>>> print(jeu.state)
>>> print(jeu.over())
>>> print(moi.game)
>>> print(moi.game.actions)
>>> print(moi.game.state)
>>> print(moi.game.over())
```

3. Les noms des classes sont obligés, `Human` et pas `Humain`, `Randy` et pas `Random`, etc

4 Human

Pour cette classe la difficulté réside dans le fait qu'il faut qu'elle soit fonctionnelle quelque soit le jeu. C'est-à-dire que l'on ne veut pas être dépendant de la syntaxe des actions.

1. Pour le jeu des allumettes `classe Matches`, les actions sont des chiffres.
2. Pour le jeu des boîtes `classe Divide`, les actions sont des paires constituées d'une lettre et d'un chiffre.
3. Pour le jeu hexapawn `classe Hexapawn`, les actions sont des paires de couple, chaque couple étant un numéro de ligne et un numéro de colonne
4. Pour le jeu du morpion `classe Morpion`, les actions sont des couples constitué d'un numéro de ligne et d'un numéro de colonne.

De plus, on veut être certain que l'action retournée est valide c'est-à-dire qu'il s'agit bien d'une des actions autorisées par le jeu. Hors, on sait que les actions autorisées appartiennent à `actions` qui est un attribut des objets de la classe `Game`.

4.1 Où trouver les actions autorisées ?

Très simplement, elles sont dans `self.game.actions`. Cet attribut est un tuple (qui est utilisable comme une liste grâce aux crochets). Il suffit donc de ne manipuler les actions qu'en fonction de leur position dans cette structure. Par exemple, pour dire que l'on souhaite la première des actions possibles on écrira

```
self.game.actions[0]
```

Tout comme les listes **python**, on peut connaître la taille d'un tuple grâce à la commande `len`

4.2 Que faut-il faire ?

Il suffit de demander à l'utilisateur de choisir le numéro de l'action à effectuer en utilisant la commande `input`. Cette commande prend en paramètre un message qui sera afficher à l'écran pour l'utilisateur, **et renvoie** une chaîne de caractères. Il faudra donc s'assurer que la réponse de l'utilisateur est transformable en un entier.

Les chaînes de caractères disposent de nombreuses méthodes, pour les connaître on tape dans un shell python la commande `help(str)`¹, **python** utilise des commandes de la forme `isXXX()` permettant de savoir (réponse `True` ou `False`) si une entité possède la caractéristique `XXX`

L'algorithme est donc

Afficher pour chaque action l'entier que doit saisir l'utilisateur

```
rep = ''
```

Tant que rep n'est pas un nombre ou que `int(rep)` n'est pas un numero autorise

```
    rep = input("Quel est votre choix ? ")
```

trouver l'action a partir de son numero

```
renvoyer action
```

Dans un shell on peut tester des exemples simples ...

```
>>> actions = [ 'a_1', 2, 3, ((0,1),(0,2)), (2,3), ('A', 3) ]
```

```
>>> rep = input("Quel est votre choix ? ")
```

```
Quel est votre choix ? a
```

```
>>> rep
```

```
'a'
```

```
>>> rep = input("Quel est votre choix ? ")
```

```
Quel est votre choix ? 1
```

```
>>> rep
```

```
'1'
```

```
>>> rep.isnumeric()
```

```
True
```

```
>>> rep.isdigit()
```

```
True
```

```
>>> rep.isdecimal()
```

```
True
```

```
>>> rep = '1a'
```

```
>>> rep.isdigit()
```

```
False
```

```
>>> rep = '15'
```

```
>>> rep.isdigit()
```

¹help s'applique à n'importe quelle classe. Par exemple `help(Hexapawn)`

```

True
>>> rep = int(rep)
>>> rep
15
>>> actions
['a_1', 2, 3, ((0, 1), (0, 2)), (2, 3), ('A', 3)]
>>> actions[rep]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> actions[0]
'a_1'
>>> actions[3]
((0, 1), (0, 2))
>>> actions[5]
('A', 3)

```

5 Randy

L'objectif est ici de choisir aléatoirement une action parmi les actions autorisées. En début du fichier on a fait `import random`, il suffit donc d'utiliser une méthode de `random` pour effectuer ce choix. Faites `help(random)` dans le shell pour choisir la méthode la plus appropriée.

6 MinMax

Un algorithme n'est pas un programme, c'est une description *abstraite* d'un processus. vous devez faire un travail de transformation entre l'algorithme et l'implémentation.

Un algorithme ne dépend pas d'un style de programmation, il y a donc un travail de compréhension pour passer d'une description algorithmique à sa réalisation.

L'algorithme fait état de 3 fonctions `choix`, `eval_min`, `eval_max`. Il faut d'abord s'occuper de leurs rôles et de leurs signatures

- `choix` cherche une action, c'est la fonction principale. Elle appelle `eval_min`
- `eval_min` cherche une valeur numérique minimale, elle appelle `eval_max`
- `eval_max` cherche une valeur numérique maximale, elle appelle `eval_min`

Ces fonctions servent à parcourir un arbre en profondeur d'abord (se reporter au cours sur min-max). Mais, au lieu de parcourir un arbre statique, on va le construire dynamiquement ce qui permet de minimiser l'espace mémoire occupé. Dans un arbre statique, tout l'arbre est disponible en mémoire, dans une construction dynamique, seule la branche² sur laquelle on travaille est présente en mémoire.

`choix` s'applique à la racine de l'arbre, `eval_min` s'applique aux sommets gérés par l'adversaire

²une branche est une chaîne joignant la racine à un sommet de l'arbre.

du joueur « racine ». Tandis que `eval_max` s'applique aux sommets gérés par le joueur « racine ». L'algorithme de `choix` cherche donc l'action qui correspond à la meilleure évaluation renvoyée par la fonction `eval_min`. Pour trouver cette action, on peut soit stocker ces valeurs, puis chercher l'optimum (algorithme 1), ou bien on peut garder à tout moment la meilleure action et la meilleure valeur associée (algorithme 2). Cette seconde version est moins couteuse en mémoire.

```
def choix(s):
    valeurs structure de stockage des valeurs
    actions_vues structure de stockage des actions evaluees
    pour chaque action a de s faire
        stocker a dans actions_vues
        calculer s' à partir de s et a
        stocker eval_min(s', pf-1) dans valeurs
    trouver a_best dans actions_vues en fonction de valeurs
    retourner a_best
```

FIGURE 1 : Algo : stocker et chercher

```
def choix(s):
    v_best = - infini # -1000 suffira ici
    a_best = None
    pour chaque action a de s faire
        calculer s' à partir de s et a
        v = eval_min(s', pf-1)
        si v > v_best
            v_best = v
            a_best = a
    retourner a_best
```

FIGURE 2 : Algo : garder à jour

6.1 Réalisation

On impose une méthode `decision` dont le but est de trouver l'action à réaliser. Elle correspond donc à la fonction `choix`. On a donc besoin de deux fonctions annexes, qui doivent être privées il suffit donc de leur donner un nom qui commence par `__`. Nos trois méthodes sont donc `decision`, `__eval_min` et `__eval_max`.

- Les paramètres de `decision` sont imposés, il s'agit de `self` et `state`. Puisque on doit faire en première instruction (voir fiche) `self.game.state = state` l'état du jeu est donc accessible par l'attribut `self.game.state`. Lorsque l'on utilise `self.game.move(a)` cela modifie l'état du jeu. L'opération inverse est `self.game.undo()`. Les actions autorisées sont contenues dans `self.game.actions`. La profondeur est fournie dans le constructeur, pour y accéder il faut faire `self.get_value('pf')`.

Et, on doit le faire une seule fois.

```
def decision(self, state):
    self.game.state = state # mise à jour de l'état initial
    # traitement bon joueur oui/non
    if self.game.turn != self.who_am_i:
        print("not my turn to play")
        return None
    # récupération du paramètre
    pf = self.get_value('pf')
    # recherche de la meilleure action
    # < ici on fait la mise en oeuvre de l'algorithme choix >
    return meilleure_action
```

- **Attention** Dans un jeu `move(a)` ne renvoie rien, par contre elle modifie l'attribut `state` du jeu

```
>>> from allumettes import Matches
>>> jeu = Matches(17)
>>> print(jeu.state)
(17, 0)
>>> print(jeu.move(2))
None
>>> print(jeu.state)
(15, 1)
>>>
```

De la même manière, dans un jeu `undo()` ne renvoie rien, mais modifie si c'est possible, l'attribut `state` du jeu

```
>>> type(jeu)
<class 'allumettes.Matches'>
>>> jeu.state
(15, 1)
>>> print(jeu.undo())
None
>>> jeu.state
(17, 0)
>>> print(jeu.undo()) # sans effet sur le jeu
None
>>> jeu.state
(17, 0)
>>>
```

- Les paramètres de `__eval_min` et `__eval_max` puisque ce sont des méthodes de la classe `MinMax` elles ont comme premier paramètre `self` qui va permettre d'accéder à toutes les informations utiles et, en particulier à l'état du jeu qui est dans `self.game.state`. Il est donc **inutile** de garder le paramètre "s", on a juste besoin du paramètre "pf" puisqu'il est modifié à chaque appel. Par ailleurs, pour les appels à ces méthodes on écrira simplement `self.__eval_min(pf -1)` ou `self.__eval_max(pf -1)`
- Pour pouvoir examiner toutes les actions autorisées pour un certain état du jeu, il suffira de faire une boucle `for`

```
for a in self.game.actions:
    self.game.move(a) # change l'état s -> (s,a)
    v = self.__eval_XXX(pf -1) # on appelle min ou max
    traitement de la valeur v
    self.game.undo() # retour à l'état s
```

Important remarquez bien qu'en début de boucle on utilise `move` et qu'en fin de boucle on utilise `undo` – regardez la fiche de cours sur `MinMax`

- Quand est-ce que l'on est sur une « feuille » ?
Soit lorsque la partie est terminée `self.game.over()`, soit lorsque l'on a atteint la profondeur maximale autorisée donc `pf == 0`. L'instruction est donc

```
if self.game.over() or pf == 0:
    return self.estimated()
```