Fiche Jalon 01 : Pas à pas

mmc
marc-michel dot corsini at u-bordeaux dot fr

31 Janvier 2022

1 Quelques éléments sur le travail

La fonction dans laquelle on travaille est définie par

```
def valid_state(self, cfg:tuple) -> bool:
```

Cela signifie que dans cette fonction on accède aux informations associées à `self` et à `cfg`. On sait que l'on reçoit un `tuple` qui est comme une liste python, mais dont on ne peut pas modifier le contenu. Puisque c'est un tuple, on peut accéder aux valeurs qu'il contient en utilisant la notation `[i]` où *i* désigne une position. Par exemple :

```
>>> c = 1,2,3
>>> print(c)
(1, 2, 3)
>>> c[0]
1
>>> c[1]
2
```

2 Ce qu'il faut faire

On nous demande de vérifier 8 points, 4 dans la partie « Quelque soit le jeu choisi » et 4 dans la partie « Ce qu'il faut faire dans le jeu HexaPawn ». Si ces 8 points sont vérifiés il faut renvoyer `True` sinon on renverra `False` Les valeurs True (vrai) et False (faux) sont des booléens. Pour résoudre ce problème, il est plus facile de vérifier chaque point, ce qui permet d'organiser le code de la manière suivante :

```
if point_1 est faux: return False
if point_2 est faux: return False
if point_3 est faux: return False
if point_4 est faux: return False
if point_5 est faux: return False
if point_6 est faux: return False
if point_7 est faux: return False
if point_8 est faux: return False
return True
```

Maintenant qu'on sait comment le code sera organisé on peut traiter chacun des points les uns à la suite des autres.

1. Pour le point 1, on nous demande de nous assurer que le tuple fourni est de taille 2. Le tuple c'est la variable `cfg`. Pour connaître la taille d'un tuple on utilise la commande `len()` Pour l'exemple de la section 1 cela donne

```
>>> len(c)
3
```

Si on veut tester si la longueur a une valeur particulière on utilise == pour un test d'égalité et != pour un test de dis-égalité

```
>>> len(c) == 3
True
>>> len(c) != 2
True
>>> len(c) != 3
False
```

2. Pour le point 2, on nous demande de vérifier que la première valeur est une chaîne de caractères. On sait que la première valeur dans un tuple est en position 0. Pour savoir si c'est une chaîne de caractères on utilise la commande `isinstance()` qui prend deux informations. En premier la valeur que l'on veut tester, en second le type supposé de cette valeur. Le « type » en python c'est par exemple `bool` pour un booléen, `int` pour une valeur entière, `float` pour un réel, ou `str` pour une chaîne de caractères. Voir la fiche `aide_jalon_01.pdf`

3. Le point 3, demande de vérifier que le second élément du tuple est un entier, c'est exactement la même méthode de résolution

4. Le point 4 demande de vérifier que dans la chaîne de caractères il n'y a que des élements qui sont dans `self.PAWN` Il faut donc par courir la chaîne, et regarder, pour chaque valeur si elle est dans les éléments autorisés. Pour accéder aux éléments d'une chaîne, on utilise une boucle `for`

```
>>> for x in mot:
...     print("la lettre est", x)
...
la lettre est b
la lettre est o
la lettre est n
la lettre est s
la lettre est o
la lettre est i
la lettre est r
```

Pour savoir si une valeur est autorisée, on utilise la commande `in`

```
>>> ok = "axn"
>>> for x in mot:
...     if x in ok: print(x,"est dans", ok)
...     else: print(x,"n'est pas autorisé")
...
b n'est pas autorisé
o n'est pas autorisé
n est dans axn
s n'est pas autorisé
o n'est pas autorisé
i n'est pas autorisé
r n'est pas autorisé
```

Nous on ne veut pas faire des print, on veut juste renvoyer `False` si ce n'est pas autorisé, on va donc écrire

```
if x not in self.PAWN: return False
```

5. Le point 5, consiste à vérifier « Qu'il n'y a pas plus de pions d'une couleur que de colonnes ». Il faut donc récupérer le nombre de colonnes. Pour cela, on va interroger `self` pour savoir quel est le nombre de colonnes.
`self` peut nous donner, le nombre de lignes 'nbl', le nombre de colonnes 'nbc', si le terrain est un cylindre 'cylindre', et s'il est prioritaire de manger les pions adverses 'priorite', en utilisant la commande `get_parameter`, il peut nous donner aussi les pions utilisés qui sont

stockés dans la variable `self.PAWN` au position 1 (pour le premier joueur), 2 pour le second joueur.

```
self.get_parameter('nbl')
self.get_parameter('nbc')
self.get_parameter('cylindre')
self.get_parameter('prioirite')
self.PAW[1]
self.PAW[2]
```

Bien entendu, si on veut, on peut créer des variables locales dans notre fonction, afin de stocker cette information. L'intérêt des variables locales c'est d'être moins longues à écrire que le nom complet par exemple

```
nbl = self.get_parameter('nbl')
nbc = self.get_parameter('nbc')
X = self.PAW[1]
O = self.PAW[2]
```

On pourra utiliser les noms courts `nbl, nbc, X, O` au lieu des noms longs quand on en a besoin. **Attention** une variable locale, veut dire qu'à l'extérieur de la fonction la variable n'existe pas

```
>>> def ma_fonction(p):
...     x = 42
...     y = 13
...     return x+(y*p)
...
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
>>> ma_fonction(2)
68
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Pour savoir quel est le nombre de fois où une lettre particulière est dans un mot on utilise la commande `count()` Par exemple on pourra écrire

```
>>> mot = "bonsoir"
>>> lettre = 'o'
>>> mot.count(lettre)
2
>>> mot.count('o')
2
>>> mot.count("o")
2
>>> "bonsoir".count(lettre)
2
```

Combien y a t-il de fois la lettre « o » dans le mot « bonsoir ». Mettre "o" ou 'o' c'est pareil (en python), mettre une variable locale ou bien son contenu a le même effet. En réalité, python au moment de l'évaluation remplace chaque variable par sa valeur d'où un résultat identique si on fait le travail pour lui.

6. Pour le point 6. On nous demande de vérifier qu'il n'y a pas plus d'un pion qui a atteint le camp adverse. La raison est que lorsqu'un joueur a réussi à mettre un pion dans le camp adverse, la partie est arrêtée, il est donc impossible qu'il y en ai plus d'un.
   Pour cela il faut être capable de récupérer dans `cfg[0]` l'information sur le camp adverse. Comme `cfg[0]` est une chaîne de cartactères où sont mis bout à bout les lignes du damier, on utilise la technique de découpage (on dit « slicing » en **python**). Cette technique va utiliser la syntaxe `[i:j:k]` qui signifie que l'on va prendre les informations en positions i jusqu'à j non compris, par pas de k, c'est-à-dire , successivement les positions $i, i+k, i+2k \ldots$.

Sachant qu'un damier de nl lignes et nc colonnes, va être stocké dans une chaîne de $nl \times nc$ caractères, on aura que la sous-chaîne `[0:nc]` va contenir les `nc` premières informations (soit la première ligne). Pour vous simplifier la vie, j'ai mis dans le fichier `outils.py` du répertoire `tools` des petites fonctions qui automatisent le découpage d'une chaîne de caractères en renvoyant les sous-chaînes correspondant aux lignes et aux colonnes (il y a aussi des fonctions pour les diagonales, mais elles ne sont pas interessantes pour le jeu hexapawn).

(a) `lines` prend en premier paramètre une chaîne de caractères à découper, en second le nombre de lignes (nbl) et en troisième le nombre de colonnes (nbc). Et renvoie la liste des sous-chaînes de longueur nbc (il y en aura nbl dans la liste)

(b) `columns` prend les mêmes paramètres mais cette fois renvoie les informations par colonnes, il y aura en sortie une liste de nbc sous-chaînes, chacune de longueur nbl.

Pour utiliser la fonction `lines` il faudra rajouter dans votre fichier au tout début la commande

```
from tools.outils import lines
```

La ligne `0` et la ligne `-1` sont les lignes à examiner puisqu'elles correspondent respectivement au camp de base de `self.PAWN[1]` et `self.PAWN[2]`

7. Le point suivant, va s'assurer que l'on n'a pas à la fois un pion « gagnant » dans chacun des camps, il faut juste s'assurer que la somme des pions « étrangers » est bornée par 1.

8. Le dernier point est le plus délicat. On doit s'assurer que les déplacements que l'on peut constater sur le terrain sont compatibles avec le nombre fourni par `cfg[1]` Un pion qui n'est pas dans son camp de base, est un pion qui a bougé, on va donc compter le nombre total de déplacements observables et s'assurer que ce total est inférieur ou égal à la valeur `cfg[1]`