Aide jalon 03

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

Rev. 1 : 22 Fevrier 2022

1 Liste et Dictionnaire, vademecum

En Python, listes (list) et dictionnaires (dict) sont des containers, c'est-à-dire des structures de stockage, on peut donc les créer, ajouter un élément, supprimer un élément. De plus ces structures sont « itérables », c'est-à-dire que l'on peut les parcourir. Les listes peuvent être vues comme des structures contigües et indexées par des entiers (à partir de 0), contrairement aux dictionnaires.

1.1 Création

```
L = [] # creation d'une liste vide
D = {} # creation d'un dictionnaire vide
```

1.2 Ajout

Deux méthodes d'ajout dans les listes

```
L.append(v) # ajout en fin
L.insert(p, v) # ajout a la position p
```

Pour ajouter dans un dictionnaire

```
D[clef] = v
```

Une clef est une entité qui peut être transformée par une fonction de « hashage » en un nombre. Les clefs qui se prêtent facilement à cette transformation sont les nombres et les chaînes de caractères

```
>>> D['toto'] = 42
>>> D[-2.5] = 'coucou'
```

Une liste, ou un dictionnaire peuvent servir à stocker n'importe quelle information, n'importe quelle structure de données

1.3 Suppression

Pour supprimer une valeur dans une liste on utilise la commande remove, pour supprimer une valeur dans un dictionnaire on utilise la commande del

```
L.remove(v)
del D[clef]
```

1.4 Parcours

Dans ce qui suit, `gestion_clef` et `gestion_valeur` désignent n'importe quel code travaillant sur l'information passée en paramètre. On dispose de 3 manières pour parcourir aisément une liste

```
for i in range(len(L)):
    gestion_clef(i)
    gestion_valeur(L[i])

for e in L:
    gestion_valeur(e)

for i,e in enumerate(L):
    gestion_clef(i)
    gestion_valeur(e)
```

Pour les dictionnaires, il y a un parcours standard

```
for k in D:
    gestion_clef(k)
    gestion_valeur(D[k])
```

et on dispose de 3 méthodes particulières

```
D.keys() # renvoie un iterable sur les clefs de D
D.values() # renvoie un iterable sur les valeurs de D
D.items() # renvoie un iterable sur les paires (clef,valeur)
```

Ce qui offre trois types de parcours supplémentaires pour les dictionnaires

```
for k in D.keys():
    gestion_clef(k)
    gestion_valeur(D[k])

for v in D.values():
    gestion_valeur(v)

for k,v in D.items():
    gestion_clef(k)
    gestion_valeur(v)
```

Le premier des 3 parcours, utilisant `D.keys()`, est en fait le parcours standard. L'ordre des 3 parcours utilisant les méthodes des dictionnaires et le même que celui des parcours de listes – si vous avez pris des notes en TD vous retrouverez les similarités que j'avais exposé à l'oral.

Voici un petit exemple dans lequel on souhaite stocker dans un dictionnaire, pour chaque nombre, la valeur de son carré, de son cube et de son inverse :

```
>>> D = {}
>>> for i in range(1, 6):
...     D[i] = {'carre': i*i, 'cube': i**3, 'inverse': round(1/i, 3)}
...
>>> for k in D:
...     print("clef", k, "valeur", D[k])
...
clef 1 valeur {'carre': 1, 'cube': 1, 'inverse': 1.0}
clef 2 valeur {'carre': 4, 'cube': 8, 'inverse': 0.5}
clef 3 valeur {'carre': 9, 'cube': 27, 'inverse': 0.333}
clef 4 valeur {'carre': 16, 'cube': 64, 'inverse': 0.25}
clef 5 valeur {'carre': 25, 'cube': 125, 'inverse': 0.2}
>>>
```

1.5 Savoir si ...

Pour savoir si un élément ou une clef est dans une liste :

```
if e in L:
    print('yes')
else:
    print('no')
```

```
if i in range(len(L)):
    print('yes')
else:
    print('no')
```

Pour savoir si un élément ou une clef est dans un dictionnaire :

```
if e in D.values():
    print('yes')
else:
    print('no')
```

```
if k in D: # ou if k in D.keys():
    print('yes')
else:
    print('no')
```

Comme on le voit, l'appartenance « naturel » pour une liste est un test sur les valeurs, pour un dictionnaire c'est sur les clefs.

2 NegAlphaBeta_memory

Le but de la mémoire est d'éviter, en premier lieu les recalculs inutiles. Un calcul sera considéré comme inutile si l'information stockée a été obtenu à un profondeur plus grande ou égale, ou bien si le calcul est considéré comme exact.
La première chose que l'on fait quand on « rentre » dans une méthode est de récupérer la clef, et de consulter le contenu de la mémoire après les éventuelles obligations

```
def methode(self, ...):
    # truc obligatoire si methode est decision

    clef = self.game.has_code
    if clef in self.decision.memory:
        # clef en mémoire, récupération
        memoire = self.decision.memory[clef]
        # traitement avec éventuel arrêt
```

```
        if memoire['pf'] >= pf or memoire['exact'] == True:
            # arret
            return memoire['score']
        #si on est ici, c'est qu'il y a un truc en memoire MAIS
        #pas suffisant pour s'arreter, on peut utiliser le score
        #on peut utiliser best_action (heuristique)
    # si on est ici c'est qu'on va poursuivre la méthode normale
```

2.1 Comment récupérer la clef du fils ?

Les deux méthodes `decision` et `__cut` effectue une boucle sur les actions

```
for a in self.game.actions:
    self.game.move(a)
    v = - self.__cut(pf-1, ...)
    self.game.undo()
    # exploitation de v
```

Comme la commande `move` modifie l'attribut `state` on va pouvoir obtenir la clef du fils en ajoutant une ligne de code

```
for a in self.game.actions:
    self.game.move(a)
    clef_fils = self.game.hash_code
    v = - self.__cut(pf-1, ...)
    self.game.undo()
    # exploitation de v et de clef_fils
```

2.2 Quelles sont les zones impactées dans le code ?

Le marqueur `# ICI` indique les zones impactées par la gestion de la mémoire dans la version « classique ».

```
def decision(self, state):
    """ the main method """
    self.game.state = state
    if self.game.turn != self.who_am_i:
        print("not my turn to play")
        return None
    beta = self.WIN+1
    alpha = -beta
    _v, _a = alpha, None
    pf = self.get_value('pf')

    # ICI
    for a in self.game.actions:
        self.game.move(a)
        # ICI
        _ = - self.__cut(pf-1, alpha, beta)
        self.game.undo()
        if _ > _v:
            # ICI
            _v, _a = _, a
    # ICI
    return _a

decision.memory = {}

def __cut(self, pf:int, alpha:float, beta:float) -> float:
    """ we use, max thus cut_beta """
    # ICI
    if pf == 0 or self.game.over():
        _c = 1 if self.who_am_i == self.game.turn else -1
        # ICI
        return _c * self.estimation()

    for a in self.game.actions:
        if alpha >= beta:
            # ICI
            return beta
        self.game.move(a)
        # ICI
        _M = - self.__cut(pf-1, -beta, -alpha)
        self.game.undo()
        # ICI
        alpha = max(_M, alpha)

    # ICI
    return alpha
```