Chapitre 6

Le dilemme du prisonnier

6.1. Nouveaux thèmes abordés dans ce chapitre

- dictionnaire
- fonction lambda

6.2. Les dictionnaires

Les dictionnaires sont des objets pouvant en contenir d'autres, à l'instar des listes. Cependant, au lieu de disposer ces informations dans un ordre précis, ils associent chaque objet contenu à une clé (la plupart du temps, une chaîne de caractères).

6.2.1. Création d'un dictionnaire

Un dictionnaire est un type modifiable. Nous pouvons donc d'abord créer un dictionnaire vide, puis le remplir. Contrairement aux listes, il n'y a pas de fonction spéciale (telle que `append()`) à appeler pour ajouter un objet.

```
dico = {}
dico['red'] = 'rouge'
dico['yellow'] = 'jaune'
dico['blue'] = 'bleu'
dico['green'] = 'vert'
dico['pink'] = 'rose'
```

On pourra afficher ce dictionnaire à l'écran facilement :

```
print(dico)
```

```
{'blue': 'bleu', 'pink': 'rose', 'green': 'vert', 'yellow': 'jaune', 'red': 'rouge'}
```

Le dictionnaire apparaît comme une suite d'éléments séparés par des virgules, le tout entre accolades. Chaque élément est composé d'une paire d'objets (ici deux chaînes de caractères, mais on pourrait avoir d'autres types d'objets) séparés par un double point. Le premier objet est la clé et le second est une valeur.

On remarquera que les paires n'apparaissent pas dans l'ordre dans lequel elles ont été introduites. Rappelez-vous qu'un dictionnaire n'est pas ordonné.

Pour accéder à une valeur, il suffit de connaître sa clé :

```
print(dico['pink'])
```

affichera `rose`.

Les dictionnaires sont des objets. On peut donc leur appliquer des méthodes spécifiques. La méthode `keys()` renvoie les clés utilisées dans le dictionnaire :

```
print(dico.keys())
```

```
dict_keys(['blue', 'pink', 'green', 'yellow', 'red'])
```

Pour connaître les valeurs, la méthode à utiliser est `values()` :

```
print(dico.values())
```

```
dict_values(['bleu', 'rose', 'vert', 'jaune', 'rouge'])
```

On peut aussi extraire du dictionnaire une séquence équivalente de tuples avec la méthode `items()`. Cela nous sera utile quand nous voudrons parcourir un dictionnaire avec une boucle.

```
print(dico.items())
```

```
dict_items([('blue', 'bleu'), ('pink', 'rose'), ('green', 'vert'), ('yellow', 'jaune'), ('red', 'rouge')])
```

6.2.2. Manipuler un dictionnaire

La fonction intégrée `len()` permet de connaître le nombre d'entrées du dictionnaire :

```
len(dico)
```

donnera comme résultat 5.

On a vu au § 6.2.1 comment ajouter des entrées. On peut aussi remplacer la valeur correspondant à une clé. Par exemple :

```
dico['blue'] = 'bleu ciel'
```

Pour résumer, si la clé existe, la valeur correspondante est remplacée ; si elle n'existe pas, une nouvelle entrée est créée.

On peut supprimer une entrée avec le mot-clé `del` :

```
del(dico['blue'])
print(dico)
```

```
{'pink': 'rose', 'green': 'vert', 'yellow': 'jaune', 'red': 'rouge'}
```

La méthode `pop()` supprime également la clé précisée, mais elle renvoie en plus la valeur supprimée. Cela peut être utile parfois.

```
couleur = dico.pop('pink')
print(couleur)
```

écrira `rose`.

D'une façon analogue à ce qui se passe avec les listes et les tuples, l'instruction `in` permet de savoir si un dictionnaire contient une clé déterminée :

```
'red' in dico
```

donnera comme résultat `True`.

Si l'on donne une clé qui n'est pas dans le dictionnaire, cela provoquera une erreur :

```
print(dico['cyan'])
```

provoquera l'erreur : KeyError: 'cyan'

Pour pallier ce problème, il existe la méthode `get()` :

```
print(dico.get('red', 'inconnu'))
```

écrira `rouge`, car `red` est dans le dictionnaire.

```
print(dico.get('cyan', 'inconnu'))
```

écrira `inconnu`, car `cyan` n'est pas dans le dictionnaire.
Le premier argument transmis à cette méthode est la clé de recherche, le second est la valeur que nous voulons obtenir en retour si la clé n'existe pas dans le dictionnaire.

6.2.3. Parcours d'un dictionnaire

Vous pouvez utiliser une boucle `for` pour traiter successivement tous les éléments contenus dans un dictionnaire, mais attention :

- au cours de l'itération, ce sont les clés du dictionnaire qui seront successivement affectées à la variable de travail, et non les valeurs ;
- l'ordre dans lequel les éléments seront parcourus est imprévisible (puisqu'un dictionnaire n'est pas ordonné).

```
for i in dico:
    print(i, dico[i])
```

écrira :

```
blue bleu
pink rose
green vert
yellow jaune
red rouge
```

La manière de procéder suivante est plus élégante, pour un résultat identique :

```
for cle, valeur in dico.items():
    print(cle, valeur)
```

La méthode `items()` renvoie une suite de tuples (clé, valeur).
On peut aussi ne parcourir que les clés :

```
for cle in dico.keys():
    print(cle)
```

ou que les valeurs :

```
for valeur in dico.values():
    print(valeur)
```

Trier un dictionnaire

Il n'est pas possible de trier un dictionnaire, puisque, encore une fois, c'est une structure non ordonnée. Par contre, il est possible d'écrire un dictionnaire selon un certain ordre, mais il faut d'abord créer une liste de tuples (une liste, elle, peut être triée sans problèmes).
Voici la ligne de code pour créer une liste de tuples triée selon les clés :

On verra au § 6.3 ce que sont ces `lambda`...

```
dico_trie = sorted(dico.items(), key=lambda x : x[0])
    print(dico_trie)
```

Le résultat sera :

```
[('blue', 'bleu'), ('green', 'vert'), ('pink', 'rose'), ('red', 'rouge'), ('yellow', 'jaune')]
```

Et voici la ligne de code pour créer une liste de tuples triée selon les valeurs :

```
dico_trie = sorted(dico.items(), key=lambda x : x[1])
    print(dico_trie)
```

Le résultat sera :

```
[('blue', 'bleu'), ('yellow', 'jaune'), ('pink', 'rose'), ('red', 'rouge'), ('green', 'vert')]
```

6.2.4. Copie d'un dictionnaire

La méthode `copy()` permet d'effectuer une vraie copie d'un dictionnaire. En effet, comme pour les listes (voir § 5.4.4), l'instruction `mon_dico = dico` ne crée pas un nouveau dictionnaire, mais une nouvelle référence vers le dictionnaire `dico`.

```
mon_dico = dico.copy()
```

Exercice 6.1

Construisez le dictionnaire des douze mois de l'année avec comme valeurs le nombre de jours respectif.
Utilisez la méthode `pprint()` du module `pprint` pour afficher ce dictionnaire à l'écran.

Exercice 6.2

Écrivez une fonction qui échange les clés et les valeurs d'un dictionnaire (ce qui permettra par exemple de transformer un dictionnaire anglais/français en un dictionnaire français/anglais). On suppose que le dictionnaire ne contient pas plusieurs valeurs identiques.

Exercice 6.3

En cryptographie, un chiffre de substitution simple consiste à remplacer chaque lettre d'un texte par une autre lettre (toujours la même). On utilise pour cela une table chiffrante. Par exemple :

Clair	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Chiffré	W	B	H	A	Y	P	O	D	Q	Z	X	N	T	S	F	L	R	U	V	M	C	E	K	J	G	I

Ainsi, le mot «RENARD» sera chiffré « UYSWUA ».

1. Écrivez un programme qui permet de (dé)chiffrer un texte avec la table ci-dessus.
2. Utilisez votre programme pour déchiffrer le message suivant :

HYHQYVMCSHDQPPUYAYVCBVMQMCMQFS

P. S. On peut parcourir les lettres d'une chaîne de caractères avec l'instruction :

```
for lettre in chaine:
```

Exercice 6.4

Vous avez à votre disposition sur le site web compagnon un texte non accentué et non ponctué. Écrivez un programme qui compte les occurrences de chacune des lettres de l'alphabet dans ce texte, et qui en fait un histogramme comme l'exemple fictif ci-dessous :

```
    : ||||||||||||||||||||
a   : ||||||||||||
d   : ||||
e   : ||||||||||||||||||||||||||
i   : |||||||||
l   : |||||||||
n   : ||||||||||
o   : |||||||
p   : ||
r   : ||||||||
s   : ||||||||
t   : ||||||||||||
u   : ||||||
```

6.3. Les fonctions lambda

Python permet une syntaxe intéressante pour définir des mini-fonctions d'une ligne. Empruntées au langage Lisp, ces fonctions dites *lambda* peuvent être employées partout où une fonction est nécessaire. Attention ! Les fonctions lambda sont limitées à une seule instruction.

La syntaxe d'une fonction lambda est la suivante :

```
ma_fonction = lambda arg1,arg2,... :instruction
```

où `arg1,arg2,...` est la liste des arguments de `ma_fonction`. Par exemple :

```
carre = lambda x: x*x
print(carre(9))
```

donnera comme résultat 81.

```
mult = lambda x,y=1: x*y
print(mult(6))
```

donnera comme résultat 6.

```
print(mult(6,3))
```

donnera comme résultat 18.

Il est évidemment possible que l'instruction soit une fonction définie ailleurs. Par exemple :

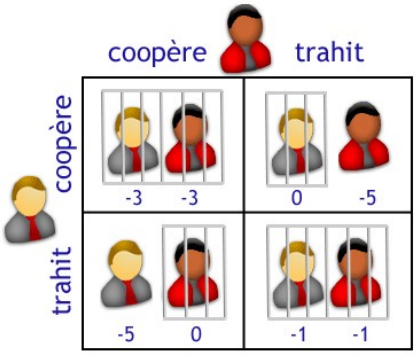
```
def signe(x):
    if x>0:
        return 1
    elif x<0:
        return -1
    else:
        return 0

f = lambda x : 2*signe(x)
print(f(-5))
```

donnera comme résultat –2.

6.4. Le dilemme du prisonnier - règles du jeu

Dans le jeu du « dilemme du prisonnier », deux détenus sont emprisonnés dans des cellules séparées. La police fait à chacun des deux le même marché :

« Tu as le choix entre dénoncer ton complice ou non. Si tu le dénonces et qu'il te dénonce aussi, vous aurez une remise de peine d'un an tous les deux. Si tu le dénonces et que ton complice te couvre, tu auras une remise de peine de 5 ans (et tu seras libéré), mais ton complice tirera le maximum. Mais si vous vous couvrez mutuellement, vous aurez tous les deux une remise de peine de 3 ans. »



Dans cette situation, il est clair que si les détenus s'entendent, ils s'en tireront globalement mieux que si l'un trahit l'autre. Mais l'un peut être tenté de s'en tirer encore mieux en trahissant son complice. Craignant cela, l'autre risque aussi de trahir son complice pour ne pas être le dindon de la farce. Le dilemme est donc : « faut-il coopérer avec son complice (et donc le couvrir) ou non ? »

Le dilemme du prisonnier devient plus intéressant et plus réaliste lorsque la durée de l'interaction n'est pas connue. On peut alors envisager de se souvenir du comportement d'un joueur à son égard et développer une stratégie en rapport. Par exemple, si je sais que mon adversaire ne coopère jamais, mon intérêt de ne pas coopérer non plus, sous peine d'être systématiquement grugé. Par contre, si je sais que mon adversaire coopérera toujours quoi qu'il arrive, j'aurai intérêt à être vicieux et ne jamais coopérer pour maximiser mon gain.

Exemple de partie

```
Coups et score du joueur Donnant donnant
['C', 'T', 'T', 'T', 'T', 'C', 'T', 'C', 'T', 'T']
19

Coups et score du joueur Aléatoire
['T', 'T', 'T', 'T', 'C', 'T', 'C', 'T', 'T', 'C']
19
```

6.5. Code du premier programme (les duels)

Voyons tout d'abord un premier programme où deux joueurs s'affrontent en duel.

dilemme.py

```python
# dilemme du prisonnier itéré, version duel

from random import choice

choix = ['T','C']  # T : trahit, C : coopère

def gain(lui,moi):
    if lui=='C' and moi=='C':
        return 3
    elif lui=='C' and moi=='T':
        return 5
    elif lui=='T' and moi=='C':
        return 0
    elif lui=='T' and moi=='T':
        return 1
```

```
# Toujours seul
# ne coopère jamais
def toujours_seul(liste_lui,liste_moi):
    return 'T'

# Bonne poire
# coopère toujours
def bonne_poire(liste_lui,liste_moi):
    return 'C'

# Aléatoire
# joue avec une probabilité égale 'T' ou 'C'
def aleatoire(liste_lui,liste_moi):
    global choix
    return choice(choix)

# Donnant donnant
# coopère seulement si l'autre joueur a coopéré au coup précédent.
def donnant_donnant(liste_lui,liste_moi):
    if len(liste_lui)>0:
        return liste_lui[-1]
    else:  # premier coup
        return 'C'

# Majorité
# coopère seulement si l'autre joueur a coopéré en majorité.
def majorite(liste_lui,liste_moi):
    if len(liste_lui)>0:
        if liste_lui.count('C') > len(liste_lui)//2:
            return 'C'
        else:
            return 'T'
    else:  # premier coup
        return 'C'


# Une partie entre deux joueurs différents

liste = {}
score = {}

liste['Aléatoire'] = []
liste['Donnant donnant'] = []

for joueur in liste.keys():
    score[joueur] = 0

nb_coups = 0
nb_total_coups = 10   # à modifier

while nb_coups < nb_total_coups :
    coup_joueur1 = aleatoire(liste['Donnant donnant'],liste['Aléatoire'])
    coup_joueur2 = donnant_donnant(liste['Aléatoire'],liste['Donnant donnant'])
    liste['Aléatoire'].append(coup_joueur1)
    liste['Donnant donnant'].append(coup_joueur2)
    score['Aléatoire'] += gain(coup_joueur2,coup_joueur1)
    score['Donnant donnant'] += gain(coup_joueur1,coup_joueur2)
    nb_coups += 1

for joueur in liste.keys():
    print("Coups et score du joueur",joueur)
    print(liste[joueur])
    print(score[joueur])
    print()
```

Analyse du programme

La première partie du programme ne présente rien de nouveau : on définit les gains et les stratégies des joueurs (les noms des stratégies sont données par les noms des joueurs).

La grande nouveauté apparaît avec les dictionnaires :

```
liste = {}
score = {}
```

On crée d'abord deux dictionnaires : `liste` donnant la liste des coups joués par les joueurs, et `score` donnant leur score respectif.
On initialise ensuite les deux listes de coups :

```
liste['Aléatoire'] = []
liste['Donnant donnant'] = []
```

puis on met les scores à 0 :

```
for joueur in liste.keys():
    score[joueur] = 0
```

Enfin, le duel commence et durera `nb_total_coups` :

```
nb_coups = 0
nb_total_coups = 10   # à modifier

while nb_coups < nb_total_coups :
```

Chaque joueur joue selon sa stratégie en tenant compte des coups précédents (le premier paramètre est la liste des coups de l'adversaire, le second la liste des coups du joueur) :

```
    coup_joueur1 = aleatoire(liste['Donnant donnant'],liste['Aléatoire'])
    coup_joueur2 = donnant_donnant(liste['Aléatoire'],liste['Donnant donnant'])
```

On ajoute ce coup à la liste des coups :

```
    liste['Aléatoire'].append(coup_joueur1)
    liste['Donnant donnant'].append(coup_joueur2)
```

puis on met à jour les scores et on recommence pour le coup suivant :

```
    score['Aléatoire'] += gain(coup_joueur2,coup_joueur1)
    score['Donnant donnant'] += gain(coup_joueur1,coup_joueur2)
    nb_coups += 1
```

On affiche finalement les coups joués lors du duel et les scores des deux joueurs :

```
for joueur in liste.keys():
    print("Coups et score du joueur",joueur)
    print(liste[joueur])
    print(score[joueur])
    print()
```

6.6. Code partiel du second programme (le tournoi)

Dans ce second programme, on pourra faire s'affronter plus de deux joueurs. Chaque joueur affrontera en duel chacun des autres joueur, y compris lui-même !

tournoi.py

```
# dilemme du prisonnier itéré, version tournoi
```

Le début du code est strictement identique à la version duel. Voir § 6.5.

```
# Le tournoi

liste = {}
strategie = {}
score = {}
```

```
duel = {}

# ajouter des joueurs ci-dessous, selon les modèles des joueurs existants
# commencer ici
liste['Toujours seul'] = []
liste['Bonne poire'] = []
liste['Majorité'] = []
liste['Aléatoire'] = []
liste['Donnant donnant'] = []
# ...

strategie['Toujours seul'] = lambda lui, moi : toujours_seul(lui,moi)
strategie['Bonne poire'] = lambda lui, moi : bonne_poire(lui,moi)
strategie['Majorité'] = lambda lui, moi : majorite(lui,moi)
strategie['Aléatoire'] = lambda lui, moi : aleatoire(lui,moi)
strategie['Donnant donnant'] = lambda lui, moi : donnant_donnant(lui,moi)
# ...
# terminer là

nb_total_coups = 10  # à modifier

for joueur in liste.keys():
    score[joueur] = 0

for i in liste.keys():  # i et j sont les joueurs
    for j in liste.keys() :
        liste[i] = []   # on recommence une partie
        liste[j] = []
        if i>=j:
            nb_coups = 0
            score_joueur1 = 0
            score_joueur2 = 0
            while nb_coups < nb_total_coups :
                coup_joueur1 = strategie[i](liste[j],liste[i])
                coup_joueur2 = strategie[j](liste[i],liste[j])
                liste[i].append(coup_joueur1)
                if i!=j:
                    liste[j].append(coup_joueur2)
                score_joueur1 += gain(coup_joueur2,coup_joueur1)
                score_joueur2 += gain(coup_joueur1,coup_joueur2)
                nb_coups += 1
            duel[(i,j)] = score_joueur1
            if i!=j:
                duel[(j,i)] = score_joueur2
            score[i] += score_joueur1
            if i!=j:
                score[j] += score_joueur2

# affichage des résultats

def trie_par_valeur(d):
    #retourne une liste de tuples triée selon les valeurs
    return sorted(d.items(), key=lambda x: x[1])

def trie_par_cle(d):
    #retourne une liste de tuples triée selon les clés
    return sorted(d.items(), key=lambda x: x[0])

score_trie = trie_par_valeur(score)
score_trie.reverse()
for i in range(0,len(score_trie)):
    print(score_trie[i][0],":",score_trie[i][1])
print()
duel_trie = trie_par_cle(duel)
for i in range(0,len(duel_trie)):
    print(duel_trie[i][0][0],"contre",duel_trie[i][0][1],"gagne",duel_trie[i][1],"pts")
```

Analyse du programme

On aura besoin ici de quatre dictionnaires :

```
liste = {}
```

```
strategie = {}
score = {}
duel = {}
```

Le dictionnaire `liste` mémorise la liste des coups durant un duel :

```
liste['Toujours seul'] = []
liste['Bonne poire'] = []
liste['Majorité'] = []
liste['Aléatoire'] = []
liste['Donnant donnant'] = []
```

Les stratégies sont « rangées » dans un dictionnaire. Remarquez l'utilisation des fonctions lambda. On est obligé de les utiliser parce que les fonctions-stratégies ont des paramètres.

```
strategie['Toujours seul'] = lambda lui, moi : toujours_seul(lui,moi)
strategie['Bonne poire'] = lambda lui, moi : bonne_poire(lui,moi)
strategie['Majorité'] = lambda lui, moi : majorite(lui,moi)
strategie['Aléatoire'] = lambda lui, moi : aleatoire(lui,moi)
strategie['Donnant donnant'] = lambda lui, moi : donnant_donnant(lui,moi)
```

Si la fonction n'a pas de paramètre, on peut se passer du `lambda`.
Par exemple :

```
def moi():
    print("Je m'appelle Jean")


fonct = {}
fonct['moi'] = moi()
fonct['moi']
```

écrira « Je m'appelle Jean ».

On met à zéro les scores des joueurs :

```
for joueur in liste.keys():
    score[joueur] = 0
```

Chaque joueur (`i`) affrontera une fois en duel chacun des autres joueurs (`j`), y compris lui-même. La condition `if i>=j:` est là pour n'autoriser qu'un duel et non pas deux.

Il faut aussi faire attention, quand `i` égale `j`, de ne pas compter les scores à double, de ne pas mettre tous les coups dans une seule liste et encore de mémoriser le bon score. Cela explique les trois `if i!=j:` du code ci-dessous.

```
for i in liste.keys():  # i et j sont les joueurs
    for j in liste.keys() :
        liste[i] = []   # on recommence une partie
        liste[j] = []
        if i>=j:
            nb_coups = 0
            score_joueur1 = 0
            score_joueur2 = 0
            while nb_coups < nb_total_coups :
                coup_joueur1 = strategie[i](liste[j],liste[i])
                coup_joueur2 = strategie[j](liste[i],liste[j])
                liste[i].append(coup_joueur1)
                if i!=j:
                    liste[j].append(coup_joueur2)
                score_joueur1 += gain(coup_joueur2,coup_joueur1)
                score_joueur2 += gain(coup_joueur1,coup_joueur2)
                nb_coups += 1
            duel[(i,j)] = score_joueur1
            if i!=j:
                duel[(j,i)] = score_joueur2
            score[i] += score_joueur1
            if i!=j:
```

```
                score[j] += score_joueur2
```

Le dictionnaire `duel` va nous servir à détailler les scores à la fin de la simulation. On pourra ainsi mieux analyser les résultats du tournoi.

La fin du code permet d'afficher les résultats, triés par ordre décroissant pour les scores, et par ordre alphabétique pour les duels.

```
def trie_par_valeur(d):
    #retourne une liste de tuples triée selon les valeurs
    return sorted(d.items(), key=lambda x: x[1])

def trie_par_cle(d):
    #retourne une liste de tuples triée selon les clés
    return sorted(d.items(), key=lambda x: x[0])

score_trie = trie_par_valeur(score)
score_trie.reverse()
for i in range(0,len(score_trie)):
    print(score_trie[i][0],":",score_trie[i][1])
print()
duel_trie = trie_par_cle(duel)
for i in range(0,len(duel_trie)):
    print(duel_trie[i][0][0],"contre",duel_trie[i][0][1],"gagne",duel_trie[i][1],"pts")
```

Exercice 6.5 - Tournoi

Le but de cet exercice est de définir une stratégie, c'est-à-dire de définir une règle pour savoir quand accepter et quand refuser la coopération avec un autre joueur. Les seules informations connues sont la liste des coups que l'adversaire a joués jusque-là, et la vôtre.

1. Programmez quelques stratégies dans le programme du § 6.5. N'oubliez pas le commentaire qui décrira succinctement votre stratégie.
2. Testez ces stratégies en modifiant le programme du § 6.6 et choisissez la meilleure.
3. Donnez votre stratégie préférée au professeur en vue du tournoi qui mettra en lice tous les élèves de la classe. Le nom de cette stratégie sera votre prénom.

Références

- Delahaye J.-P., *L'altruisme récompensé ?*, Pour la Science 181, novembre 1992, pp. 150-6
- Delahaye J.-P., Mathieu Ph., *L'altruisme perfectionné*, Pour la Science 187, mai 1993, pp. 102-7
- Delahaye J.-P., *Le dilemme du prisonnier et l'illusion de l'extorsion*, Pour la Science 435, janvier 2014, pp. 78-83
- Delahaye J.-P., Mathieu Ph., *Adoucir son comportement ou le durcir*, Pour la Science 462, avril 2016, pp. 80-5

Exercice 6.6 - Le dilemme du renvoi d'ascenseur

Situation ambiguë : je suis face à un inconnu au rez-de-chaussée de cette grande tour, devant ce petit ascenseur à une seule place dont la porte est ouverte. Il n'y a pas de bouton pour le rappel lorsqu'il sera monté, et je n'ai pas du tout, mais alors pas du tout, envie de gravir les huit étages à pied. L'inconnu me dit : « Laissez-moi passer en premier, car, arrivé en haut, je vous renverrai l'ascenseur. »

Dois-je accepter ? Pourquoi devrais-je le croire et accepter qu'il passe devant moi ? D'ailleurs tiendra-t-il sa promesse si je le laisse passer ? C'est le dilemme du renvoi d'ascenseur.

Lorsque nous imaginons que nous devons y jouer à de multiples reprises avec toutes sortes de gens, cette situation fait ressortir de multiples aspects inattendus. Cette variante du dilemme itéré du prisonnier (voir ex. 6.5) possède des propriétés remarquables.

Refaites le tournoi de l'exercice 6.5 avec les gains ci-dessous :

```
def gain(lui,moi):
    if lui=='C' and moi=='C':
        return 3
    elif lui=='C' and moi=='T':
        return 8                    # au lieu de 5
    elif lui=='T' and moi=='C':
        return 0
    elif lui=='T' and moi=='T':
        return 1
```

Que constatez-vous ?
Pourrez-vous trouver une meilleure stratégie pour ce dilemme ?

Référence : Delahaye J.-P., Mathieu Ph., *Le dilemme du renvoi d'ascenseur*, Pour la Science 269, mars 2000, pp. 102-6

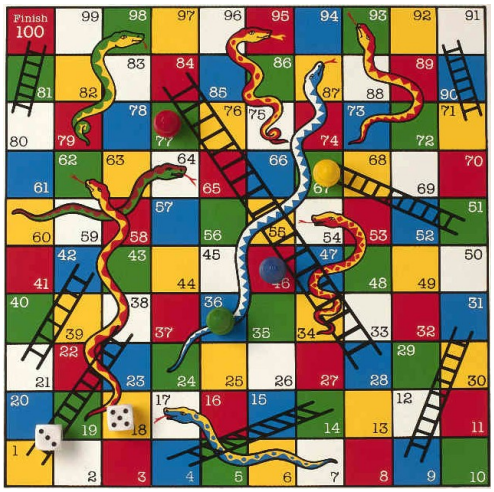
Exercice 6.7

Représentez la position du jeu d'échecs ci-contre en utilisant un dictionnaire.
La clé sera la position d'une case (un tuple, par exemple ('A', 7), la valeur sera la pièce qui s'y trouve (par exemple « dame noire »).
Ne mettez dans le dictionnaire que les cases occupées.

Exercice 6.8 - Le jeu des échelles

Règles du jeu

Le joueur lance un dé et avance d'autant de cases que de points sur le dé.
S'il tombe sur une case dans laquelle il y a le pied d'une échelle, il monte le long de celle-ci jusqu'en haut. S'il tombe sur une case dans laquelle il y a la tête d'un serpent, il doit redescendre jusqu'à la queue du serpent.
La partie se termine quand on arrive sur la case 100. Si le joueur tire un dé qui ne lui permet pas d'arriver exactement sur la case 100, il recule du nombre de points supplémentaires sur le dé.



Évidemment, une case ne peut être le départ ou l'arrivée que d'une seule échelle et un seul serpent.

Première question

La question qui nous intéresse est la suivante : « Comment de fois faudra-t-il (en moyenne) lancer le dé pour terminer la partie ? » Simulez 100'000 parties sur le plateau ci-dessus.

Deuxième question (plus difficile)

Déplacez (ou pas) horizontalement les serpents et les échelles du plateau ci-dessus de manière à :

a) maximiser le nombre de lancers de dés
b) minimiser le nombre de lancers de dés.

Exercice 6.9

Première partie

Écrivez un programme qui code un message en Morse :

```
entrée : code morse
```

donnera comme résultat :

```
['-.-.', '---', '-..', '.', ' ', '--', '---', '.-.', '...', '.']
```

Vous trouverez l'alphabet Morse à l'adresse : www.apprendre-en-ligne.net/crypto/morse/

Seconde partie

Écouteurs obligatoires !

Utilisez ensuite la méthode `Beep` du module `winsound` pour transformer les traits et les points en signaux sonores.

Voici ce que dit la documentation Python sur la méthode `Beep` :

winsound.**Beep**(frequency, duration)

Beep the PC's speaker. The *frequency* parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The *duration* parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, **RuntimeError** is raised.

6.7. Ce que vous avez appris dans ce chapitre

- Vous avez vu comment créer un dictionnaire et le manipuler.
- Faites bien attention qu'un dictionnaire n'est pas ordonné. Cela peut parfois occasionner de mauvaises surprises...
- Vous avez aussi vu les fonctions `lambda`. Elles peuvent paraître superflues, mais sont utiles dans certains contextes, comme celui du jeu du dilemme du prisonnier, version Tournoi (§ 6.6). Généralement, on utilise plutôt le mot-clé `def`.
- Enfin, vous avez vu (et surtout entendu) une manière primitive de générer du son avec Python (exercice 6.9).