Antonin Fontaine
TS2

Année 2015/2016

Informatique et Science du Numérique

« Programmation d'un jeu de cartes en Python »

Visual Studio

GitHub

Table des matières

I) Présentation du projet.

Nous avions pour but de programmer un jeu de carte simple qui permettrait d'explorer toutes les "facettes" de la programmation en python, c'est pourquoi nous avons choisi la « bataille ».

Les règles sont simples : on joue avec un jeu de 52 cartes divisé en 2 : chaque joueur à 26 cartes. À chaque tour, chaque joueur tire la première carte de son paquet et la compare avec son adversaire. Le joueur ayant tiré la plus grosse carte remporte les deux cartes et les remets à le fin de son paquet, et on recommence.
En cas d'égalité, il y a "bataille" : Chaque joueur tire deux autres cartes qu'il pose face cachée sur les premières, et retire deux dernières cartes qu'il compare à nouveau, celui qui remporte gagne toutes les cartes présentes sur la table. (en cas de nouvelle égalité, on recommence le processus de bataille).
La partie se termine quand l'un des deux joueurs n'a plus de carte : il a perdu.
Ordre des cartes : 2 < 3 < 4 < 5 < 6 <7 <8 < 9 < 10 <Valet < Dame < Roi < As

II) Variables et programme.

1) creation()

C'est la fonction appelée en début de programme pour créer un paquet de 52 cartes. Ce paquet est en fait une liste de 52 tuples représentant les 52 cartes. Ces tuples contiennent 2 éléments : la valeur et la couleur de chaque carte. La valeur est matérialisée par des numéros allant de 2 à 14 pour être comparée. En effet, le passage par des numéros est indispensable pour matérialiser la valeur de l'As et des « habillés » (Valet = 11, Dame = 12, Roi = 13, As = 14).

Cette fonction crée une liste vide 'jeu' et 2 listes : 'couleurs' et 'valeurs'. Elle va ensuite former et ajouter les 52 cartes une par une dans la liste 'jeu'. La construction des cartes se fait à l'aide de deux boucles 'for' l'une dans l'autre : La première tourne pour 'c' parcourant la liste des couleurs, et pour chaque valeur de 'c', la deuxième tourne pour 'v' parcourant la liste des valeurs. À chaque carte créée(à chaque tuple (v, c) créé), elle est ajoutée à la liste 'jeu'. À la sortie de cette boucle, nous avons une liste contenant 52 éléments : 52 tuples représentant les 52 cartes : 13 par couleur.

2) melanger(liste)

Cette fonction permet de mélanger le jeu créé précédemment. Elle prend en entrée une liste et retourne en sortie une autre liste qui est en fait la même que l'entrée, à une différence près : l'ordre des éléments est changé et aléatoire.

La fonction melanger(liste) commence par créer une liste vide, future liste mélangée.

Elle prend une carte au hasard dans la liste d'entrée à l'aide de la fonction du module random,

choice(). Cette carte est ajoutée à la liste 'ListeMelangee' et est retirée de la liste de départ. Tout ça dans une boucle 'while' de laquelle on sort lorsque la liste de départ est vide (autrement dit, quand toutes les cartes ont été ajoutées dans la 2ème liste dans un ordre aléatoire et retirées de la première.).

3) distribuer()

La bataille se joue avec 2 joueurs. Il faut donc séparer le paquet mélangé en 2 : 'jeu1' et 'jeu2'.

On fait appel pour cela à 2 fonctions : 'distribuerj1' et 'distribuerj2'. La première prend les 26 premières cartes du paquet pour les placer dans le jeu1. La seconde place les 26 dernières cartes dans le jeu2.

Nous avons donc maintenant 2 listes, jeu1 et jeu2, qui contiennent chacune 26 cartes mélangées représentant les jeux des deux joueurs.

4) Tirer()

Cette fonction très courte prend en entrée une liste et retourne le premier élément de cette liste(ici, la première carte.), en le supprimant de la liste initiale. Elle nous sert surtout à gagner de la place à chaque appel de cette fonction.

5) Partie()

Cette fonction représente les « règles du jeu », c'est à l'intérieur de celui ci que le programme principal va tourner lors du déroulement d'une partie.
On commence par initialiser 3 variables : 'reste' de type liste ; 'nombre_de_tours' de type entier; 'nombre_de_batailles' de type entier. 'reste' nous servira lors d'une égalité de carte (bataille).

```
120 def partie():
121     reste = []
122     nombre_de_tours = 0
123     nombre_de_batailles = 0
```

Une partie de bataille s'arrête lorsqu'un des deux joueurs n'a plus de cartes dans son jeu, c'est pourquoi toute la suite du programme est comprise dans une boucle 'while' qui tourne tant que les deux joueurs ont encore des cartes. (si l'un d'eux n'a plus de carte, la boucle s'arrête)

```
carte1 = tirer(jeu1)
carte2 = tirer(jeu2)
```

On créé deux variables : 'carte1' et 'carte2' qui sont des tuples et que l'on obtient en appelant la fonction tirer() avec comme paramètre le jeu (la liste de cartes) dans lequel on doit tirer la carte.

On compare ensuite les 2 cartes, ou plutôt, la valeur de ces deux cartes, autrement dit, le premier élément des tuples représentant les cartes. On fait appel à un « if » puis à un « elif » pour tester les deux cas : si la carte1 ou la carte2 remporte. On ajoute les deux cartes au jeu gagnant grâce à la méthode .append().

En cas d'égalité (« else »), on fait appel à la fonction bataille() avec en paramètres : les deux

cartes égales, la liste vide 'reste', et l'entier 'nombre_de_bataille'.

```
else:
    bataille(carte1, carte2, reste, nombre_de_batailles)
```

bataille() :

On commence par incrémenter de 1 la variable 'nombre_de_batailles'.
On ajoute au reste les 2 cartes égales (pour les garder en mémoire).
On contrôle toujours si il reste bien des cartes pour les deux joueurs.

```
if len(jeu1) != 0 and len(jeu2) != 0:
```

On tire ensuite les deux cartes intermédiaires(toujours à l'aide de la fonction « tirer() ») qui ne sont pas comparées et ajoutées directement au reste.
carte1 et carte2 sont maintenant remplacées par deux nouvelles cartes tirées dans chacun des jeux des deux joueurs.
Leurs valeurs sont ensuite comparées, en suivant le même schéma que dans la fonction partie.

On ajoute au jeu gagnant les deux cartes comparées, ainsi que la liste « reste » à l'aide d'une boucle « for » qui ajoute chaque éléments de la liste « reste » l'un après l'autre.

```
elif carte1[0] > carte2[0]:
    for k in range(len(reste)):
        jeu1.append(reste[k])
    jeu1.append(carte1)
    jeu1.append(carte2)
```

Si les deux cartes sont de nouveau égales, on appelle de nouveau la fonction bataille (récursivité) avec comme paramètres : les deux nouvelles cartes égales, la liste « reste » qui n'est plus vide cette fois ci, et le « nombre_de_batailles ».

Lorsque le ou les appels à la fonction bataille() sont terminés, on vide la liste « reste », on incrémente l'entier « nombre_de_tours » et on recommence !(tant qu'il reste des cartes dans les deux jeux)

Si l'un des deux jeux est vide, on indique quel joueur a gagné, en combien de tours, et avec combien de batailles.

6) Programme principal.

Il crée d'abord un jeu à l'aide de la fonction création.
Puis il le mélange avec la fonction mélanger.
Ensuite il distribue.(il sépare le jeu mélangé en 2)
Il appelle enfin la fonction partie qui tourne jusqu'à la fin de la partie.

```
"""Programme principale"""
jeu = creation()

jeu = melanger(jeu)
jeu1 = distribuerj1(jeu)
jeu2 = distribuerj2(jeu)
partie()
```

III) Les solutions pour les sous-programmes.

J'ai travaillé sur les fonctions creation(), partie() et bataille(), puis un peu sur l'interface graphique en collaboration avec mes coéquipiers, qui y ont consacré plus de temps.

1) creation() :

Cette fonction permet de créer une liste de 52 cartes à l'aide de deux boucles « for » pour ne pas avoir à créer le jeu manuellement. Il y a une solution plus courte pour créer une liste à l'aide d'itérables qui tient sur une ligne mais qui est beaucoup moins lisible et intuitive pour une personne lisant le code. Ces boucles sont placées dans l'ordre pour que le paquet de carte soit trié par couleur et non par valeur.
Je crée aussi un dictionnaire nommé "traduction". Tout en comparant les valeurs de cartes comme le valet ou l'as (à l'aide d'entiers), on peut afficher les cartes avec le visuel pour les joueurs humains.

2) partie() :

Ce sous-programme tourne dans une boucle 'while' mais j'ai adopté cette solution après avoir fait des essais avec des tests « if » (comme dans « bataille() ») puis avec une boucle infinie et des « try/except » pour sortir quand un jeu était vide. Dans un souci de simplicité, j'ai adopté la boucle while car plus simple à manipuler et moins lourde à comprendre.

Des simples blocs d'instructions « if/elif/else » m'ont parus tout de suite fonctionnels pour comparer la valeur des cartes (on compare ici le premier élément de la liste : un entier)

```
if carte1[0] > carte2[0]:
```

3) bataille() :

Cette fonction ne tourne pas dans un while comme partie() mais on vérifie à chaque "groupe" d'action si les deux joueurs ont toujours des cartes dans leurs jeux. Ce choix est dû à l'utilisation de la récursivité : dans la fonction bataille(), si les deux nouvelles cartes sont une nouvelles fois égales, on appelle la fonction bataille(), à l'intérieur d'elle même, pour recommencer le processus en gardant des paramètres comme la liste « reste ».

IV) Problèmes rencontrés.

1) La « traduction »

L'ordinateur ne sait pas ce que représente un Valet, ou que l'As (le 1) soit en fait la plus forte carte … c'est pourquoi j'utilise les entiers 11, 12, 13 et 14 pour les habillés et l'As.
(Voir II)1) creation())
En revanche, pour l'utilisateur, il est toujours mieux d'afficher « As » plutôt que « 14 », question d'habitude … J'avais déjà entendu parler des dictionnaires et j'utilise ce type de variables pour effectuer la traduction entre la machine et l'utilisateur.

```
traduction = {11: "Valet", 12: "Dame", 13: "Roi", 14: "As"}
```

Ce dictionnaire est composé de clefs (les entiers de 11 à 14) ainsi que de leurs valeurs associées. (11 correspond au Valet, 12 à la Dame, etc …)

Avant d'afficher des cartes, dans partie() par exemple, le programme test si ces cartes ont une valeur qui correspond à une clef du dictionnaire : `if carte1[0] in traduction:`

- Si non, on passe outre et on affiche la carte tel qu'elle. (8 de Carreau par exemple)
- Si oui, on crée carte_traduite, copié-collé de la carte de départ, sauf qu'on remplace son premier élément par la valeur du dictionnaire correspondante à l'aide de la méthode .get. (par exemple, (11 ; Carreau) devient (Valet ; Carreau))

```
carte_traduite1 = (traduction.get(carte1[0]), carte1[1])
```

2) Batailles successives :

Je suis resté longtemps bloqué sur cet aspect, entêté par l'idée d'utiliser une boucle tant-que les cartes restent égales. Résultats : trop compliqué de garder en mémoire les cartes dans la liste reste, erreurs si l'un des 2 jeux est vide, etc …

La récursivité, c'est à dire le fait d'appeler une fonction dans sa propre définition, a changé beaucoup de choses … Pour moi, c'était réservé aux dessins de fractales avec « Turtle » et ça me semblait hors de porté … Mon professeur m'en a parlé, j'ai essayé et ça a tout de suite marché !

3) Parties interminables :

Le code fonctionne, aucune erreur, et pourtant, il y a un problème.
Une partie dure en moyenne plus de 4 000 tours et, parfois, ne se finit jamais (chaque joueur se met à gagner chacun son tour : aucun vainqueur ne se détache).
Un sommaire debug à base de « print() » disséminés un peu partout ne révèle aucun problème, tout comme un debug plus approfondi grâce à Visual studio …

« C'est peut-être normal » : hypothèse vite faussée par les statistiques trouvées sur le Web. En effet, une partie dure maximum 500 tours et n'est jamais infinie …

J'ai découvert la solution à ce problème un peu par hasard.(je crois qu'on appelle ça la « sérendipité ») Il s'agit d'un problème dans l'ordre d'ajout des cartes lorsque un joueur gagne :

```
if carte1[0] > carte2[0]:
    jeu1.append(carte1)
    jeu1.append(carte2)
elif carte2[0] > carte1[0]:
    jeu2.append(carte1)
    jeu2.append(carte2)
```

- Si le joueur 1 gagne, on ajoute à son jeu : d'abord sa carte, puis celle de son adversaire.
- Si le joueur 2 gagne, on ajoute les cartes dans le mêmes ordre que dans le cas précèdent : voilà le problème.

L'ordre d'ajout des cartes ne dépend donc pas du gagnant : en ajoutant d'abord la carte du gagnant, puis l'autre, on résout tous ces problèmes !

V) Evolution temporelle du projet.

Nous avions d'abord comme objectif de modéliser un jeu de 52 cartes. J'ai d'abord programmé la fonction création qui permet de créer ce jeu, ou plutôt une liste de cartes, chacune représentée par deux éléments : la valeur et la couleur, contenues dans un tuple (liste non-modifiable). Avant même de terminer l'écriture de cette fonction, je savais déjà qu'elle allait retourner en sortie une liste de 52 tuples, mes coéquipiers ont donc pu programmer en parallèle les fonctions melanger() et distribuer().

Je me suis ensuite mis à l'écriture de partie() et de bataille() qui m'a pris la majorité de mon temps consacré au projet.

D'abord la fonction partie() sans gestion des batailles qui n'a pas posé beaucoup de problèmes, et enfin bataille() qui m'a pris plus de temps.

VI) Améliorations possibles :

1) Autres jeux :

Pourquoi pas reprendre nos fonctions de bases comme création() ou distribuer() pour les adapter à d'autres jeu et d'autres règles ?(Solitaire, Belote, etc …)

2) Objet :

Re-programmer la bataille en « orienté objets » permettrait de découvrir ce domaine qui prédomine dans la programmation en général …
(C'est ce que j'ai fais en m'aidant de cours, forums, etc …)

Le code source est sur :

https://goo.gl/wZFWkQ