INFO-F-106 : PROJET D'INFORMATIQUE
SET

Jérôme De Boeck Jacopo De Stefani Gwenaël Joret Fabio Sciamannini
Nikita Veshchikov

version du 14 mars 2016

Présentation générale

Le projet en une phrase

L'objectif du projet est d'implémenter une variante pour un joueur "contre la montre" du jeu de cartes SET.

Les règles du jeu SET

Dans le jeu SET, une carte correspond à un vecteur (x_1, x_2, x_3, x_4) de taille 4 où chaque x_i appartient à l'ensemble $\{0, 1, 2\}$. Il y a donc $3^4 = 81$ cartes dans le jeu. Chacune des quatres dimensions d'une carte (x_1, x_2, x_3, x_4) est un *attribut* de la carte. Graphiquement, ces différents attributs sont representés en utilisant

- un type de symbole par carte ;
- un nombre de copies du symbole par carte (1, 2, 3) ;
- une couleur par carte (rouge, bleu, vert) ;
- un remplissage par carte (vide, hachuré, rempli).

Pour illustration, l'ensemble des 81 cartes représentées selon la version commercialisée du jeu est donné à la Figure 1.

Un *set* est un ensemble de trois cartes distinctes (x_1, x_2, x_3, x_4), (y_1, y_2, y_3, y_4), (z_1, z_2, z_3, z_4) qui satisfont la propriété suivante : Pour chaque attribut, soit les trois cartes assignent la même valeur à cet attribut, soit elles assignent chacune une valeur distincte. Par-exemple,

$$(0, 1, 0, 2)$$
$$(0, 0, 2, 2)$$
$$(0, 2, 1, 2)$$

est un set. Par-contre, l'ensemble suivant

$$(1, 0, 0, 1)$$
$$(0, 1, 2, 1)$$
$$(2, 2, 2, 1)$$

n'est pas un SET. (Pourquoi ?).

Règles du SET classique

La version classique de SET (pour deux joueurs et plus) fonctionne comme suit. La pile de cartes (*deck*) est tout d'abord mélangée. Ensuite, les 12 premières cartes sont déposées face visible sur la table. Ces cartes constituent le tableau de jeu (*board*). Il est habituel de les arranger en 3×4 ou 4×3, comme dans l'exemple suivant.

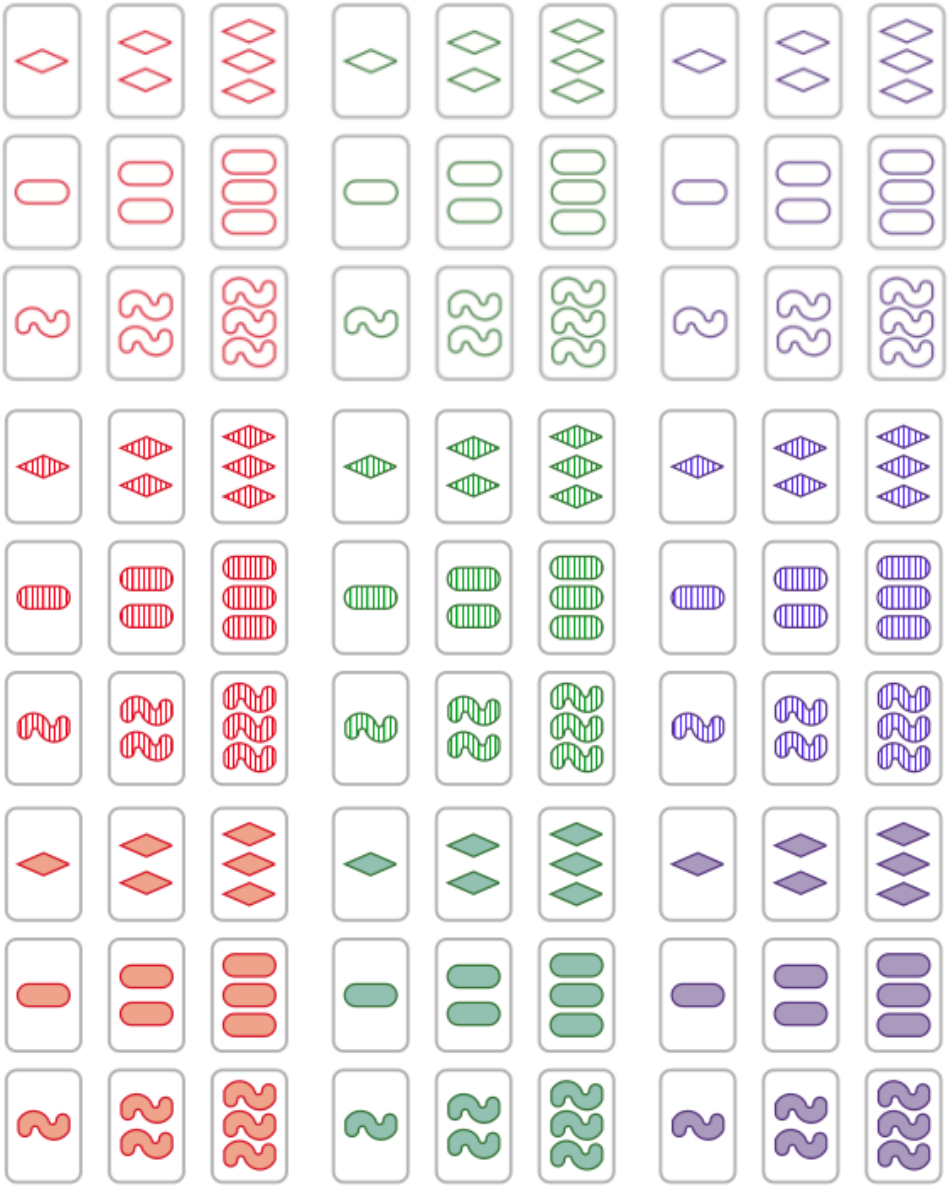

FIGURE 1 – Jeu de cartes au complet.

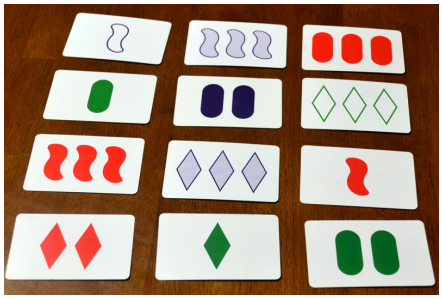

Tableau de jeu de SET

L'objectif pour chaque joueur est d'identifier le plus vite possible un set sur le tableau de jeu. Une fois un tel ensemble identifié, le joueur le montre aux autres joueurs pour vérification. Il prend ensuite les trois cartes et les remplace avec trois nouvelles cartes de la pioche. Une fois la partie terminée, le gagnant est le joueur ayant identifié le plus de sets.

Et s'il n'y a pas de set ?

Il est possible que les cartes présentes sur le tableau de jeu ne contiennent simplement pas de set. Si tous les joueurs sont convaincus qu'il n'y a pas de set devant eux, de commun accord ils ajoutent trois cartes supplémentaires de la pioche. Si la pioche est vide, la partie est alors terminée.

Variante un joueur

La variante qui nous intéressera dans le cadre de ce projet est proche des règles décrites ci-dessus. Les différences sont les suivantes. Tout d'abord, il n'y a qu'un seul joueur, jouant contre la montre. La dynamique principale du jeu, identifier des sets le plus vite possible, reste inchangée.

Ensuite, le cas où il n'existe pas de set sur le tableau de jeu est traité différemment. Si le joueur affirme qu'il n'existe pas de set, l'ordinateur (jouant le rôle d'arbitre) vérifie d'abord si c'est correct. Si c'est le cas, deux situations sont possibles :

1. La pioche n'est pas vide. On ajoute les 3 cartes suivantes de la pioche sur le tableau de jeu. (Le tableau de jeu peut donc grandir et contenir 15, voire 18 cartes ...)
2. La pioche est vide. L'ordinateur déclare alors la partie terminée.

Si par-contre le joueur affirme qu'il n'y a pas de set en jeu alors qu'il en existe un, une pénalité de trois cartes est infligée comme suit : trois cartes sont choisies au *hasard* dans la défausse (*graveyard*) et ajoutées à la fin de la pioche.

Historique et propriétés mathématiques

Le jeu SET a été imaginé par Marsha Jean Falco en 1974, inspirée par son travail en génétique qu'elle menait à l'Université de Michigan. Il a fallu attendre 1991 pour la commercialisation du jeu, par la compagnie *Set Enterprise*.

Voici quelques propriétés mathématiques du jeu SET qui seront utiles dans le cadre de ce projet. Tout d'abord, observons que trois entiers $x, y, z \in \{0, 1, 2\}$ ont la propriété d'être soit tous égaux soit tous distincts si et seulement si $(x + y + z) \bmod 3 = 0$. Par conséquent, trois cartes $(x_1, x_2, x_3, x_4), (y_1, y_2, y_3, y_4), (z_1, z_2, z_3, z_4)$ du jeu forment un set si et seulement si elles sont distinctes et leur somme modulo 3 est le vecteur nul $(0, 0, 0, 0)$. Cette propriété permet de tester très facilement si trois cartes forment un set.

Une seconde propriété du jeu SET, beaucoup moins évidente, est la suivante : Tout ensemble d'au moins 21 cartes contient un set. (Remarque : Il n'est pas nécessaire d'utiliser cette propriété pour réaliser ce projet, cependant vous pouvez l'exploiter dans votre code si vous le souhaitez.) Notons que 21 est le plus petit nombre ayant cette propriété, il existe en effet un ensemble de 20 cartes ne contenant pas de set. Ceci fut découvert par David Van Brink à l'aide d'un ordinateur en 1997. Remarquons que cela n'était pas un calcul trivial pour les ordinateurs de l'époque (notons que $\binom{81}{21} \simeq 1.36 \times 10^{19}$...). En 2001, nul

autre que Donald Knuth [1] a voulu s'en convaincre et pour ce faire a écrit un programme vérifiant cette propriété de manière très efficace (via l'exploitation des symétries du problème), qu'il a rendu public [2].

Le jeu SET possède de nombreuses autres propriétés intéressantes d'un point de vue mathématique mais celles-ci sortent du cadre de ce projet ; le lecteur interessé n'aura aucun mal à trouver des articles à ce sujet sur le web.

Organisation

Pour toute question portant sur ce projet, n'hésitez pas à contacter par e-mail ou à rencontrer — prenez rendez-vous ! — le titulaire du cours ou la personne de contact de la partie concernée.

Titulaire. Gwenaël Joret – gjoret@ulb.ac.be – O8.111

Assistants.

Partie 1 : Jérôme De Boeck – jdeboeck@ulb.ac.be – N3.207

Partie 2 : Jacopo De Stefani – jacopo.de.stefani@ulb.ac.be – N8.210

Partie 3 : Fabio Sciamannini – fabio.sciamannini@ulb.ac.be – N3.202

Partie 4 : Nikita Veshchikov – nveshchi@ulb.ac.be – N8.213

Objectifs pédagogiques

Ce projet *transdisciplinaire* permettra de solliciter vos compétences selon différents aspects.

- Des connaissances vues aux cours de programmation, langages, algorithmique ou mathématiques seront mises à contribution, avec une vue à plus long terme que ce que l'on retrouve dans les divers petits projets de ces cours. L'ampleur du projet requerra une analyse plus stricte et poussée que celle nécessaire à l'écriture d'un projet d'une page, ainsi qu'une utilisation rigoureuse des différents concepts vus aux cours.
- Des connaissances non vues aux cours seront nécessaires, et les étudiants seront invités à les étudier par eux-mêmes, aiguillés par les *tuyaux* fournis par l'équipe encadrant le cours. Il s'agit entre autres d'une connaissance de base des interfaces graphiques en `Python 3`.
- Des compétences de communication seront également nécessaires : à la fin de la partie 4, les étudiants remettront un rapport expliquant leur analyse, les difficultés rencontrées et les solutions proposées. Une utilisation correcte des outils de traitement de texte (utilisation des styles, homogénéité de la présentation, mise en page, *etc.*) sera attendue de la part des étudiants. Une orthographe correcte sera bien entendu exigée.
- En plus d'un rapport, les étudiants prépareront une présentation orale, avec *transparents* ou *slides* (produits par exemple avec LaTeX, `LibreOffice Impress`, `Microsoft PowerPoint`), ainsi qu'une démonstration du logiciel développé. À nouveau, on attendra des étudiants une capacité à présenter et à vulgariser leur projet (c'est-à-dire rendre compréhensible leur présentation pour des non informaticiens, ou en tout cas pour des étudiants ayant eu le cours de programmation mais n'ayant pas connaissance de ce projet-ci).

En résumé, on demande aux étudiants de montrer qu'ils sont capables d'appliquer des concepts vus aux cours, de découvrir par eux-mêmes des nouvelles matières, et enfin de communiquer de façon scientifique le résultat de leur travail.

Consignes générales

A relire avant chaque remise d'une partie du projet !

- L'ensemble du projet est à réaliser en `Python 3`.
- Le projet est organisé autour de quatre grandes parties

1. Un des pères fondateurs de l'informatique ; si vous rencontrez ce nom pour la première fois un court détour par sa page wikipedia s'impose !
2. `http://www-cs-faculty.stanford.edu/~uno/programs.html`

- En plus de ces quatre parties à remettre, chaque étudiant doit préparer une *présentation orale* de 7 minutes, ainsi qu'une *démonstration* de 3 minutes ; celles-ci se dérouleront durant la seconde moitiée du mois d'avril.
- Chacune des quatre parties du projet compte pour 20 points. La présentation compte également pour 20 points en tout, ce qui fait un total de 100 points.
- Chacune des quatres grandes parties devra être remise sous deux formes :
  - via l'Université Virtuelle (`http://uv.ulb.ac.be`), qui contient une tâche pour chaque partie ;
  - sur papier, au secrétariat étudiants du Département d'Informatique.
- Après chaque partie, un correctif sera proposé. Vous serez libres de continuer sur base de ce correctif mais nous vous conseillons de plutôt continuer avec votre travail en tenant compte des remarques qui auront été faites.
- Les « Consignes de réalisation des projets » (cf. `http://www.ulb.ac.be/di/consignes_projets_INFO1.pdf`) sont d'application pour ce projet individuel. *(Exception : Ne tenez pas compte du point 3.1.3 de ces consignes concernant les archives zip, voir ci-dessous pour les consignes de soumission des fichiers sur l'UV).* Vous lirez ces consignes en considérant chaque partie de ce projet d'année comme un projet à part entière. Relisez-les régulièrement !
- Si vous avez des questions relatives au projet (incompréhension sur un point de l'énoncé, organisation, *etc.*), n'hésitez pas à contacter le titulaire du cours ou la personne de contact de la partie concernée, et non votre assistant de TP.
- **Il n'y aura pas de seconde session pour ce projet !**

Veuillez noter également que le projet vaudra **zéro** sans exception si :

- le projet ne peut être exécuté correctement via les commandes décrites dans l'énoncé ;
- les noms de fonctions sont différents de ceux décrits dans cet énoncé, ou ont des paramètres différents ;
- à l'aide d'outils automatiques spécialisés, nous avons détecté un plagiat manifeste (entre les projets de plusieurs étudiants, ou avec des éléments trouvés sur Internet). Insistons sur ce dernier point car l'expérience montre que chaque année une poignée d'étudiants pensent qu'un petit copier-coller d'une fonction, suivi d'une réorganisation du code et de quelques renomages de variables passera inaperçu... Ceci sera sanctionné d'une note nulle pour toutes les personnes impliquées, sans discussion possible. Afin d'évitez ce genre de situations, veillez en particulier à ne pas partager de bouts de codes sur forums, pages Facebook, etc.

Consignes pour la soumission de fichiers sur l'UV

Une dernière consigne, d'ordre technique, concerne la soumission de vos fichiers sur l'UV lors de la remise des différentes parties : Pour chaque partie, nous vous demandons de

- créer localement sur votre machine un répertoire de la forme `NOM_Prenom` (exemple : `DUPONT_Jean`) dans lequel vous mettez tous les fichiers à soumettre
- compresser ce répertoire via un utilitaire d'archivage produisant un .zip
- de soumettre le fichier archive .zip, et *uniquement* ce fichier, sur l'UV

Cette étape d'archivage s'avère nécessaire car l'UV modifie automatiquement les noms des fichiers envoyés en y ajoutant le nom de l'étudiant. En particulier l'instruction `import setFunctions.py` ne fonctionnerait pas si chaque fichier était soumis séparément (puisque le fichier `setFunctions.py` aurait alors été renommé par l'UV).

Bon travail !

1 Partie 1 : Création des structures de données et fonctions de base

L'objectif de cette première partie est de mettre en place les structures de données et fonctions de base du jeu. (Il faudra attendre la partie 2 pour avoir un jeu complètement fonctionnel ; en particulier, la boucle principale sera implémentée lors de la partie 2).

1.1 Structures de données

Les variables à utiliser dans votre projet doivent respecter scrupuleusement les *noms* et les *types* fournis dans ce document. Libre à vous de créer d'autres variables si vous estimez cela utile. Pensez alors à justifier rapidement vos choix dans les commentaires de votre code.

Paramètres

L'ensemble des nombre de symboles, symboles, remplissages et couleurs possibles pour une carte seront définis dans des variables globales de type dictionnaire. Nous utiliserons comme dictionnaires et valeurs possibles pour chaque attribut :

1. `elements` qui contient les nombres 1, 2, et 3 pour le nombre de symboles sur une carte ;
2. `symbols` qui contient les codes unicode (en décimal) des trois symboles utilisés (l'utilisation des caractères unicodes en Python sera expliquée plus loin). Vous êtes libres de personnaliser la représentation et donc de choisir ces caractères unicode. Faites cependant attention à choisir des symboles qui peuvent se distinguer facilement. Dans l'exemple donné plus loin, les codes unicode 9762, 9822 et 9806 ont été utilisés (symboles "radioactivité", "cheval", et "balance" respectivement).
3. `fillings` qui contient les charactères représentant le remplissage (plein, ligné, ou vide). Nous prendrons les lettres `'P'`, `'L'` et `'V'` respectivement comme exemple dans ce texte mais vous pouvez choisir d'autres représentations.
4. `colors` qui contient les codes couleur utilisés (voir plus loin pour l'usage des balises couleurs). Les couleurs habituelles du jeu Set—rouge, bleu et vert—correspondent aux codes couleur `31`, `32` et `34` respectivement. Libre à vous d'utiliser d'autres couleurs (du moment que cela reste lisible).

Les clés de chacun de ces 4 dictionnaires doivent valoir 0, 1 et 2. Par exemple, avec les couleurs rouge, bleu et vert, les couleurs seraient représentées par le dictionnaire

```
colors = {0 : 31, 1 : 32 , 2 : 34}
```

Le code couleur du rouge serait dans ce cas stocké dans `colors[0]`.

Variables

Une carte sera représentée par une variable de type *list*. Cette liste contiendra 4 entiers qui représenteront dans l'ordre les clés du nombre de symbole, du symbole, du remplissage et de la couleur de la carte.

Par exemple une carte représentant deux balances lignés rouges sera représenté par la liste `[1,2,1,0]`.

Trois variables importantes qui seront manipulées durant une partie de Set sont :

1. une variable `board` de type *list* qui représente les cartes qui sont retournées sur la table.
2. une variable `deck` de type *list* qui représente le tas des cartes qui n'ont pas encore été jouées.
3. une variable `graveyard` de type *list* qui représente l'ensemble des cartes qui ont été éliminées.

1.2 Fonctions

Une série de fonctions de base vous est demandée, libre à vous d'implémenter des fonctions supplémentaires.

1. `init()` : return `(deck, board, graveyard)`
   Initialise la variables du programme `deck`, `board` et `graveyard` et les retourne sous forme d'un tuple.
   Cette fonction devra placer les `81` cartes différentes dans la variable `deck`, c'est-à-dire toutes les listes à quatre éléments contenant les valeurs 0, 1 ou 2. Vous pouvez initialiser la liste des cartes à l'aide de boucles imbriquées. Par la suite nous vous demanderons d'initialiser le `deck` via la compréhension de listes (vu au cours de programmation) mais ceci n'est pas obligatoire pour la partie 1.
   Elle mélange ensuite le `deck` à l'aide de la méthode `shuffle(list)` de la librairie `random`, initialise la variable `board` en y mettant les 12 premières cartes du `deck`, et les retire du deck.
   Finalement, elle initialise la variable `graveyard` à une liste vide.
2. `display_deck(deck)` : return `None`
   Affiche l'entièreté du `deck`, dix-huit cartes par ligne maximum. Les cartes sont séparées par un espace. La méthode pour afficher une carte au bon format est définie plus loin.
3. `display_board(board)` : return `None`
   Affiche le `board`. L'ensemble des cartes doit être affiché sur 3 colonnes. La première ligne contient les trois premières cartes, la deuxième ligne les trois suivantes, etc. Le board de départ doit ainsi être affiché sur 4 lignes. Dans chaque colonne, les premiers caractères de chaque carte doivent être alignés, peu importe la taille d'affichage des cartes. Le `board` doit ainsi former une grille où lignes et colonnes sont alignées.
   Note : dans cette partie du projet, il n'y a pas de retrait de carte, le `board` contiendra donc toujours 12 cartes. Il y aura lieu d'adapter cette fonction lors de la partie 2 pour gérer le cas où il y a moins de 12 cartes sur le `board`.
4. `select_set(board)` : return *list* ou `None`
   Permet de choisir un ensemble de 3 cartes dans le `board` et retourne une liste contenant les trois cartes sélectionnées. Les cartes sont référencées par leur numéro de ligne et leur numéro de colonne. La carte en haut à gauche est la carte 11 et celle en haut à droite est la carte 13. Pour sélectionner les cartes, l'utilisateur devra encoder leur numéro en les séparant par un espace, par exemple : "11 13 22".
   Une fois que l'utilisateur a encodé les valeurs, la fonction devra vérifier s'il y a bien trois valeurs encodées, si elles correspondent bien à des cartes du `board` et sont bien différentes. Si ce n'est pas le cas la fonction doit le signaler à l'utilisateur. Si l'encodage des cartes est correct, la fonction devra afficher les trois cartes et demander à l'utilisateur de confirmer son choix en tapant "c", ou de le rejeter en tapant tout autre caractère.
   Si l'utilisateur confirme, la fonction renvoie les trois cartes sous forme de liste. Si l'utilisateur ne confirme pas ou qu'il s'est trompé dans un de ses choix de carte, la fonction renvoie `None`.
5. `is_set(cards)` : return *bool*
   Cette fonction vérifie si la liste `cards` formée de 3 cartes forme bien un Set. Si c'est le cas la fonction renvoie `True`. Si `cards` ne contient pas 3 cartes ou ne forme pas un Set, elle renvoie `False`.
   Pour vérifier si `cards` forme bien un Set vous pouvez comparer un à un les différents attributs en vérifiant s'ils sont tous identiques ou tous différents via deux nouvelles fonctions. La figure 2 donne des exemples de listes représentent des cartes qui forment un Set ou non.

```
[0,1,0,0]      [2,1,2,1]      [2,0,1,2]  [1,1,2,0]        [0,2,2,0]
[0,2,2,1]      [0,1,1,1]      [0,1,0,2]  [0,1,2,1]        [1,0,2,1]
[0,0,1,2]      [1,2,0,1]      [1,2,2,2]  [2,1,2,2]        [2,1,0,0]

 Set ok !      Pas un set      Set ok !   Set ok !       Pas un set
           Erreur attribut 1                          Erreur attribut 2 & 3
```

FIGURE 2 – Exemple de Set

Pour les deux dernières fonctions (`select_set(board)` et `is_set(cards)`), il vous est demandé de commenter en détail chacune de leurs différentes étapes.

Affichage d'une carte

Une fonction `print_card(card)` devra être implémentée pour afficher une carte au format "{remplissage}{symbole(s)}" dans la bonne couleur. Par exemple `print_card`([1,2,1,0]) pourrait afficher .

Remarque : Dans cet exemple le remplissage est modélisé en utilisant les lettres `'P'`, `'L'` ou `'V'`. Cette notation n'est pas très esthétique mais n'est qu'une solution temporaire avant l'implémentation d'une interface graphique à la partie 3. Les symboles peuvent être obtenus à partir de la fonction `chr(i)` qui renvoie le symbole correspondant au code unicode *i*. Exemple :

`print(chr(9762))` ⟶ ☢

L'affichage d'une chaîne de charactères dans une couleur ayant un code couleur *c* demande une syntaxe particulière. Pour afficher `Hello World !` dans la couleur 31 (rouge) vous devez exécuter :

`print("\x1B[ 31mHello World !\x1B[0m")` ⟶ `Hello World !`

Vos chaînes de charactères doivent être précédées de "\x1B[*c*m" et suivies de "\x1B[0m" où *c* est le code de la couleur. Pour cela nous vous conseillons de définir une variable globale *CSI* ="\x1B[" et d'encadrer vos string qui doivent changer de couleur (code couleur *c*) à l'aide d'une assignation du type :

```
string = CSI+str(c)+"m" + string + CSI + "0m"
```

Remarque importante : certaines versions de Windows n'interpretent pas correctement les balises couleurs telles que décrites ci-dessus. Deux solutions possibles :
- installez Linux !
- n'utilisez pas de balises pour afficher les couleurs des cartes dans un premier temps (ou ajoutez un caractère `'R'`, `'B'` ou `'V'` servant de substitut). Une fois que votre code fonctionne sur votre ordinateur, venez le tester sur les ordinateurs des salles machines du bâtiment NO en rajoutant les balises appropriées autour des strings qui représentent une carte. Attention à nous remettre la bonne version de votre code !

1.3 Fichier à soumettre

L'entièreté de votre code devra être implémenté dans un fichier `set.py`. Lors de l'exécution de votre fichier en ligne de commande, votre fichier devra initialiser les variables `deck`, `board` et `graveyard` en ayant pris le soin de mélanger le `deck` avant de choisir les cartes du `board`. Un menu devra ensuite afficher le `board` de départ et proposer à l'utilisateur de :

1. afficher le `deck` dans l'ordre actuel
2. mélanger le `deck`
3. choisir dans le board un ensemble de trois cartes et vérifier si c'est un set
4. quitter le jeu

Rappel : sur l'UV vous devez soumettre l'archive .zip que vous aurez produite, comme expliqué dans les consignes, et uniquement cette archive !

La figure 3 montre un exemple d'exécution du programme. En plus des commentaires visant à expliquer votre code, veillez à apporter une attention particulière à ceux des fonctions `select_set(board)` et `is_set(cards)`.

1.4 Rendu

Personne de contact : Jérôme De Boeck – jdeboeck@ulb.ac.be – N3.207
Remise : Vendredi 13 novembre 2015 à 13 heures.
A remettre : Les scripts en `Python3` :
- Au secrétariat étudiants : une version imprimée de ces scripts ;
- Sur l'UV (`http://uv.ulb.ac.be/`) : une version électronique de ces mêmes scripts.

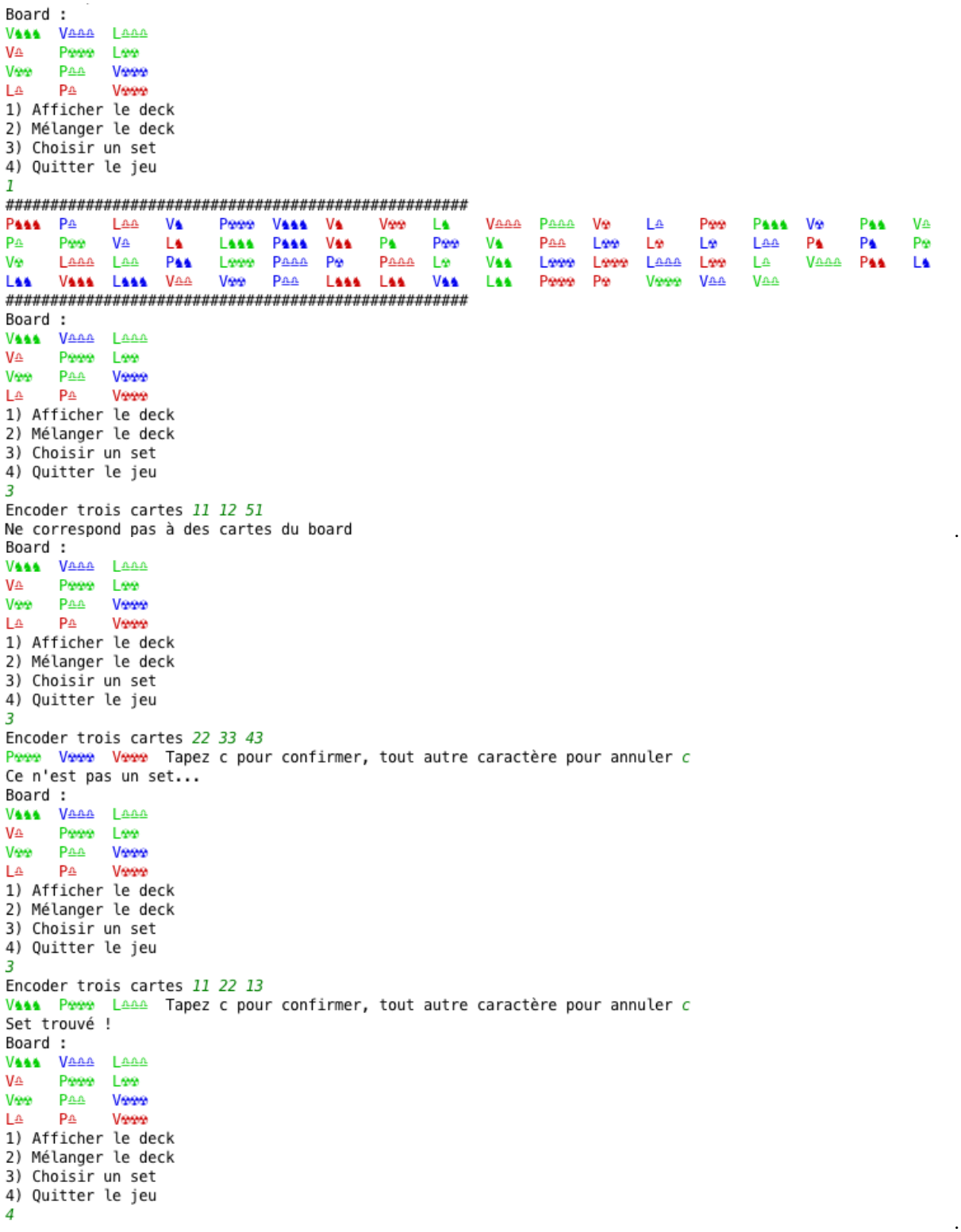

FIGURE 3 – Exemple d'exécution du programme

2 Partie 2 : Jeu fonctionnel

L'objectif de la partie 2 du projet est d'obtenir une première version du jeu fonctionnant sans interface graphique, c-à-d en ligne de commande. Pour atteindre cet objectif, nous vous demandons de réutiliser les structures de données et les fonctions développées dans la partie 1. Plus précisément vous devrez :

- Implémenter la boucle principale du jeu, c-à-d :
  - Afficher le tableau de jeu
  - Gérer l'interaction entre l'utilisateur et le jeu.
  - Réagir à la sélection des cartes par le joueur.
  - Mettre à jour l'état du jeu (*deck*, *graveyard* et *board*) selon les règles décrites dans la section *Présentation générale*.
- Mesurer le temps écoulé depuis le début de la partie

Vous écrirez la boucle principale dans un fichier `set.py`, et toutes les autres fonctions dans un fichier `setFunctions.py`.

2.1 Boucle principale

Algorithme 1 Pseudocode boucle principale du jeu

```
1: Initialisation jeu
2: while !(Conditions Terminaison) do
3:   Affichage board
4:   Interaction avec le joueur
5:   if (Joueur veut quitter le jeu) then
6:     Terminaison jeu
7:   end if
8:   if (Joueur déclare absence set) then
9:     if (Jeu confirme absence set) then
10:      Cas I - Vérification terminaison jeu
11:    else
12:      Cas II - Penalité
13:    end if
14:  end if
15:  if (Joueur choisit set) then
16:    if (Jeu vérifie set) then
17:      Cas III - Avancement du jeu
18:    else
19:      Cas IV - Mauvais choix
20:    end if
21:  end if
22: end while
23: Affichage informations
```

Le programme démarre avec l'*initialisation* de toutes les structures de données nécessaires pour l'exécution du jeu, via un appel à la fonction `init()` définie lors de la partie 1.

L'implémentation du jeu *Set* que nous vous proposons est une version un joueur, au tour par tour. Le joueur peut quitter la partie à chaque tour de jeu. (Nous ne vous demandons pas ici d'implémenter de sauvegarde de l'état du jeu). S'il décide de ne pas quitter, à chaque tour de jeu le joueur aura deux choix : soit déclarer l'absence d'un set, soit choisir un ensemble de trois cartes sur le tableau de jeu qui, selon lui, forment un set. Suite à ce choix, quatre situations différentes sont possibles :

1. Utilisateur déclare l'absence d'un set - Absence vérifiée (Cas I)
2. Utilisateur déclare l'absence d'un set - Un set existe (Cas II)
3. Utilisateur choisit un set - set vérifié (Cas III)
4. Utilisateur choisit un set - l'ensemble des cartes n'est pas un set (Cas IV)

A chacune de ces possibilités correspond un cas d'éxécution différent que vous devrez implémenter dans une fonction comme expliqué dans les sections suivantes. Vous devrez respecter les signatures des fonctions (c-à-d le type et ordre des paramètres et le type et ordre des valeurs retournés par la fonction) indiqués dans les sections suivantes.

2.1.1 Terminaison du jeu

Une partie peut se terminer pour deux raisons : soit le *deck* est épuisé (il ne contient plus aucune carte) et l'ensemble des cartes restantes dans le *board* ne contient plus de set, soit simplement parce-qu'il ne reste plus de cartes en jeu (auquel cas le *graveyard* contient 81 cartes).

Une fois le jeu terminé, le programme doit afficher un message qui indique la raison pour laquelle le jeu s'est terminé, suivi par le nombre de cartes restantes dans le *deck*, le nombre de cartes dans le *graveyard* et sur le *board*, ainsi que la durée totale de la partie (voir Figure 4).

FIGURE 4 – Exemple de l'écran final du jeu

2.1.2 Vérification présence set

Nous vous demandons aussi d'écrire une fonction qui, étant donné une liste de cartes, permet de vérifier s'il existe au moins un set dans cet ensemble de cartes.

Signature de la fonction

`exists_set(card_list):` return True/False

où :
- `card_list` : liste de cartes qui represente le *board*
- la fonction retourne `True` s'il existe au moins un set dans la liste de cartes, False sinon.

2.1.3 Affichage du tableau de jeu

Le tableau du jeu devra être affiché en utilisant la fonction `display_board()`. Nous vous demandons d'afficher le tableau à une position fixe dans le terminal, en haut à gauche (voir Figure 5). Pour ce faire, vous devrez utiliser une commande du système d'exploitation qui permet d'effacer le contenu du terminal. Cette commande dépend de l'OS : `clear` sur les machines de type Unix (Linux, Mac OS X, Android), `cls` sous Windows.

Vous pouvez faire l'hypothèse que les correcteurs utiliseront une machine du premier type et simplement utiliser la commande `clear`. Vous pouvez aussi décider de rendre votre programme indépendant de l'OS (voir plus loin).

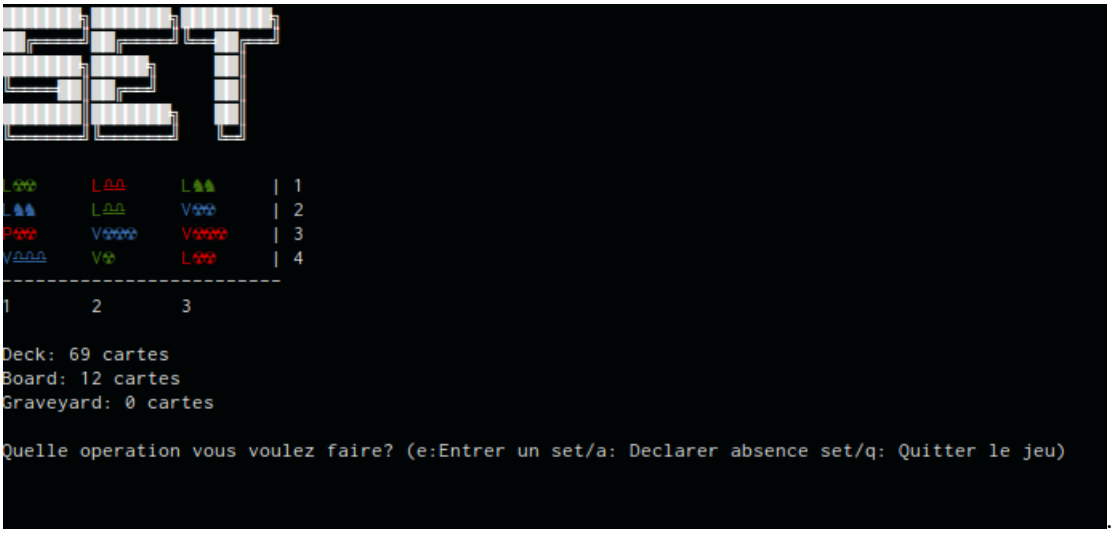

FIGURE 5 – Exemple de l'écran principal du jeu

Afin d'appeler une commande système dans un programme python, vous devrez importer la librairie `os` [3] dans votre programme et utiliser une de ses fonctions :

```
os.system(<commande>)
```

où `<commande>` doit être remplacé par la commande système que vous voulez appeler. Si vous souhaitez rendre votre programme portable sur différents systèmes d'exploitation, vous pouvez obtenir des informations sur l'OS que vous êtes en train d'utiliser via l'instruction :

```
os.name()
```

Cette fonction va par-exemple retourner `"posix"` si vous êtes sur Linux ou Mac OS X, et `"nt"` dans le cas où vous utilisez Windows.

Vous devez toujours afficher le *board* sur trois colonnes, en tenant compte que le nombre de lignes dépendra du nombre de cartes sur le *board*. Si celui-ci est inférieur à 12, vous indiquerez les cases vides du *board* ; vous pouvez par-exemple utiliser une croix (caractère unicode 10060), ou tout autre caractère approprié de votre choix. Pour simplifier l'affichage, les cases vides seront toujours en dernière position sur le *board*.

2.1.4 Cas I - Vérification terminaison jeu

Algorithme 2 Pseudocode vérification terminaison jeu

```
1: if Deck vide then
2:   Terminaison jeu
3: else
4:   Ajouter 3 premières cartes du Deck au Board
5:   Continuer jeu
6: end if
```

Si le joueur déclare l'absence d'un set dans le *board*, et si effectivement il n'existe aucun set, il faut alors procéder à la vérification de la terminaison du jeu (cf. Algorithme 2), qui dépend du contenu du *deck*.

Si le *deck* contient encore des cartes, on pioche les trois 3 premières cartes et on les ajoute au *board*. (L'ajout de ces cartes au *board* peut potentiellement introduire de nouveaux sets dans le *board* et, par conséquent, permettre au joueur de continuer la partie). Par contre, si le *deck* est vide, le jeu se termine.

3. `https://docs.python.org/3/library/os.html`

Signature de la fonction

`case_I(board,deck)`: return `(board,deck,end_game)`

où :
- `board` : Liste de cartes qui represente le board
- `deck` : Liste de cartes qui represente le deck
- `end_game` : Variable booleenne qui signale la terminaison du jeu

2.1.5 Cas II - Pénalité

Dans la version de Set que nous vous proposons, une pénalité est prévue si le joueur déclare qu'il n'y a pas de set sur le *board* alors qu'en réalité il en existe un. Cette pénalité consiste à piocher 3 cartes au hasard dans le *graveyard* (si non vide) et à les ajouter à la fin du *deck*. Ensuite, le jeu procède avec l'itération suivante de la boucle principale.

Signature de la fonction

`case_II(deck,graveyard)`: return `(deck,graveyard)`

où :
- `deck` : Liste de cartes qui represente le `deck`
- `graveyard` : Liste de cartes qui represente le `graveyard`

2.1.6 Cas III - Avancement du jeu

Ici le joueur a correctement détecté un set dans le *board*. Les trois cartes qu'il a sélectionnées sont déposées dans le *graveyard*. Si le *deck* n'est pas vide, les trois premières cartes du *deck* sont retirées et placées sur le tableau de jeu. Ensuite, le jeu procède avec l'itération suivante de la boucle principale.

Signature de la fonction

`case_III(board,deck,graveyard,selected_cards)`: return `(board,deck,graveyard)`

où :
- `board` : Liste de cartes qui represente le `board`
- `deck` : Liste de cartes qui represente le `deck`
- `graveyard` : Liste de cartes qui represente le `graveyard`
- `selected_cards` : Liste de cartes selectionnées par l'utilisateur.

2.1.7 Cas IV - Mauvais choix

Si le joueur choisit un ensemble de trois cartes qui ne forme pas un set, il faudra simplement afficher un message d'erreur et procéder avec l'itération suivante de la boucle principale. (Remarquons qu'il n'y a ici pas de pénalité pour un mauvais choix).

Signature de la fonction

`case_IV()`: return `None`

2.1.8 Affichage des informations du jeu

A la fin de la partie, nous vous demandons d'afficher sur le terminal :
- Un message qui décrit la raison pour laquelle le jeu s'est terminé.
- La durée de la partie (voir Section 2.2).
- Nombre de cartes dans le *deck*.
- Nombre de cartes sur le *board*.
- Nombre de cartes dans le *graveyard*.

2.2 Gestion du temps

La performance d'un joueur est mesurée en calculant le temps qu'il a pris pour terminer une partie. Nous vous demandons de calculer ce temps en secondes. Afin d'implémenter cette fonctionnalité, vous devrez utiliser la librairie `datetime` [4] de Python. Par exemple, l'instruction suivante

```
datetime.datetime.now()
```

renvoit un objet qui contient la date et l'heure courante.

Notons que nous ne vous demandons pas d'afficher le temps écoulé en temps réel pendant le jeu. Ceci nécessiterait le concept de multithreading—qui sort du cadre de ce projet—enseigné aux cours INFO-F201 (Systèmes d'exploitation) et INFO-F202 (Langages de programmation 2). Il est simplement demandé de calculer le temps écoulé entre le début et la fin de la partie.

Remarque : Si ceci est votre première rencontre avec le jeu de cartes Set, il est probable que vos premières parties soient longues, typiquement 20-30 minutes. Cependant avec un peu d'entrainement vous arriverez rapidement en-dessous de la barre des 10 minutes.

2.3 Fichiers à soumettre

A l'exception de la boucle principale, toutes les fonctionnalités décrites ci-dessus, et éventuellement vos fonctions supplémentaires, doivent être incluses dans un fichier nommé `setFunctions.py`. Chaque fonction sera accompagnée d'un bref commentaire décrivant ce qu'elle fait.

La boucle principale devra elle être présente dans un fichier séparé nommé `set.py`.

Rappel : sur l'UV vous devez soumettre l'archive .zip que vous aurez produite à partir de vos fichiers, comme expliqué dans les consignes, et uniquement cette archive !

Veuillez à respecter scrupuleusement les signatures des fonctions décrites ci-dessus, tout projet ne les respectant pas sera sanctionné d'une note nulle. (C'est en effet essentiel pour que vos correcteurs puissent facilement tester votre code !)

2.4 Rendu

Personne de contact : Jacopo De Stefani – jacopo.de.stefani@ulb.ac.be – N8.210
Remise : Vendredi 11 décembre 2015 à 13 heures.
À remettre : Les scripts en `Python3` :
– Au secrétariat étudiants : une version imprimée de ces scripts ;
– sur l'UV (`http://uv.ulb.ac.be/`) : une version électronique de ces mêmes scripts.

4. `https://docs.python.org/3.5/library/datetime.html`

3 Partie 3 : Interface graphique

La troisième partie du projet consiste en la mise en œuvre d'une *interface graphique* (*Graphical User Interface* en anglais, d'où GUI) pour votre logiciel qui permettra à l'utilisateur d'interagir avec votre programme de manière plus conviviale. Dans ce but, Python fournit par défaut une bibliothèque nommée `TkInter`. D'autres bibliothèques graphiques pour `Python` existent également, telles que `PyQt`, `PyGTK` ou encore `wxPython`, mais nous vous demandons d'utiliser `TkInter` dans le cadre de ce projet afin de faciliter le travail des correcteurs. Par ailleurs, nous vous demandons qu'au terme de cette troisième partie, votre code gère les erreurs produites lors de l'exécution à l'aide de gestion d'exceptions comme vu au cours de programmation.

Remarque générale concernant les dessins des cartes : vous êtes entièrement libres de choisir la représentation que vous souhaitez pour les cartes du jeu Set, il n'est aucunement demandé d'imiter les dessins de la version commercialisée. Au contraire, la créativité est encouragée ! (NB : Si vous utilisez des dessins faits par des tierces personnes, attribuez-les explicitement à leur(s) auteur(s) quelque part dans votre programme, par-exemple dans un menu *'about'*).

3.1 Structure et fonctionnement d'une GUI

Une GUI est typiquement composée d'un ensemble d'éléments nommés *widgets* agencés ensembles. Une fenêtre, un bouton, une liste déroulante, une boîte de dialogue ou encore une barre de menus sont des exemples de widgets typiques. Il peut aussi s'agir d'un *conteneur* qui a pour rôle de contenir et d'agencer d'autres widgets.

Une GUI va typiquement être structurée, de manière interne, sous la forme d'un *arbre* de widgets. Dans `TkInter`, tout programme doit avoir un widget « racine », qui est une instance de la classe `Tkinter.Tk`, créant en fait une fenêtre principale, dans laquelle on peut ensuite ajouter d'autres widgets. À l'appel d'un constructeur de widget, il vous faudra toujours préciser en paramètre quel sera son widget parent (exception faite du widget racine, bien sûr).

Réaliser la structure visuelle d'une GUI (aussi appelée *maquette*) ne constitue que la moitié du travail, il faut que votre code des deux parties précédentes puisse être également lié à cette interface. À l'exécution, une GUI va réagir aux *événements*. Par exemple, un clic sur un bouton. Votre programme ne va donc plus être réduit à se lancer, effectuer des opérations et se terminer. Le principe d'une GUI est d'exécuter une boucle qui va traiter les événement jusqu'á la terminaison de l'application. `TkInter` offre notamment la possibilité de lier le clic sur un bouton à l'appel d'une fonction donnée (voir le paramètre `command` du constructeur de la classe `Tkinter.Button` par exemple). Il vous faudra savamment utiliser ces possibilités pour mettre à jour l'affichage au besoin.

3.2 Exemple de maquette et contraintes

La Figure 6 vous donne un exemple de maquette pour l'interface graphique.

Il doit être possible de fermer votre programme en cliquant sur le bouton de fermeture de la fenêtre. Notez qu'il ne vous est pas imposé de réaliser l'interface en anglais comme sur la Figure.

Veillez à ce que votre interface graphique soit ergonomique, agréable visuellement et simple d'utilisation.

Remarque sur la performance. Si vous liez une fonction qui nécessite un certain temps de calcul à un bouton de votre interface, celle-ci restera « gelée » après l'événement (clic) jusqu'à ce que la fonction termine son exécution. Dans le cadre de ce projet, nous nous contenterons de ce comportement (et ne le pénaliserons certainement pas lors de la notation). Pour éviter ce phénomène, il faut exploiter le concept de *multithreading* (qui sort du cadre de ce projet) enseigné aux cours INFO-F201 (Systèmes d'exploitation) et INFO-F202 (Langages de programmation 2).

3.3 Références utiles

La documentation disponible á l'adresse suivante pourra vous aider à développer votre interface graphique avec `TkInter` : `http://wiki.python.org/moin/Tkinter`

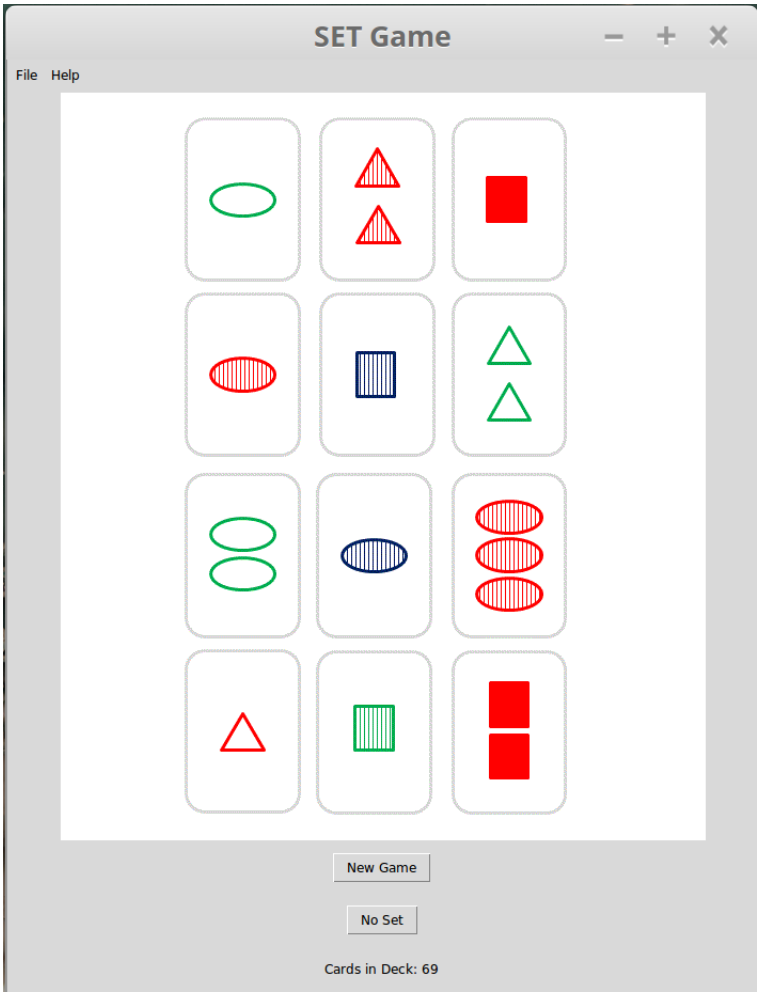

FIGURE 6 – Exemple : canevas de la fenêtre principale de `GUI.py`.

3.4 Comportement du GUI

Le canevas du réseau. L'utilisateur sélectionne les cartes à l'aide d'un click gauche de la souris. Une fois une carte sélectionnée, celle-ci doit être mise en évidence (*highlight*) sur le dessin. A chaque tour l'utilisateur choisit au plus trois cartes. S'il sélectionne trois cartes et que celles-ci forment un set, trois nouvelles cartes du *deck* les remplacent. Sinon, un message d'erreur est affiché. L'utilisateur a également la possibilité de déclarer qu'il n'y a pas de set à l'aide d'un bouton approprié, dont le comportement est tel que décrit dans la partie 2. A nouveau, si l'utilisateur s'est trompé, il y aura lieu d'afficher un message d'erreur l'informant de la pénalité.

Le panneau d'information. Le panneau d'information affiche le nombre de cartes restantes dans le *deck*.

Le panneau de commande. Le panneau de commande se compose :

1. d'un bouton qui lance une nouvelle partie,
2. du bouton "No Set".

3.5 Dessin sur un canevas `TkInter`

Nous vous fournissons ici un algorithme qui vous permettra de dessiner des rectangles et des cercles dans un canevas `TkInter` (c'est-à-dire une instance de la classe `Tkinter.Canvas`).

```
myFrame = Frame(masterWidget) # création de la Frame
```

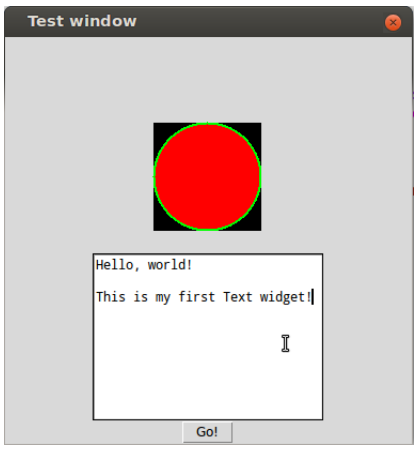

FIGURE 7 – Exemple de maquette *Tkinter*.

```
myCanvas = Canvas(myFrame, height=500, width=500) # définit la taille du canevas

# Dessine un rectangle noir de taille 100x100
# Le coin supérieur gauche se trouve en coordonnées (50, 80)
myCanvas.create_rectangle(50, 80, 150, 180, fill="black")

# Dessine un cercle (rouge avec un contour vert) inscrit dans le carré
myCanvas.create_oval(50,80,150,180, outline="green", fill="red", width=2)

# Place le frame et le canevas sur le root widget
# pour placer les objets sur un frame on peut aussi utiliser ma méthode grid
myCavnas.pack()
myFrame.pack()
```

Remarquez que les rectangles sont définis à l'aide de deux points : le coin supérieur gauche et le coin inférieur droit. Les ovales (ainsi que les cercles et les ellipses) sont définis à l'aide des rectangles (l'ovale est alors inscrit dans un rectangle).

De la même manière, on peut ajouter des objets de types `Button`, `Label`, `Menu`, `Text` (et beaucoup d'autres) sur un `Frame`. La Figure 7 montre un exemple d'une fenêtre créée à l'aide de `TkInter`. Mous vous invitons à explorer différents éléments graphiques qui existent dans `TkInter` afin de choisir les éléments qui conviennent le mieux pour votre interface. Il est également préférable d'utiliser des couleurs dans votre interface graphique.

3.6 Gestion d'exceptions

Pour cette phase, la gestion d'exceptions doit être implémentée. Nous nous attendons donc à ce que vous utilisiez pleinement et intelligemment les instructions `try-except-finally` ainsi que la commande `raise` pour gérer proprement les erreurs pouvant survenir lors de l'exécution de votre programme. En particulier, nous nous attendons à ce que les erreurs relatives aux données erronées donnent lieu à l'affichage de boîtes de dialogue qui décrivent ce qu'il s'est passé (utilisez pour cela le module `messagebox` fourni avec `TkInter`).

Vous trouverez un exemple sur la Figure 8.

3.7 Utilisation de classes

Dans la mesure du possible, nous souhaitons que vous fassiez un usage judicieux de classes (et donc exploiter les constructeurs, les attributs et les méthodes) pour structurer votre code. Nous vous demandons de créer au moins les classes suivantes :

- `"GUI"`- classe qui encapsule l'interface graphique ;
- `"Card"`- classe qui permet de représenter une carte du jeu ;

FIGURE 8 – Boîte de dialogue d'erreur.

– `"PlatformUtils"`- classe fournie qui possède deux méthodes afin d'identifier les boutons gauche et droit de la souris.

Il est également attendu de créer un fichier par classe.

3.8 Exécution du programme

Nous demandons que votre interface graphique soit lancée via l'exécution du fichier `GUI.py` (contenant la classe `GUI`), qui ne prendra aucun paramètre en ligne de commande. Notez cependant que votre programme console devra toujours fonctionner. Pensez à modulariser votre code de manière appropriée. (Pour rappel, aucun code d'exemple issu d'Internet, et/ou d'autres projets et/ou de livres n'est autorisé !)

Rappel : sur l'UV vous devez soumettre l'archive .zip que vous aurez produite à partir de vos fichiers, comme expliqué dans les consignes, et uniquement cette archive !

3.9 Rendu

Personne de contact : Fabio Sciamannini – fabio.sciamannini@ulb.ac.be – N3.202

Remise : Vendredi 4 mars 2016 à 13 heures.

À remettre : Les scripts et les autres fichiers nécessaires au fonctionnement du projet (par exemple des images ou autres fichiers texte ou scripts python que vous utiliseriez) :

– Au secrétariat étudiants : une version imprimée de vos scripts Python ;
– Sur l'UV (`http://uv.ulb.ac.be/`) : une version électronique de vos fichiers.

4 Partie 4 : Rapport et ajout de fonctionnalités supplémentaires

Dans cette partie nous vous proposons de revoir votre code afin d'en préparer une version finale, ajouter une fonctionnalité supplémentaire, et rédiger un rapport à propos de votre programme.

4.1 Fonctionnalité supplémentaire : highscore

Nous vous demandons d'intégrer la fonctionnalité suivante dans votre programme. Un joueur doit pouvoir consulter une liste des meilleurs jeux (top 3 des meilleurs temps). Cette liste doit être triée en ordre croissant, chaque entrée étant constituée du nom du joueur et du temps réalisé (mesuré en secondes). Elle doit être sauvée dans un fichier, si le programme est fermé et puis relancé on doit toujours pouvoir consulter les hiscores.

Quant un joueur termine le jeu en effectuant un meilleur temps qu'une des entrées dans la liste des hiscores, cette liste doit être mise à jour (en gardant toujours le top 3). Voici un exemple d'une liste de "highscores" :

```
Alice       250
Bob         320
Charlie     427
```

Si David joue et termine en 293 secondes, alors il doit prendre la place de Bob, ainsi Bob descend d'une position et Charlie disparait de la liste :

```
Alice       250
David       293
Bob         320
```

Votre programme doit demander le nom d'un joueur si et seulement si il a battu un des temps dans le top 3 et peut donc entrer dans cette liste. Un même joueur peut évidemment se retrouver dans la liste plus d'une fois. Si dans notre exemple précédent Alice joue encore une fois et termine en 200 secondes, alors la liste devient :

```
Alice       200
Alice       250
David       293
```

4.2 Autres fonctionnalités supplémentaires

Nous vous proposons d'ajouter également d'autres fonctionnalités supplémentaires dans votre projet. Cette partie n'est pas obligatoire mais peut vous rapporter des points bonus et sera fort appréciée (si faite convenablement). N'ajoutez ces fonctionnalités que si le reste de votre projet fonctionne parfaitement, autrement ces bonus ne seront pas pris en compte.

Voici une liste non exhaustive des fonctionnalités supplémentaires possibles. Les descriptions que nous en donnons sont intentionnellement vagues, la créativité est encouragée !

- Difficulté (nombre de cartes sur la table) : un joueur peut choisir la difficulté du jeu. En diminuant la difficulté le joueur voit plus de cartes sur la table, en l'augmentant il en voit moins.
- Difficulté (quantité d'attributs) : chaque carte possède plus ou moins d'attributs en fonction de la difficulté.
- Joker : si une des trois cartes est un joker, alors trois cartes forment automatiquement un set.
- Pause : un joueur doit pouvoir mettre le jeu en pause. Les cartes ne sont pas visibles durant la pause.
- Hints : on doit pouvoir utiliser un boutton `Hint` qui aide à trouver un set, en montrant deux des trois cartes d'un set par-exemple.
- Pénalités : Varier la pénalité dans le cas d'une erreur (moins de points, plus de temps, remise des cartes dans le jeux, etc).
- Tutoriel interactif pour apprendre à jouer.
- Vous pouvez en inventer d'autres !

Remarque : Le joueur doit toujours garder la possibilité de jouer au jeu 'classique' de Set (tel que décrit à la partie 3, avec les hiscores en plus). Si vous décidez d'implementer ces bonus, vous devrez également les décrire dans le rapport.

4.3 Rapport

En plus du code, nous vous demandons de rédiger un rapport d'environ 5 pages (page de garde, annexes et table des matières non comprises). Ce rapport devra contenir les éléments suivants :

- une page de garde qui reprend vos coordonnées, la date, le titre, *etc.* ;
- une table des matières ;
- une introduction et une conclusion ;
- des sections décrivant ce qui a été réalisés ;
- des exemples d'exécution du jeu ;
- éventuellement des références bibliographiques si applicable.

Le rapport doit détailler les éventuelles difficultés rencontrées, l'analyse et les solutions proposées, ainsi qu'une bonne explication des algorithmes (e.g. pseudo-code avec des exemples et/ou des diagrammes, pas de code source). Il doit être clair, et compréhensible pour un étudiant imaginaire qui aurait suivi le cours INFO-F-101 (Programmation) mais n'aurait pas fait le projet ; ni trop d'informations ni trop peu. Toutes les parties doivent être détaillées et présentées dans un ordre logique. Expliquez toutes les notions et terminologie que vous introduisez.

Le rapport peut être écrit soit en LaTeX, soit à l'aide des logiciels de traitement de texte `LibreOffice Writer`, `OpenOffice Writer` ou `Microsoft Word`. Il est obligatoire d'utiliser les outils de gestion automatique des styles, de la table de matière et de numérotation des sections. Nous vous conseillons d'utiliser le système LaTeX, très puissant pour une mise en page de qualité. Il est évident qu'une bonne orthographe sera exigée.

Rappel : sur l'UV vous devez soumettre l'archive .zip que vous aurez produite à partir de vos fichiers (code et rapport), comme expliqué dans les consignes, et uniquement cette archive !

4.4 Rendu

Personne de contact : Nikita Veshchikov – nveshchi@ulb.ac.be – N8.213

Remise : Vendredi 25 mars 2016 à 13 heures.

À remettre : L'ensemble des scripts en `Python3` nécessaires au fonctionnement de votre projet, ainsi que le rapport final :

- Au secrétariat étudiants : une version imprimée de ces scripts, et du rapport final ;
- Sur l'UV (`http://uv.ulb.ac.be/`) : une version électronique de ces mêmes scripts et du rapport final, dans son format source (.doc(x), .odt, ou .tex) et sa version PDF.