TP n°7 : corrigé

PTSI Lycée Eiffel

février 2015

Ce corrigé ne comporte que des commentaires des programmes Python regroupés dans le fichier demineur.py disponible à l'adresse habituelle (dont vous venez normalement si vous êtes en train de lire ces lignes).

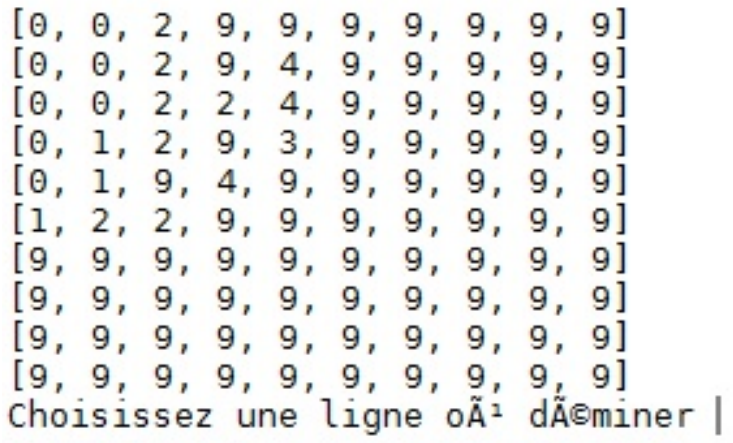
1 Quelques fonctions sur les tableaux.

1. Le principe du programme est simple : on compare la longueur de chacune des lignes du tableau (c'est-à-dire des éléments t[i], qui sont eux-même des listes) à celle de la première ligne. On profite bien entendu de la structure des définitions de fonctions en Python : dès qu'une ligne n'est pas de la même longueur que la première, on sort du programme en retournan False (rappelons que le return arrête immédiatement l'exécution). Si cette éventualité ne s'est jamais produite, c'est qu'on est arrivés jusqu'à la dernière ligne sans encombre, on peut alors retourner la valeur True.
2. Le calcul de la longueur de ligne maximale est très similaire à celui de la valeur maximale dans une liste, vu en cours. La complétion consiste ensuite simplement, pour chaque ligne du tableau, à ajouter un nombre de zéros égal à l'écart entre la longueur de cette ligne et la longueur de la ligne la plus longue.
3. Encore du classique : on parcourt tous les éléments du tableau (la variable i parcourt les lignes, la variable j les éléments à l'intérieur d'une ligne), et on incrémente la variable n à chaque fois qu'on croise un élément égal à x. Pour avoir la liste des coordonnées, on est obligés de modifier un peu les boucles pour que les variables i et j prennent comme valeur les indices des listes (dans le programme précédent, i était une liste, j un élément de cette liste), et on ajoute la construction d'une liste l de coordonnées.
4. Pas grand chose à commenter ici, i parcourt les lignes du tableau, on additionne simplement les longueurs (a priori variables) de ces lignes.
5. Très similaire au précédent.

2 Application à la programmation d'un démineur.

1. La liste l (définie ligne 81) est simplement constituée de toutes les coordonnées possibles dans une matrice à n lignes et p colonnes (numérotées à partir de 0, comme toujours en Python). La matrice t est une matrice n lignes p colonnes initialement remplie de zéros (il y a d'autres façons de créer une telle matrice, mais attention, certaines peuvent vous procurer de mauvaises surprises ensuite). Rappelons que la commande sample du module random (importée à la ligne 78) permet de piocher au hasard un échantillon de k éléments différents au sein d'une liste. On choisit donc ici k coordonnées aléatoires dans la grille, oùon place ensuite des chiffres 9 (la syntaxe de la ligne 85 est un peu étrange, mais i est ici une liste de deux nombres, i[0] correspond à la première coordonnées, c'est-à-dire au numéro de la ligne dans la matrice, et i[1] au numéro de colonne).

2. J'ai d'abord écrit un programme créant la liste des coordonnées des cases voisines de la case (i,j) dans une matrice de taille n fois p. Ce programme voisins est très laborieux, puisqu'il gère d'abord les cas particuliers des quatre coins de la grille (qui n'ont que trois cases voisines rentrées « à la main »), puis ceux des cases situés sur un bord de la grille (première ligne, dernière ligne, première colonne, dernière colonne) qui n'ont que cinq cases voisines, et enfin le reste (dans le else de la ligne 108) où on aura huit cases voisines. Ce n'est pas très beau mais ça marche ! Compter le nombre de voisins qui sont des 9 est ensuite simple : on parcourt la liste des cases voisines, et on incrémente un compteur à chaque fois qu'on croise un 9.
3. On utilise bien évidemment ici le programme précédent, ainsi que la possiblité en Python de définir directement des listes en compréhension, ce qui permet d'écrire la définition de fonction sur une seule ligne.
4. Le programme demineur1 se contente de suivre à la lettre la structure proposée par l'énoncé. On demande donc au joueur les dimensions de la grille et le nombre de mines, on crée une grille t (ligne 131) qui ne servira presque plus ensuite, une deuxième grille u (ligne 132) qui contient le nombre de voisins mines de chaque case, et enfin la grille g (ligne 133) qui sera la grille du joueur, initialement remplie de 9, qui évoluera au fur et à mesure que le joueur progresse dans son déminage. La variable m (ligne 134) représente le nombre de cases restant à déminer, et la variable x (ligne 135) restera égale à 1 tant que le joueur n'a pas sauté sur une mine. Tant qu'il reste des cases à déminer et qu'on n'a pas sauté (ligne 136), on affiche la grille, et on demande au joueur de choisir une case à déminer (lignes 140 et 141, les -1 en fin de ligne servent à ramener la numérotation des lignes et des colonnes à partir de 1, ce qui est plus confortable pour le joueur). Si la case a déjà été explorée (la condition de la ligne 142 ne sera pas vérifiée) on ne fait rien. Sinon, on regarde si la case est une mine (ligne 143, si c'est le cas, le joueur explose et on va sortir de la boucle avec le message d'échec de la ligne 149), sinon on remplace la case correspondante par son nombre de mines adjacentes dans la grille du joueur (ligne 146) et on met à jour le nombre de cases restant à déminer. Si on atteint le message de la ligne 151, c'est qu'on est sortis de la boucle sans avoir explosé, on a donc gagné. Un exemple de ce que donne l'exécution de ce programme (vous noterez un léger problème d'encodage que je n'ai pas cherché à régler) :



5. Pas grand chose à dire, la variable z représente le choix du joueur entre les deux actions possibles, le reste n'a pas beaucoup évolué. Cette nouvelle version n'est d'ailleurs pas très jouable car les 'M' dans la grille prennent plus de place en largeur que les chiffres, il faudrait améliorer l'affichage.
6. Le programme nettoyage est particulièrement sale : il sert simplement à repérer dans la grille du joueur les cases qui sont voisines de cases où se trouve un zéro et qui n'ont pas encore été déminée, pour les déminer automatiquement. La variable a représente bêtement le fait qu'on trouve ou non de telles cases dans la grille. Si a=0, c'est qu'il n'y en a plus, on peut arrêter le nettoyage. En pratique, ce nettoyage prendra donc du temps, puisqu'il faudra plusieurs passages

dans la grille : au premier on démine les cases voisines des 0, au suivant les cases voisines des nouveaux zéros apparus etc. On fait un dernier tour où il ne se passe rien avant de se rendre compte qu'il est temps d'arrêter. Le programme demineur3 qui suit reprend la même structure que demineur2, avec un mettoyage effectué à chaque tour. J'ai toutefois supprimé la variable m comptant le nombre de cases restant à déminer car elle n'était pas pratique à mettre à jour en cas de nettoyage. À la place, le programme s'arrête quand il n'y a plus de 9 dans la grille, ce qui a l'inconvénient que le joueur ne sera considéré comme gagnant que s'il n'y a plus de cases à déminer ET qu'il a mis des drapeaux sur toutes les cases contenant des mines. En fait, c'est même pire que ça puisque le programme ne se rendra pas compte que le joueur peut avoir mis des drapeaux sur des cases contenant des mines ! Je vous laisse corriger ce bug si vous êtes courageux.

7. Non, je ne fais pas partie des plus forts en Python, je zappe cette question. En fait, créer l'interface graphique n'est pas très compliqué avec un module dédié, ce qui est plus pénible est de gérer les clics à la souris du joueur. Si vous y tenez, vous trouverez sans difficulté de très beaux démineurs Pygame sur le web (pour ceux que ça intéresse, signalons que démineur se dit « minesweeper » en anglais).

3 Un autre jeu faisant intervenir des tableaux.

Pas grand chose à signaler sur ce dernier programme, j'ai recopié les programmes voisinsmines et remplissage pour qu'ils comptent le nombre de 1 voisins d'une case donnée au lieu du nombre de 9, et le programme jeudelavie se contente de faire n étapes où il fait évoluer les cases de la matrice suivant les règles énoncées. Ce programme très rudimentaire n'a qu'un intérêt très limité, mais je n'irai pas plus loin dans ce corrigé (pour le faire, il faut évidemment créer au préalable une matrice initiale t constituée de 0 et de 1).