Devoir 3
SR01

Vivien LECLERCQ – Jimmy LUONG

Exercice 1 :

On s'intéresse ici à la modélisation d'un jeu de la vie en Python.

Introduction :

Nous avons décidé d'utiliser la programmation en orientée objet, plus simple et plus compréhensible pour cet exercice.

Notre programme ne contient qu'une seule classe *MaFenetre*. Cette classe hérite de la classe *Tk* de la librairie tkinter qui permet de modéliser des interfaces graphiques simples en Pyhton.

Initialisation de notre classe :

Lors de l'appelle du constructeur de notre classe, notre programme s'initialise en appelant le constructeur de la superclasse. Puis, il initialise avec des valeurs par des défauts, les attributs de notre classe. Ces attributs sont les suivants :

- *size* : taille par défaut de la fenêtre
- *speed* : vitesse de notre moteur par défaut
- *percents_life* : pourcentage de vie dans notre moteur par défaut
- *grid_size* : taille de la grille de notre moteur
- *initialized* : booléen pour savoir si le moteur a été initialisé
- *running* : booléen pour savoir si le moteur est en train de fonctionné.

Ces valeurs par défauts sont conformes à la capture d'écran présentée dans le sujet.

Initialisation des variables :

Lors de l'appel du constructeur de *MaFenetre*, la méthode *creer_widget()* est appelée. Cette dernière permet de créer les éléments graphiques de notre programme à savoir : un canvas, des boutons et des scales.

En ce qui concerne le canvas, celui-ci est vide à sa création.

Pour ce qui est des boutons, chaque bouton dispose d'une fonction de callback qui lui est propre. Par exemple la bouton « Lancer » dispose de la méthode *lancer()* comme fonction de callback.

Enfin, pour les scales, chaque scale dispose de sa propre fonction de callback qui n'est autre qu'une méthode de notre classe *MaFenetre*. Ces fonctions sont appelées dès que l'utilisateur modifie la valeur du scale en question. Ces méthodes vérifient toujours que le moteur n'est pas en fonctionnement avant de modifier les attributs de la classe par le biais du booléen *running* qui doit être à False de manière à ne pas perturber le fonctionnement du moteur. Par

exemple, le scale sur la vitesse dispose de la méthode onSpeedChanged comme fonction de callback et est appelée lorsque l'utilisateur modifie la vitesse.

Initialisation de notre moteur :

Notre moteur s'initialise par le biais de notre méthode *initialiser()*. Cette dernière est appelée lors d'un clic sur le bouton « Initialiser ».

Il est important de préciser qu'une initialisation du moteur, et donc une modification des valeurs du moteur et de l'interface graphique, n'a de sens que lorsque que le moteur est à l'arrêt de manière à ne pas perturber le fonctionnement d'un moteur en cours d'exécution. D'où la vérification de notre attribut *running* à False.

Notre méthode create_grid() permet de remplacer une grille par une nouvelle dans notre canvas. Elle lit la valeur du scale « Taille de la grille » et divise la taille de la grille par cette valeur pour connaître la taille d'une cellule. Puis elle trace, dans le canvas, des lignes horizontales et verticales à intervalles réguliers.

La méthode *init_matrice()* permet de créer une matrice A de taille correspondant à la valeur du scale « Taille de la grille » et de la remplir aléatoirement par des '0' ou des '1' en fonction de la valeur du scale « Pourcentage de vie ». Nous utilisions la librairie Numpy pour ce faire. Une copie de A est aussi créée dont son utilité sera présentée par la suite. Ces matrices sont des attributs de notre classe.

Ensuite, la méthode *show_matrice()* permet d'afficher la matrice A. Elle commence par supprimer toutes les cellules de la grille. Puis, à l'aide d'une double boucle for, elle explore toutes les cases de la matrice. Pour chaque case de la matrice on appelle la méthode *create_cell(i, j, alive)*

Enfin, la méthode *create_cell(i, j, alive)* permet d'afficher une cellule en fonction d'une position (*i* et *j*) et d'une valeur booléenne (*alive* qui est en réalité un entier qui vaut soit '1' soit '0'). Il s'agit tout simplement de la création d'un rectangle rouge ou blanc (selon la valeur de *alive*) à la bonne place dans la grille.

Une fois ces méthodes appelées l'attribut *initialized* passe à True.

Lancement de notre moteur :

Notre moteur se lance par le biais de notre méthode *lancer()*. Cette dernière est appelée lors d'un clic sur le bouton « Lancer ».

Il est impératif de noter que notre moteur se lance que s'il a été initialisé au préalable. Cette vérification passe par l'attribut *initialized* qui doit valoir True lors de l'appel de cette méthode pour pouvoir lancer le moteur.

Notre moteur est en fait une boucle tant que la variable *running* est à True. Cette boucle explore chacune des cases de la matrice A à l'aide de deux boucles for. Pour chaque case,

puisque nous avons une matrice torique, nous faisons une abstraction sur (j moins un), (j plus un), (i moins un) et (i plus un). Pour chacune des cases de A nous comptons le nombre de cellules voisines en vie. Ensuite, en fonction des règles du jeu de la vie donnée en TD, nous modifions les nouvelles valeurs de la matrice dans la matrice B. La matrice B est une variable temporaire qui permet d'enregistrer les valeurs des A au fur et à mesure car si pour chaque case on modifie directement A, le système n'a plus de sens. A la fin de la boucle, nous affectons la nouvelle valeur de A, nous attendons grâce à la méthode *sleep()* de la librairie time. Ce temps d'attente dépend de la valeur du scale « Vitesse », une vitesse nulle donne un temps d'attente de deux secondes et une vitesse de soixante donne un temps d'attente de zéro. Enfin, nous mettons à jour l'affichage de A par le biais de la méthode *show_matrice()* et l'interface graphique.

Arrêt de notre moteur :

Notre moteur s'arrête par le biais de notre méthode *arreter()* Cette dernière est appelée lors d'un clic sur le bouton « Arrêter ».

Cette méthode se contente de passer l'attribut *running* à False de manière à stopper le fonctionnement de notre moteur.

Il est important de noter que notre méthode *arreter()* n'arrête pas complétement le moteur mais qu'elle le met en pause. En effet arrêter puis relancer le moteur est possible en cliquant sur le bouton « Arrêter » puis sur le bouton « Lancer ». Il est ainsi possible d'arrêter le moteur, de changer la vitesse, puis de relancer.

Quitter notre programme :

Notre programme s'arrête par le biais de notre méthode *quitter()* Cette dernière est appelée lors d'un clic sur le bouton « Quitter ».

Cette méthode se contente de passer l'attribut *running* à False de manière à stopper le fonctionnement de notre moteur.

Puis elle appelle la méthode *quit()* hérité de la classe *Tk* de manière à fermer la fenêtre.

La fonction main :

En Python la fonction main est représentée de la manière suivante : « if __name__ == "__main__": ».

Dans cette fonction nous créons un objet de type *MaFenetre*, nous modifions son titre et nous appelons la méthode *mainloop()* de manière à prendre en charge les différents événements.

Conclusion :

Ainsi nous avons implémenter le jeu de la vie en Python de manière beaucoup plus simple et beaucoup plus structurée qu'en C par le biais de la programmation orientée objet.

Notre programme comporte quelques défauts à améliorer : pour certaine valeur de *grid_size* la grille affiche des cases blanches, il faudrait redimensionner notre fenêtre à la volée. Néanmoins, notre programme remplit les taches qui lui sont demandées dans le sujet.