TP – Programmation objet

Important : Vous devez vérifier à chaque fois que vous écrivez du code que celui-ci fonctionne. A vous d'écrire les tests unitaires et/ou les assertions nécessaires (ces tests n'ont pas besoin de se retrouver dans la version finale du travail que vous rendrez : ils peuvent être retirés ou mis en commentaire).

I – Top chrono !

On souhaite programmer une classe Chrono qui peut servir de chronomètre ou de minuteur pour de nombreuses applications.

1) Ecrire le constructeur de la classe qui hérite de la classe `threading.Thread` du module `threading` (qu'il faudra importer). Pour l'instant cette classe ne fait rien mais doit appeler le constructeur de sa classe mère. Par la suite, il faudra rajouter l'initialisation des attributs que nous introduirons au fur et à mesure de la programmation de la classe.

Le module threading permet de créer et gérer des **fils de traitement** (ou « **thread** » en anglais). Ce sont des programmes qui s'exécutent en parallèle du programme principal[1]. Ils doivent posséder une méthode `run` qui ne possède pas d'argument et constitue le programme principal du fil d'exécution (celui qui va s'exécuter en parallèle du programme principal). Cette méthode ne sera jamais appelée directement. C'est la méthode `start` de la classe `Thread` qui se charge de l'appeler après avoir fait tout un tas d'opérations permettant de s'assurer de l'exécution de la méthode `run` en tant que fil d'exécution indépendant.
Voir le lien http://www.xavierdupre.fr/app/teachpyx/helpsphinx/c_parallelisation/thread.html pour davantage d'informations.

2) Exécuter le programme principal. Quelle commande permet l'affichage de tous les attributs et les méthodes de la classe `Chrono` ? Lister ces méthodes et attributs et vérifier qu'il y a bien une méthode `start` et une méthode `run` (ce seront les seules que nous utiliserons).

3) Ecrire une surcharge de la méthode `start` pour la classe `Chrono` qui initialise un attribut privé `debut` à la valeur de l'heure système actuelle (récupérée à l'aide de la fonction `time` du module `time`) et un autre attribut privé `en_action` à `True`. **Très important** : La dernière instruction de la méthode `start` doit être l'appel à la méthode `start` de sa classe mère.

4) Ecrire une surcharge de la méthode `run` qui attend (en utilisant la fonction `sleep` du module `time`) tant que l'attribut `en_action` est à `True` et se termine dès qu'il est à `False`.

On vient de créer une boucle infinie car `en_action` n'est jamais remis à `False`. Avant de tester votre programme, il faut donc faire la question suivante.

5) Ecrire une méthode `stop` qui inscrit l'heure actuelle dans un attribut privé `fin`, arrête le chrono en mettant `en_action` à `False` et renvoie la durée écoulée depuis le déclenchement du chrono en secondes. Vérifier

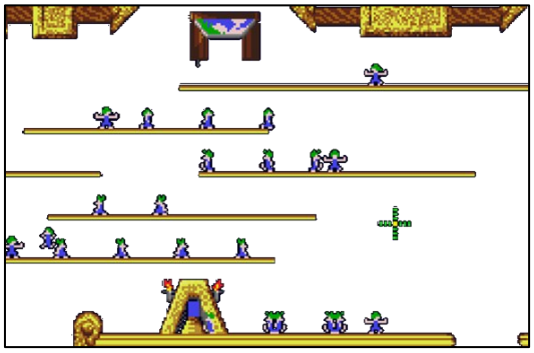
[1] En réalité seul un nombre de fils limité peuvent être exécutés réellement en parallèle (cela dépend du nombre de cœur du processeur), mais le système d'exploitation s'arrange pour simuler une exécution en parallèle.

en créant une instance de `Chrono` qu'on peut bien démarrer le chrono et l'arrêter. L'état du chrono peut être déterminé en visualisant la valeur de l'objet chrono dans la console.

6) Ecrire une méthode `get_time` qui renvoie la durée écoulée depuis le déclenchement du chrono (en secondes), sans arrêter le chrono s'il est en marche et qui renvoie la durée mesurée par le chrono (entre le `start` et le `stop`) si le chrono est à l'arrêt.

7) On souhaite ajouter une méthode `pause` qui met le chrono en pause et renvoie le temps écoulé jusqu'ici si le chrono est en fonctionnement et une méthode `reprise` qui fait repartir le chrono s'il était en pause (et n'a pas d'effet si le chrono était en marche) et renvoi le temps écoulé. Proposer une façon d'implémenter ces fonctionnalités et faire les modifications nécessaires à la classe.

II - Jeu des Lemmings[2]



On se propose de réaliser un programme inspiré du jeu Lemmings. Dans ce jeu, les lemmings marchent dans une grotte représentée par une grille à deux dimensions dont chaque case est soit un mur, soit un espace vide, un espace vide pouvant contenir au maximum un lemming à un instant donné. Les lemmings apparaissent l'un après l'autre à une position de départ, et disparaissent lorsqu'ils atteignent une case de sortie. Chaque lemming a une coordonnée verticale et une coordonnée horizontale désignant la case dans laquelle il se trouve, ainsi qu'une direction (gauche ou droite). Les lemmings se déplacent à tour de rôle, toujours dans l'ordre correspondant à leur introduction dans le jeu, de la manière suivante :

- si l'espace immédiatement en-dessous est libre, le lemming tombe d'une case ;
- sinon, si l'espace immédiatement devant est libre (dans la direction du lemming concerné), le lemming avance d'une case ;
- enfin, si aucune de ces deux conditions n'est vérifiée, le lemming se retourne.

Notre programme doit permettre de voir évoluer une colonie de lemmings. On aura une classe `Lemming` pour les lemmings, une classe `Case` pour les cases de la grotte, et une classe principale `Jeu` pour les données globales.

⇒ La classe principale `Jeu` contient un attribut `grotte` contenant un tableau à deux dimensions de cases, et un attribut `lemmings` contenant un tableau des lemmings actuellement en jeu. Son constructeur initialise la grotte, à partir d'un fichier texte où `#` représente un mur, un espace une case vide et `S` la case de sortie. La position de départ des lemmings sera toujours la deuxième case de la première ligne.
Cette classe fournie notamment les méthodes suivantes :

- `affiche` affiche la carte avec les positions et directions de tous les lemmings en jeu ;
- `tour` fait agir chaque lemming une fois et affiche le nouvel état du jeu ;
- `demarre` lance une boucle infinie attendant des commandes de l'utilisateur. Exemples de commande : `l` pour ajouter un lemming, `q` pour quitter et `<entrée>` pour jouer un tour.

⇒ La classe `Lemming` contient des attributs entiers positifs `l` et `c` indiquant la ligne et la colonne auxquelles se trouve le lemming, et un attribut `d` valant `1` si le lemming se dirige vers la droite et `-1` s'il se dirige vers la gauche. Il sera utile d'avoir un attribut `j` pointant sur l'instance de la classe `Jeu` pour laquelle le lemming a été créé, pour accéder au terrain et à la liste des lemmings. Le constructeur de la classe prend en argument les `l`, `c` et `j` pour pouvoir initialiser les attributs comme il se doit.
Cette classe fournie en outre les méthodes suivantes :

- `__str__` renvoie `'>'` ou `'<'` selon la direction du lemming ;
- `action` déplace ou retourne le lemming ;
- `sort` retire le lemming du jeu.

[2] Exercice tiré du livre « Numérique et sciences Informatique – Terminale » de T. Balabonski, S. Cochon, J-C Filliâtre et K. Nguyen

⇒ La classe `Case` contient un attribut `terrain` contenant le caractère représentant la caractéristique de la case (mur, vide, sortie), et un attribut `lemming` contenant l'éventuel lemming présent dans cette case et `None` si la case est libre.
Cette classe fournie les méthodes suivantes :

- `__str__` renvoie le caractère à afficher pour représenter cette case ou son éventuel occupant ;
- `libre` renvoie `True` si la case peut recevoir un lemming (elle n'est ni un mur, ni occupée) ;
- `depart` retire le lemming présent de la case ;
- `arrivee` prend en argument un lemming `lem` et le place sur la case ou le fait sortir du jeu si la case est une sortie.

Avec toutes ces indications, à vous de programmer ces 3 classes et de faire tourner le jeu sur la grotte donnée en exemple (fichier « grotte.txt »). Vous pouvez partir du fichier « TP_Partie_II_Trame.py » pour vous aider.

III - Bonus : La classe Minuteur

En s'inspirant de la classe Chrono, écrire une classe Minuteur qui permette de créer un objet minuteur auquel on indique la durée de décompte et qui appelle une fonction donnée en argument quand le décompte est terminé (ou affiche un texte indiquant la fin du décompte si aucune fonction n'est fournie).