Projet collectif : Le jeu de la vie

Le jeu de la vie a été inventé par le mathématicien britannique John H. Conway (1937-2020). C'est un exemple de ce qu'on appelle un **automate cellulaire**. Il se déroule sur un tableau rectangulaire $(L \times H)$ de cellules. Une cellule est représentée par ses coordonnées x et y qui vérifient $0 \leqslant x < L$ et $0 \leqslant y < H$.

Une cellule peut être dans deux états : **vivante** ou **morte**. La dynamique du jeu s'exprime par les règles de transition suivantes :
– une cellule vivante reste vivante si elle est entourée de 2 ou 3 voisines vivantes et meurt sinon ;
– une cellule morte devient vivante si elle possède exactement 3 voisines vivantes.

La notion de « voisinage » dans le jeu de la vie est celle des 8 cases qui peuvent entourer une case donnée (on parle de voisinage de Moore). Pour implémenter la simulation, on va tout d'abord donner une modélisation objet du problème, puis procéder à son implémentation.

Partie A Modélisation objet

1. Quelles classes peut-on dégager de ce problème au premier abord ?

2. Quelles sont quelques-unes des méthodes qu'on pourrait leur donner ?

3. Dans quelle classe pouvons-nous représenter simplement la notion de voisinage d'une cellule ? Et le calculer ?

4. Une cellule est au bord si $x = 0$, $x = L - 1$, $y = 0$ ou $y = H - 1$. Combien de voisins possède une cellule qui n'est pas au bord ?
Combien de voisins possède une cellule qui est au bord ?

5. Que pourrions-nous aussi considérer comme voisin de droite de la case en haut à droite de la grille ? Et comme voisin du haut ?

Partie B Implémentation des cellules

1. a. Implémenter tout d'abord une classe `Cellule` avec comme attributs :
- un booléen `actuel` initialisé à `False` ;
- un booléen `futur` initialisé à `False` ;
- une liste `voisins` initialisée à `None`.

CONSEIL

Ces attributs seront ici considérés comme « privés ». La valeur `False` signifie que la cellule est morte et `True` qu'elle est vivante.

b. Ajouter les méthodes suivantes :
- `est_vivant()` qui renvoie l'état actuel (vrai ou faux) ;
- `set_voisins()` qui permet d'affecter comme voisins la liste passée en paramètre ;
- `get_voisins()` qui renvoie la liste des voisins de la cellule ;
- `naitre()` qui met l'état futur de la cellule à `True` ;
- `mourir()` qui permet l'opération inverse ;
- `basculer()` qui fait passer l'état futur de la cellule dans l'état actuel.

c. Ajouter à la classe `Cellule` une méthode `__str__()` qui affiche une croix (un `X`) si la cellule est vivante et un tiret (`-`) sinon. Expliquer brièvement l'utilité d'une telle méthode `__str__()` en Python.

d. Ajouter une méthode `calcule_etat_futur()` dans la classe `Cellule` qui permet d'implémenter les règles d'évolution du jeu de la vie en préparant l'état futur à sa nouvelle valeur.

2. Implémenter la classe `Grille`.

a. Créer la classe `Grille` et y placer les attributs suivants considérés comme « publics » :
- `largeur` ;
- `hauteur` ;
- `matrix` : un tableau de cellules à 2 dimensions (implémenté en Python par une liste de listes).

Fournir une méthode `__init__` permettant l'initialisation d'une `Grille` de `Cellules` avec une `largeur` et une `hauteur` (une nouvelle `Cellule` sera créée par l'appel `Cellule()`).

b. Ajouter les méthodes :
- `dans_grille()` qui indique si un point de coordonnées i et j est bien dans la grille ;
- `setXY()` qui permet d'affecter une nouvelle valeur à la case (i, j) de la grille ;
- `getXY()` qui permet de récupérer la cellule située dans la case (i, j) de la grille ;
- `get_largeur()` qui permet de récupérer la largeur de la grille ;
- `get_hauteur()` qui permet de récupérer la hauteur de la grille ;
- `est_voisin()`, une méthode statique qui vérifie si les cases (i, j) et (x, y) sont voisines dans la grille.

c. Ajouter une méthode `get_voisins()` qui renvoie la liste des voisins d'une cellule.

d. Fournir une méthode `affecte_voisins()` qui affecte à chaque cellule de la grille la liste de ses voisins.

e. Donner une méthode `__str__()` qui permet d'afficher la grille sur un terminal.

f. On veut remplir aléatoirement la `Grille` avec un certain taux de `Cellule` vivantes. Fournir à cet effet, une méthode `remplir_alea()` avec le taux (en pourcentage) en paramètre.

3. On joue à présent !

a. Concevoir une méthode `jeu()` permettant de passer en revue toutes les `Cellules` de la `Grille`, de calculer leur état futur, puis une méthode `actualise()` qui bascule toutes les cellules de la `Grille` dans leur état futur.

b. Programme principal : définir enfin un `main` pour terminer l'implémentation du jeu de la vie avec un affichage en console en utilisant les méthodes précédentes. On donne la méthode suivante qui permet d'effacer l'écran dans un terminal `ANSI` :

```
def effacer_ecran():
    print("\u001B[H\u001B[J")
```

AIDES :

Partie A Modélisation objet

1. On peut proposer une classe `Cellule` représentant l'état d'une cellule et son évolution, et une classe `Grille` représentant le plateau de jeu.

2. Dans la classe `Cellule`, il faut pouvoir faire naître ou mourir la cellule, connaître son état actuel et futur et l'afficher sous la forme d'un caractère. Dans `Grille`, il faut pouvoir connaître sa largeur et sa hauteur, récupérer et réaffecter le contenu (la cellule) qui se trouve dans une position (i, j).

3. La notion de voisinage se calcule bien dans la grille mais une cellule doit aussi connaître ses voisins (ou leur nombre) pour calculer son état futur.

4. Une cellule qui n'est pas au bord admet toutes les cases qui l'entourent comme voisins. Il suffit de les compter. Distinguez les cellules dans les angles des autres cellules du bord.

5. Imaginez une grille qui se replie sur elle-même.

Partie B Implémentation des cellules

1. a. et **b.** Les attributs sont donnés et les méthodes sont pour la plupart traitables en 1 seule ligne. Ne pas oublier de mettre `self` en premier argument des méthodes !

c. Simple test : on renvoie le caractère demandé selon l'état de la cellule.

d. On implémente les règles d'évolution du jeu de la vie dans cette méthode en mettant seulement à jour l'attribut `futur`. Si possible ne pas utiliser d'accès aux attributs privés de la classe.

2. a. Implémentation de la grille : la méthode `__init__()` permet de mettre en place les attributs `largeur`, `hauteur` et `matrix` et de les initialiser. Utilisez une compréhension pour initialiser aisément `matrix` qui est un tableau de cellules à deux dimensions.

b. Pour `est_voisin()` qui prend en entrée deux cases de la grille, on souhaite exprimer que ces deux cases diffèrent d'au plus une unité en abscisse ou en ordonnée. Pensez au décorateur `@staticmethod`.

c. On passe en revue toutes les cases contiguës (8 voisins potentiels) de la case (x, y) de la grille et, si elles sont bien dans la grille, on les accumule dans une liste. On renvoie la liste obtenue.

d. Parcourez les cellules de la grille et affectez leurs voisins avec le résultat fourni par la méthode `get_voisins` de `Grille`.

e. Affichez ligne par ligne les cellules de la grille.

f. Utilisez la méthode `random()` qui renvoie un nombre aléatoire entre 0 et 1 puis faites naître et basculer les cellules tirées au sort.

3. a. On passe en revue toutes les cellules de la grille et on fait ce qui est demandé dans les deux cas.

b. Pensez à faire dans l'ordre : l'instanciation, le remplissage aléatoire, le calcul des voisinages, puis jouez en marquant une pause grâce à la méthode `sleep()` du module `time`.

Term NSI **GRILLE D'EVALUATION :** **Noms :** **Note :** **/12**	Rien d'écrit	Ne fonctionne pas	Fonctionne imparfaitement	Fonctionne avec beaucoup d'aide	Fonctionne avec aide	Fonctionne sans aide	Fonctionne de manière optimale et avec les commentaires
Architecture générale du script							
Objet Cellule							
Objet Grille							
Organisation du groupe							