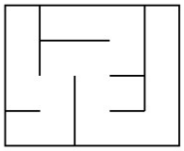
Labyrinthe

L'activité va permettre d'implémenter un programme qui crée et qui résout des labyrinthes.
La programmation orientée objet, les structures de données piles et les arbres sont utilisées.

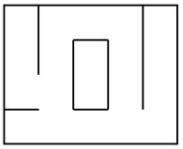
1. Présentation

L'objectif de ce mini-projet est de voir différents algorithmes permettant de construire des labyrinthes, puis des méthodes de résolution.

- Les labyrinthes étudiés sont dits **parfaits**, c'est à dire chaque cellule peut joindre une autre par un chemin unique.
- À l'inverse, les labyrinthes pouvant contenir des îlots, des boucles ou des cellules inaccessibles sont dits **imparfaits**, et ne seront pas étudiés ici.



Labyrinthe parfait



Labyrinthe imparfait

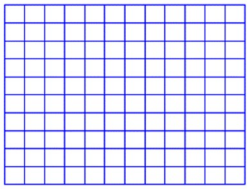
2. Grille

La construction du labyrinthe commence par la construction de la grille.

```
class Cellule:
    """ créer un objet cellule pour constituer une grille"""

    def __init__(self):
        """définie une cellule"""
        # lors de la création de la cellule, les quatre murs de la cellule existent
        self.murs = {'N': True, 'S': True, 'E': True, 'O': True}
        self.etat = False

    def get_N_cell(self):
        """retourne si le mur nord de la cellule existe"""
        return self.murs['N']

    def get_S_cell(self):
        """retourne si le mur sud de la cellule existe"""
        return self.murs['S']

    def get_E_cell(self):
        """retourne si le mur est de la cellule existe"""
        return self.murs['E']

    def get_O_cell(self):
        """retourne si le mur ouest de la cellule existe"""
        return self.murs['O']


class Grille:
    """créer une grille constituée de cellules"""

    def __init__(self, largeur, hauteur):
        """définie une grille de j lignes et i colonnes"""
        self.larg = largeur
        self.haut = hauteur
        self.grille = [[Cellule() for y in range(self.haut)] for x in range(self.larg)]

    def draw_grid(self):
        """retourne le dessin de la grille"""
        for y in range(self.haut):
            for x in range(self.larg):
                if self.grille[x][y].get_N_cell():
                    plt.plot([x, x+1], [y+1, y+1], 'b')
```

```
            if self.grille[x][y].get_S_cell():
                plt.plot([x, x+1], [y, y], 'b')
            if self.grille[x][y].get_E_cell():
                plt.plot([x+1, x+1], [y, y+1], 'b')
            if self.grille[x][y].get_O_cell():
                plt.plot([x, x], [y, y+1], 'b')
    plt.axis('off')
    return plt
```

✍ Vérifier le fonctionnement des classes **Grille** et **Cellule** en créant la grille (ci-contre) avec les commandes ci-dessous :



```
g = Grille(12, 10)
dessin_grille = g.draw_grid()
dessin_grille.show()
```

2.1. Classe Cellule

La classe **Cellule** permet de créer un objet cellule qui va constituer la grille.
Une cellule est constituée de quatre murs (Nord, Sud, Est et Ouest) et est caractérisée par son état (visité ou pas).

✍ Compléter la classe **Cellule** en rajoutant la méthode :

- **get_etat_cell** qui retourne l'état de la cellule.

2.2. Classe Grille

La classe **Grille** permet de créer un objet grille de i colonnes et de j lignes.

Dans le constructeur de la classe Grille, la grille est créée par compréhension :

```
self.grille = [[Cellule() for j in range(self.nb_lig)] for i in range(self.nb_col)]
```

✍ Remplacer cette méthode de création (liste par compréhension) de **grille** par une autre méthode.

✍ Commenter chaque ligne de la méthode **draw_grid**.

✍ Implémenter une méthode **cell_state** qui renvoie l'état d'une cellule de la grille suivant ses coordonnées. Cette méthode doit faire appel à la méthode **get_etat_cell** de la classe **Cellule**.

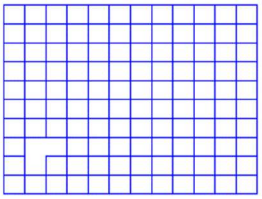
```
>>> g = Grille(12, 10)
>>> g.cell_state(2, 5)
False
```

✍ Implémenter une méthode **wall_cell_state** qui retourne la présence d'un mur de la cellule de coordonnées x, y. Se servir des méthodes de la classe **Cellule**.

```
>>> g = Grille(12, 10)
>>> g.wall_cell_state(2, 2, 'N')
True
```

✍ Implémenter une méthode **dell_wall_grid** qui efface le mur d'une cellule et qui efface aussi le mur correspondant de la cellule voisine.

```
>>> g = Grille(12, 10)
>>> g.dell_wall_grid(1, 1,'N')
>>> g.dell_wall_grid(1, 2,'E')
>>> g.wall_cell_state(1,2,'E')
False
>>> g.wall_cell_state(2,2,'O')
False
>>> dessin_grille = g.draw_grid()
>>> dessin_grille.show()
```



3. Construction du labyrinthe

Lors de la création des labyrinthes, les murs extérieurs ne seront jamais détruits (sauf pour l'entrée et la sortie du labyrinthe).

Il existe de nombreux algorithmes de construction de labyrinthes. Celui étudié correspond à une exploration exhaustive.

On part d'un labyrinthe ou tous les murs sont fermés.
Chaque cellule contient une variable booléenne indiquant son état (visité ou non).

Au début, on choisit aléatoirement une cellule dans la grille et on stocke sa position et on la marque comme visitée.
On cherche les cellules voisines possibles et non visitées.
S'il y a au moins une possibilité, on en prend une au hasard, on ouvre le mur et on recommence avec la nouvelle cellule.
S'il n'y en a pas, on revient à la case précédente et on recommence.
Lorsque l'on est revenue à la case de départ et qu'il n'y a plus de possibilité, le labyrinthe est terminé.

L'historique des emplacements des cellules peut être stocké dans une pile.
Cela peut se traduire par l'algorithme suivant :

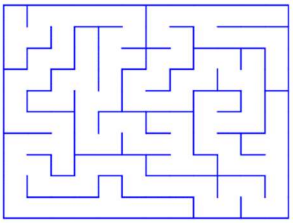
```
Choix aléatoire d'une cellule
On stocke la position de la cellule dans une pile
Tant que la pile n'est pas vide
        On récupère la cellule au sommet de la pile
        On indique que la cellule récupérée est visitée
        On recherche, pour la cellule récupérée, les orientations des cellules adjacentes non
        visitées
        S'il existe au moins une solution
                On choisit au hasard une orientation
                Pour l'orientation choisie, on ouvre le mur
                On stocke la position de la nouvelle cellule dans la pile
        Sinon
                On enlève de la pile la cellule
```

✍ Implémenter la fonction **`creat_laby(grille)`** qui prend en argument une grille vierge (ou tous les murs sont fermés) et qui retourne une grille correspondant au labyrinthe créé. Pour faciliter l'implémentation de cette fonction, il peut être utile de créer :

- La méthode **`visit_cell`** dans la classe **`Cellule`** pour indiquer que la cellule a été visitée.
- La méthode **`get_size_grid`** dans la classe **`Grille`** qui retourne les dimensions de la grille.
- Une classe **`Pile`** avec les méthodes classique d'une pile (**`sommet, pile_vide, depile, empile`**…).
- D'utiliser les méthodes **`randint`** et **`shuffle`** de la bibliothèque **`random`**.

Le résultat obtenu, à partir des instructions ci-dessous, doit ressembler à la figure ci-contre.

```
>>> g = Grille(12,10)
>>> laby = creat_laby(g)
>>> dessin_laby = laby.draw_grid()
>>> dessin_laby.show()
```



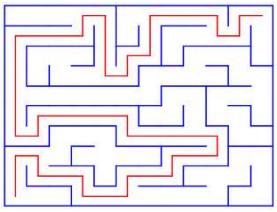
4. Résolution d'un labyrinthe

Comme pour la création d'un labyrinthe, il est possible de résoudre un labyrinthe en utilisant une structure de pile. La pile va permettre d'empiler les coordonnées des cellules parcourues.

Au début, le labyrinthe est initialisé avec aucune case visitée.
Pour chaque cellule qui entre dans la pile, on indique qu'elle est visitée.
A partir de la cellule de départ, on recherche les solutions de chemin possible (mur non présent et cellule non visitée).
On se dirige vers une cellule voisine possible jusqu'à arriver à la cellule d'arrivée.
Dans le cas d'une impasse, la cellule est sortie de la pile, et on recommence avec le sommet suivant.
La pile ainsi obtenue contient le parcours pour aller de la cellule de départ à la cellule d'arrivée.

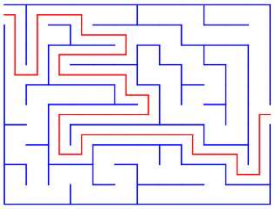
✍ Implémenter la fonction **`solve_laby`** qui prend en argument un labyrinthe et qui retourne une pile contenant la liste des cellules constituant la solution du labyrinthe.

✍ Implémenter la fonction **`draw_solve`** qui prend en argument un labyrinthe et sa solution (au format d'un pile) et qui retourne le dessin de la solution du labyrinthe (il faut définir par défaut une cellule de départ et une cellule d'arrivée).
Dans l'exemple ci-contre, la case de départ à les coordonnées (0, 0) et la case d'arrivée (à droite, en haut).



```
>>> g = Grille(12,10)
>>> laby = creat_laby(g)
>>> sol = solve_laby(laby)
>>> dessin_sol = draw_solve(laby, sol)
>>> dessin_sol.show()
```

5. Modifications



✍ Réaliser les changements au programme afin de choisir la case de départ à gauche et la case d'arrivée à droite. Pour ces deux cases, le mur ne doit être existant.
✍ Modifier le tracé de la solution en conséquence.

```
>>> g = Grille(12, 10, 10, 5)
```

6. Autre méthode de résolution

La recherche de solution peut s'apparenter à parcourir un arbre, l'arbre étant le labyrinthe avec comme nœud racine la case de départ.
✍ Implémenter une autre fonction pour résoudre le labyrinthe qui utilise la structure d'un arbre.

7. Interface

Réaliser une interface qui propose :

- de choisir les dimensions du labyrinthe,
- la visualisation du labyrinthe,
- la visualisation de sa solution,
- la création d'un fichier correspond au labyrinthe,
- la création d'un fichier correspondant à la solution du labyrinthe.

Source : https://fr.wikipedia.org/wiki/Mod%C3%A9lisation_math%C3%A9matique_de_labyrinthe
https://iut-info.univ-reims.fr/users/coutant/Creation_Resolution_Labyrinthe_Sujet.html