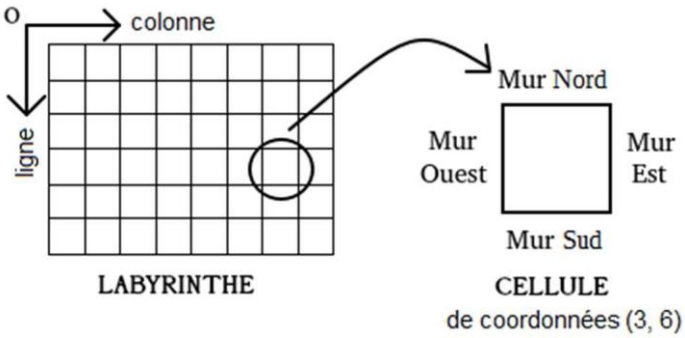
Labyrinthe

1. Objectif général



Un labyrinthe est composé de cellules possédant chacune quatre murs (voir ci-dessus). La cellule en haut à gauche du labyrinthe est de coordonnées (0, 0).

Le programme doit pouvoir dessiner dans la console des labyrinthes, sur la base d'une grille de dimension paramétrable

```
**********
*  *  *  *
**********
*  *  *  *
**********
*  *  *  *
**********
```

3 lignes x 3 colonnes

```
**********
*  *  *  *
**********
*  *  *  *
**********
*  *  *  *
**********
*  *  *  *
**********
```

4 lignes x 5 colonnes

2. Approche en programmation objet POO

On définit la classe `Cellule` ci-dessous. Le constructeur possède un attribut murs de type dict dont les clés sont 'N', 'E', 'S' et 'O' et dont les valeurs sont des booléens (`True` si le mur est présent et False sinon).

```
class Cellule:
    def __init__(self, murNord, murEst, murSud, murOuest):
        self.murs={'N':murNord,'E':murEst,'S':murSud,'O':murOuest}
```

1. Prévoir l'instruction Python suivante permettant de créer une instance cellule de la classe Cellule possédant tous ses murs sauf le mur Est.

```
cellule = Cellule(...)
```

2. Compléter la classe Cellule avec les méthodes __str_ et __repr__
3. Tester votre code puis coder une doctest

```
>>> a_cell = Cellule(True, True, True, True)
>>> a_cell
MURS: Nord:True Est:True Sud:True Ouest:True
>>> print(a_cell)
 __
|__|

>>>
```

```
# Une case du labyrinthe
class Cellule:
    '''Definir une cellule

    Une cellule est une case du labyrinthe
    '''
    def __init__(self, murNord, murEst, murSud, murOuest):
        '''Constructeur

        murs de type dict dont les clés sont 'N', 'E', 'S' et 'O'
        dont les valeurs sont des booléens (True si le mur est présent et False
sinon).
        '''
        pass

    def __repr__(self):
        '''Afficher existence des murs d'une cellule'''
        pass

    def __str__(self):
        '''Afficher dessin des murs d'une cellule

         __
        |__|
        '''
        pass
```

3. La classe Labyrinthe

3.1. Construire grille

Le constructeur de la classe Labyrinthe ci-dessous possède un seul attribut `grille`. La méthode `construire_grille` permet de construire un tableau à deux dimensions hauteur et longueur contenant des cellules possédant chacune ses quatre murs.

```
class Labyrinthe:
    '''Definir un labyrinthe'''

    def __init__(self, hauteur, longueur):
        '''Constructeur

        grille: list un tableau à deux dimensions hauteur et longueur
        contenant des cellules possédant chacune ses quatre murs.
        '''
        self.grille = self.construire_grille(hauteur, longueur)

    def construire_grille(self, hauteur, longueur):
        '''Construire une grille

        Renvoyer un tableau à deux dimensions hauteur et longueur
        contenant des cellules possédant chacune ses quatre murs.
        '''
        pass
```

- Coder la méthode construire_grille

- Coder la méthode __str__

Tester votre code avec un test en console :

```
>>> LIGNE = 2
>>> COLONNE = 3
>>> un_lab = Labyrinthe(LIGNE, COLONNE)
>>> un_lab
True True True True True True True True True True True True
True True True True True True True True True True True True

>>>
```

- Mettre en place un test automatique avec doctest

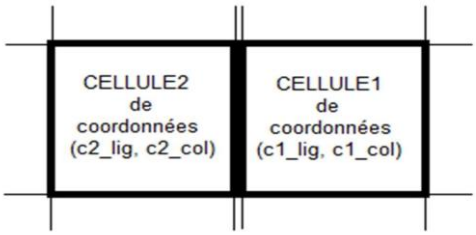
3.2. créer_passage

Pour générer un labyrinthe, on munit la classe `Labyrinthe` d'une méthode `creer_passage` permettant de supprimer des murs entre deux cellules ayant un côté commun afin de créer un passage. Cette méthode prend en paramètres les coordonnées `c1_lig`, `c1_col` d'une cellule notée `cellule1` et les coordonnées `c2_lig`, `c2_col` d'une cellule notée `cellule2` et crée un passage entre `cellule1` et `cellule2`.

```
13      def creer_passage(self, c1_lig, c1_col, c2_lig, c2_col):
14          cellule1 = self.grille[c1_lig][c1_col]
15          cellule2 = self.grille[c2_lig][c2_col]
16          # cellule2 au Nord de cellule1
17          if c1_lig - c2_lig == 1 and c1_col == c2_col:
18              cellule1.murs['N'] = False
19              ....
20          # cellule2 à l'Ouest de cellule1
21          elif ....
22            ....
23            ....
```

La ligne 18 permet de supprimer le mur Nord de cellule1. Un mur de cellule2 doit aussi être supprimé pour libérer un passage entre cellule1 et cellule2.

- Compléter le code pour traiter le cas où cellule2 est à l'Ouest de cellule1 :



- Tester la méthode avec un jeu de test adapté … par exemple

```
>>> LIGNE = 4
>>> COLONNE = 5
>>> un_lab = Labyrinthe(LIGNE, COLONNE)
>>> un_lab.creer_passage(0, 1, 0, 0)
>>> print(un_lab)
```

```
****************
*     *  *  *  *
****************
*  *  *  *  *  *
****************
*  *  *  *  *  *
****************
*  *  *  *  *  *
****************
```

Que l'on peut faire suivre par un autre test :

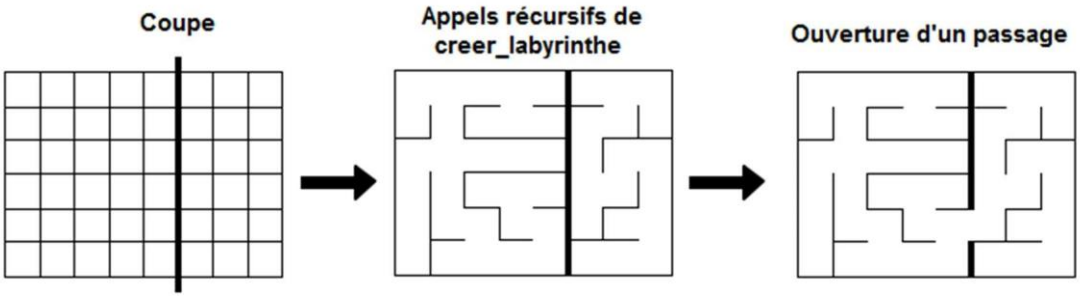
```
>>> un_lab.creer_passage(2, 1, 1, 1)
>>> print(un_lab)
```

```
**********
*  *  *  *
**********
* * * * * *
**** ******
* * * * * *
**********
* * * * * *
**********
```

- Mettre en place un test automatique avec doctest

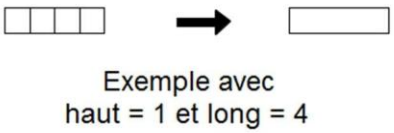
3.3. Créer_labyrinthe

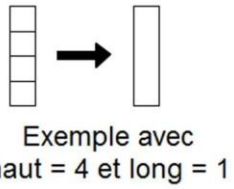
Pour créer un labyrinthe, on utilise la méthode diviser pour régner en appliquant récursivement l'algorithme `creer_labyrinthe` sur des sous-grilles obtenues en coupant la grille en deux puis en reliant les deux sous-labyrinthes en créant un passage entre eux.



La méthode `creer_labyrinthe` permet, à partir d'une grille, de créer un labyrinthe de hauteur haut et de longueur long dont la cellule en haut à gauche est de coordonnées (`ligne, colonne`).

Le cas de base correspond à la situation où la grille est de hauteur 1 ou de largeur 1. Il suffit alors de supprimer tous les murs intérieurs de la grille.





3.4. Réflexion sur le cas de base

Voici une première proposition pour traiter le cas de base. Cette proposition est-elle cohérente avec la méthode `creer_passage` étudié précédemment ?

```
24      def creer_labyrinthe(self, ligne, colonne, haut, long):
25          if haut == 1 : # Cas de base
26              for k in range(...):
27                  self.creer_passage(ligne, k, ligne, k+1)
28          elif long == 1: # Cas de base
29              for k in range(...):
30                  self.creer_passage(...)
31          else: # Appels récursifs
32             # Code non étudié (Ne pas compléter)
```

- Corriger puis compléter le cas de base
- Tester la méthode avec le jeu de test

```
LIGNE = 1
COLONNE = 6
un_lab = Labyrinthe(LIGNE, COLONNE)
un_lab.creer_labyrinthe(0, 0, LIGNE, COLONNE)
print(un_lab)
```

```
**********
*        *
**********
```

- Proposer un deuxième jeu de test pour valider tous les cas de base.

3.5. Appels récursifs

On se limite à des hauteurs et largeurs paires. On considère une grille de hauteur haut = 4 et de longueur long = 8 dont chaque cellule possède tous ses murs.
On fixe les deux contraintes supplémentaires suivantes sur la méthode `creer_labyrinthe` :

- Si `haut ≥ long`, on coupe horizontalement la grille en deux sous labyrinthes de même dimension.
- Si `haut < long`, on coupe verticalement la grille en deux sous-labyrinthes de même dimension.

L'ouverture du passage entre les deux sous-labyrinthes se fait le plus au Nord pour une coupe verticale et le plus à l'Ouest pour une coupe horizontale.

- Dessiner le labyrinthe obtenu suite à l'exécution de l'algorithme `creer_labyrinthe` sur cette grille, au début du premier appel récursif.
- Compléter la méthode `creer_labyrinthe` pour traiter les appels récursifs.
- Tester votre code avec le jeu de test

```
LIGNE = 1
COLONNE = 6
un_lab = Labyrinthe(LIGNE, COLONNE)
un_lab.creer_labyrinthe(0, 0, LIGNE, COLONNE)
print(un_lab)
```

3.6. Cas des dimensions impaires

- Modifier le code précédent pour pouvoir traiter le cas de hauteurs ou largeurs impaires.

- Mettre en place un test automatique avec doctest

3.7. Ouvrir un mur

Pour ouvrir une entrée ou une sortie dans le labyrinthe on ajoute une méthode `ouvrir_mur`

```
def ouvrir_mur(self, cel_lig, cel_col,haut, long):
    '''Ouvrir un mur exterieur

    Ouvre un mur sur la cellule désignée
    '''
```

- Prévoir une levée d'erreur si la cellule n'est pas sur un mur extérieur.
- Tester votre code dans différentes situation

- Mettre en place un test automatique avec doctest

```
>>> LIGNE = 4
>>> COLONNE = 4
>>> un_lab = Labyrinthe(LIGNE, COLONNE)
>>> un_lab.creer_labyrinthe(0, 0, LIGNE, COLONNE)
>>> un_lab.ouvrir_mur(3, 2, LIGNE, COLONNE)
>>> print(un_lab)
```

3.8. Créer les portes

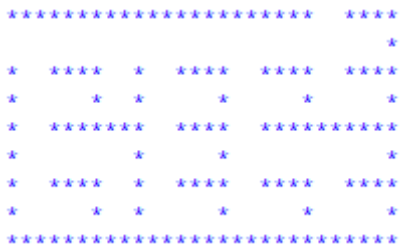
On ajoute maintenant une méthode `creer_portes` qui ouvrira deux murs, une entrée et une sortie.

```
def creer_portes(self, haut, long):
        '''Créer deux ouvertures au hasard dans un labyrinthe

        labyrinthe de hauteur haut et de longueur long
        dont la cellule en haut à gauche est de coordonnées (0, 0).
        '''
```

- Seules les cellules extérieures peuvent être ouvertes
- Le choix doit être fait au hasard
- La même cellule ne peut être une entrée et une sortie en même temps.
- Les portes peuvent être indifféremment au nord, est sud ou ouest du labyrinthe.

Exemple après création



- Tester votre code avec le programme

```
if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=False)
    LIGNE = 4
    COLONNE = 9
    un_lab = Labyrinthe(LIGNE, COLONNE)
    un_lab.creer_labyrinthe(0, 0, LIGNE, COLONNE)
    un_lab.creer_portes(LIGNE, COLONNE)
    print(un_lab)
```