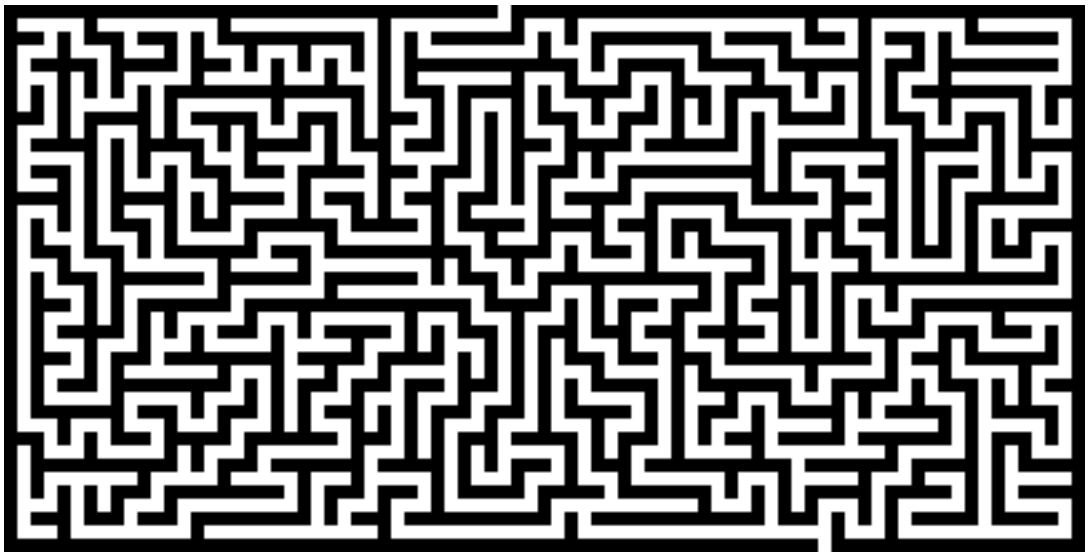
Notes de cours
Semaine 4

Cours Turing

1 Génération d'un labyrinthe en Python

Cette semaine, nous allons voir comment, à l'aide de Python, générer des images de labyrinthes aléatoires telles que celle ci-dessous.



Ce projet, plus pratique, vous permettra d'appliquer les concepts vus ces dernières semaines ainsi que de découvrir un module de manipulation d'images en Python. Dans un premier temps, nous allons nous intéresser au module `PIL` de Python qui nous permettra de créer et manipuler des images. Dans un second temps, nous aborderons la méthode de génération des labyrinthes.

1.1 Les images avec `PIL`

Le module `PIL`, de la bibliothèque *Pillow*, permet la création, la manipulation, la sauvegarde et l'affichage d'images *matricielles*. Pour le reste de cette section, nous allons découvrir les bases de Pillow. Cependant, le but de ce document n'est pas d'être exhaustif, mais simplement de fournir une introduction. Une documentation complète du module est disponible sur le site de la bibliothèque Pillow (en anglais).

Images matricielles Comme énoncé plus haut, Pillow permet la manipulation d'images matricielles. Les images matricielles sont des représentations d'images sous la forme d'une matrice de pixels. Une matrice est simplement un tableau, une grille, à deux dimensions. Les éléments de cette matrice sont ce qu'on appelle des pixels (pour *picture element*, littéralement *élément d'image* en anglais). Chaque pixel a donc une position dans l'image et indique la couleur à cet emplacement.

Code couleur Pour indiquer la couleur d'un pixel, on utilisera un *code couleur*. Les différents codes couleurs permettent d'identifier, de décrire, un ensemble de couleurs. Il en existe plusieurs, mais celui que nous allons utiliser est le code RGB (pour *red, green, blue*, soit *rouge, vert, bleu* en anglais). En RGB, on décrit une couleur en fonction de l'intensité de ses composantes de rouge, de vert et de bleu. Chaque composante a une valeur entre 0 et 255. Ainsi, chaque couleur représentable est associée à un triplet de valeurs. On aura par exemple $(0, 0, 0)$ pour le noir, et $(255, 255, 255)$ pour le blanc. Le rouge sera $(255, 0, 0)$, alors que le bleu $(0, 0, 255)$. Quant au code $(185, 237, 234)$, il correspond à un joli bleu ciel. Ce code couleur permet de représenter $256^3 = 16'777'216$ de couleurs différentes. Il existe de nombreux outils, en ligne ou à installer, qui permettent de sélectionner des couleurs et d'en obtenir le code RGB.

1.1.1 Import du module

Pour avoir accès à ce module `PIL` dans un programme Python, il faut l'*importer*. Pour importer un module, on utilise le mot-clé `import`. Dans notre cas précis, nous sommes uniquement intéressés par une unique définition : la *class* Image. Pour cela, on utilisera l'instruction suivante :

```
from PIL import Image
```

Les instructions d'import de modules sont généralement situées au tout début du fichier. Cela permet de facilement retrouver quels sont les modules utilisés par un programme et ses éventuelles dépendances à d'autres fichiers.

Installation de Pillow Si l'exécution de l'instruction donne lieu à une erreur de type `ModuleNotFoundError`, c'est que la bibliothèque *Pillow* n'est pas installée. En principe, la bibliothèque Pillow est déjà pré-installée dans l'environnement de programmation proposé et cette étape d'installation n'est pas nécessaire.

Pour installer la bibliothèque Pillow, il suffit de passer par le gestionnaire de paquets de Python, appelé `pip`. Pour cela, on exécutera la commande suivante dans un terminal :

```
pip install Pillow
```

Suivant votre installation Python, il se peut que l'utilitaire `pip` soit connu sous un autre nom, comme `pip3` par exemple. Dans ce cas, il vous faudra juste indiquer le bon nom à la place de `pip` dans la commande indiquée ci-dessus. Une fois la bibliothèque Pillow installée, le module `PIL` devrait être accessible depuis Python.

1.1.2 Création d'une image

Pour créer une image, on utilise la méthode `new` de la classe `Image`, comme suit :

```
largeur = 400  # Largeur en pixels
hauteur = 200  # Hauteur en pixels
couleur_fond = (181, 31, 31)  # Rouge groseille

# Création d'une image
mon_image = Image.new("RGB", (largeur, hauteur), couleur_fond)
```

La méthode new prend trois arguments :

1. Premièrement, un mode de couleur. Nous utiliserons ici toujours le mode RGB.
2. Deuxièmement, les dimensions de l'image, sous la forme d'un tuple de deux éléments (largeur et hauteur).
3. Finalement, et de façon optionnelle, le code d'une couleur (en mode RGB) pour le fond de l'image.

La méthode retourne comme valeur l'image nouvellement créée. Pour être précis, la valeur retournée est ce que l'on appelle une *instance* de la classe `Image`.

1.1.3 Ouverture d'un fichier

Il est aussi possible d'ouvrir une image depuis un fichier. Pour cela, on utilisera la méthode `open` de `Image`.

```
# Lecture d'une image
mon_image = Image.open("nom_du_fichier.ext")
```

Pour s'assurer que l'image soit bien en mode RGB, on peut suivre l'ouverture de l'image par un appel à la méthode `convert`, comme ceci :

```
# Création d'une image
mon_image = Image.open("nom_du_fichier.ext").convert("RGB")
```

En effet, certaines images, comme les images avec de la transparence ou certaines images en noir et blanc, n'utilisent pas le mode RGB pour les couleurs. Avec l'appel à `convert`, on s'assure que les couleurs seront bien représentées selon le mode RGB.

1.1.4 Affichage d'une image

On utilisera la méthode `show` pour afficher une image.

```
# Affichage d'une image
mon_image.show()
```

1.1.5 Sauvegarde d'une image

On utilisera la méthode `save` pour enregistrer une image.

```
# Sauvegarde d'une image dans un fichier
mon_image.save("mon_nom_de_fichier.png")
```

Notez qu'on utilisera généralement l'extension `png` pour sauvegarder nos images. Il est possible d'utiliser d'autres formats de fichiers, mais parfois ces formats utiliseront des techniques de compression de données avec pertes qui réduisent la qualité de l'image.

1.1.6 Dimensions d'une image

La largeur et la hauteur d'une image sont accessibles via les champs `width` et `height`. Il est aussi possible d'utiliser la méthode `size` afin d'obtenir un tuple de la largeur et de la hauteur.

```
print(mon_image.width)  # Affiche la largeur
print(mon_image.height)  # Affiche la hauteur
print(mon_image.size)  # Affiche la paire (largeur, hauteur)
```

1.1.7 Lecture d'un pixel

On utilisera la méthode `getpixel` afin d'obtenir la couleur d'un pixel de l'image. La méthode prend en unique argument la position du pixel dans l'image sous la forme d'une paire (x, y). La composante x indique la position sur l'axe horizontal, en nombre de pixels depuis la gauche. La composante y indique la position sur l'axe vertical, en nombre de pixels depuis le haut (et non le bas).

```
# Couleur du pixel en haut à gauche
couleur_pixel = mon_image.getpixel((0, 0))

 # Couleur du pixel en haut à droite
couleur_pixel = mon_image.getpixel((mon_image.width-1, 0))
```

La position du pixel en haut à gauche de l'image est $(0, 0)$. Comme l'on compte à partir de 0 sur les deux axes, la valeur maximale pour la position sur l'axe horizontal est la largeur de l'image *moins 1* et la valeur maximale sur l'axe vertical la hauteur *moins 1*.

Souvent, il est utile de décomposer une couleur en ses trois composantes. Pour cela, on utilise généralement en Python la syntaxe de déconstruction de tuple :

```
couleur_pixel = mon_image.getpixel((x, y))
rouge, vert. bleu = couleur_pixel

# Ou plus directement...
rouge, vert. bleu = mon_image.getpixel((x, y))
```

1.1.8 Écriture d'un pixel

La méthode `putpixel` permet de modifier la couleur d'un pixel de l'image. La méthode prend deux arguments : la position et la couleur.

```
couleur = (0, 0, 0)  # Noir
mon_image.putpixel((x, y), couleur)
```

Notez que la méthode modifie l'image et donc que les images telles qu'implémentées par le module `PIL` sont *mutables*.

1.1.9 Redimensionner une image

La méthode `resize` permet d'obtenir la copie d'une image aux dimensions données.

```
# Zoom par 5 dans une image
largeur = mon_image.width * 5
hauteur = mon_image.height * 5
ma_copie = mon_image.resize((largeur, hauteur), Image.BOX)
```

Notez que la méthode `resize` ne modifie pas l'image mais en retourne une copie. L'argument `Image.BOX` permet de faire en sorte de ne pas flouter l'image lors du redimensionnement. En effet, par défaut, lors du redimensionnement une méthode est utilisée pour éviter l'aspect net voire *pixelisé* de l'image agrandie. Or, dans notre cas, nos cherchons à préservez cette apparence très nette.

1.2 Génération d'un labyrinthe

Nous allons maintenant nous intéresser à la création d'un générateur de labyrinthes en Python. Le labyrinthe sera constitué de chemins blancs et de murs noirs, comme vous avez pu le voir en début de ce document.

Le but de ce mini-projet sera de créer un programme qui permettra à l'utilisateur de spécifier des dimensions et d'obtenir un labyrinthe de la taille appropriée sous la forme d'une image. Pour ce mini-projet, nous allons représenter les labyrinthes à l'aide d'une image de Pillow. Il sera ensuite facile soit d'afficher le résultat obtenu, soit de l'enregistrer dans un fichier.

Dans le but de simplifier votre implémentation, nous vous conseillons de séparer la partie interface avec l'utilisateur (demande des dimensions, affichage/sauvegarde de l'image) de la logique de création du labyrinthe. Pour cette dernière, nous vous conseillons d'implémenter une fonction nommée `creer_labyrinthe` qui prendra en entrée les dimensions du labyrinthe et qui retournera une image. L'image retournée contiendra le labyrinthe. Dans cette image, la largeur des murs et des couloirs sera de 1 pixel. Vous serez ensuite libre de redimensionner l'image avant de l'afficher ou de l'enregistrer.

```
def creer_labyrinthe(largeur, hauteur):
    pass
```

Cette fonction sera chargée, comme son nom l'indique, de créer le labyrinthe de toutes pièces. Le processus de création du labyrinthe sera découpé en quatre étapes. Lors de la première étape, nous allons créer le labyrinthe entièrement noir. Dans une deuxième étape, nous allons y établir des petites pièces séparées, entièrement entourées de murs. Dans un troisième temps, nous allons retirer des murs du labyrinthe, de façon aléatoire, afin de s'assurer que toutes les pièces du labyrinthe sont connectées entre elles. Finalement, nous ajouterons une entrée et une sortie au labyrinthe.

1.2.1 Création de l'image noire

Initialement, nous allons créer un labyrinthe entièrement composé de murs. Pour cela, il vous faudra créer une image noire aux dimensions données. Assurez-vous avant toute chose que les dimensions données pour le labyrinthe soient bien les deux impaires. En effet, comme nous allons alterner entre mur et couloir, et qu'il y a un mur au début et à la fin, il est important que la largeur et la hauteur soient impaires. Si ce n'est pas le cas, vous pouvez lancer une erreur avec l'instruction suivante :

```
raise ValueError("Dimensions incorrectes")
```

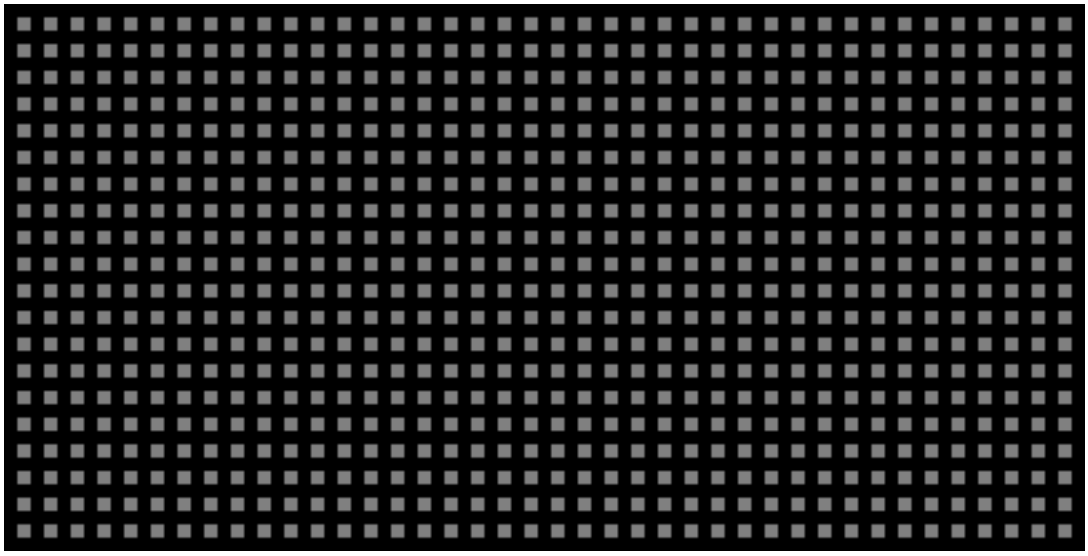
Dans le code de votre interface, vous pourrez gérer le cas où l'utilisateur entre des dimensions incorrectes.

À la fin de cette étape, le labyrinthe doit ressembler à ceci :

1.2.2 Création des pièces

Dans un second temps, nous allons peupler de pièces ce labyrinthe noir. Pour cela, nous allons colorer en gris tous les pixels de l'image à des positions (x, y) où x et y sont impaires. La couleur grise (par exemple $(128, 128, 128)$) indique que la pièce n'est pas encore connectée au reste du labyrinthe. La couleur sera ensuite changée à la prochaine étape.

À la fin de cette étape, le labyrinthe doit ressembler à ceci :



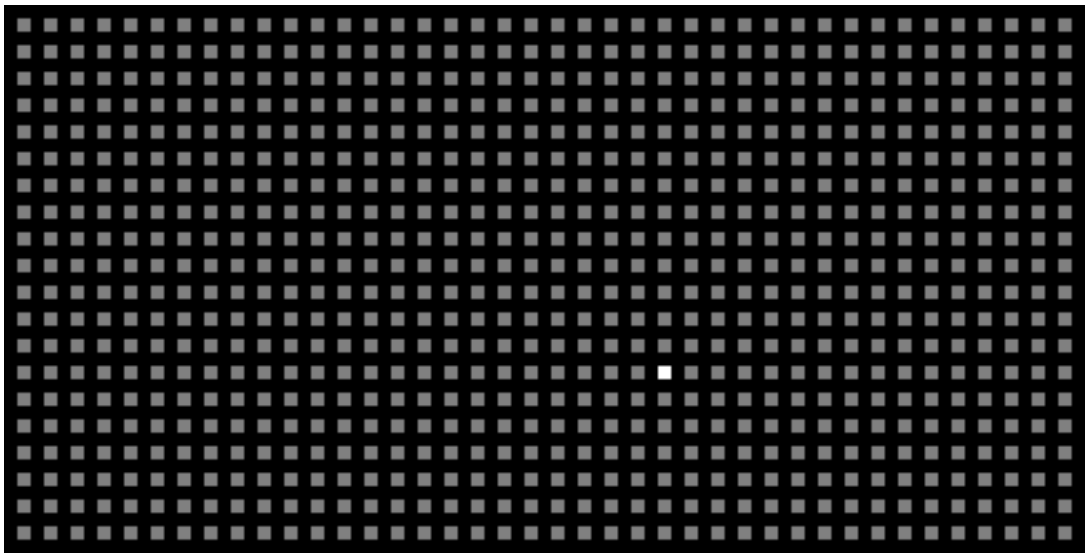
1.2.3 Connexion entre les pièces

Arrive ensuite la partie principale de la génération du labyrinthe : la connexion des différentes pièces. Pour cela, sélectionnez aléatoirement une des pièces du labyrinthe comme position de départ.

Choix de la position de départ Pour le choix de la pièce de départ, la fonction `randint` du module `random` pourra vous être utile. La fonction prend deux arguments, une borne inférieure et une borne supérieure (deux entiers) et retourne un nombre entier aléatoire entre ces deux bornes. Notez que les deux bornes sont inclusives.

Assurez-vous aussi de bien choisir une position de départ x et y où à la fois x et y sont impairs. Rappelez-vous que tous les nombres impairs peuvent être écrits sous la forme $2n + 1$, où n est un nombre entier.

Mémorisez cette position de départ et colorez-là en blanc dans l'image.

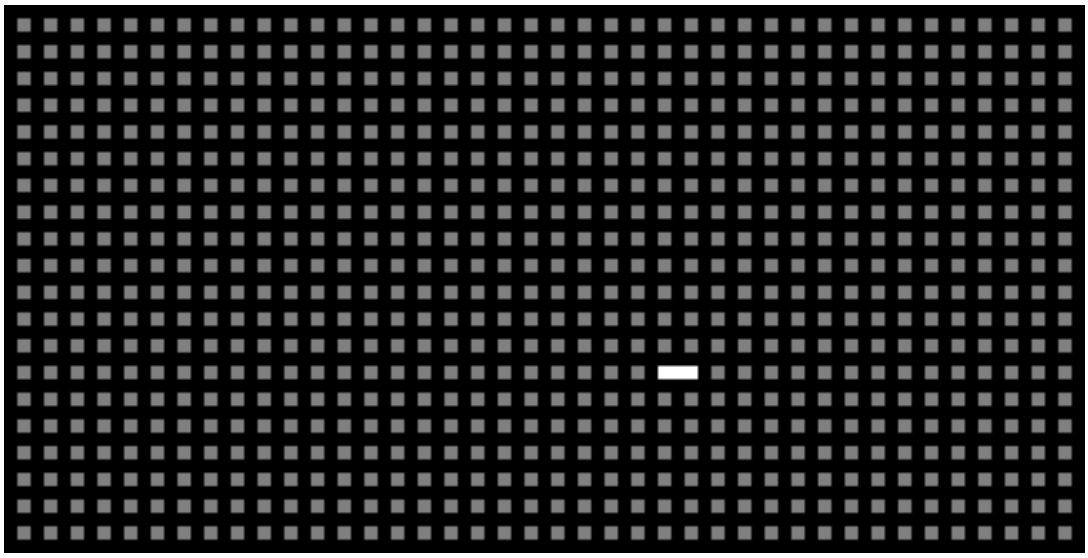


Déplacement à une pièce adjacente Ensuite, depuis la pièce courante, sélectionnez au hasard une pièce adjacente non-connectée au reste du labyrinthe (une pièce grise), puis faites sauter le mur entre les deux pièces (en coloriant le mur et la nouvelle pièce en blanc).

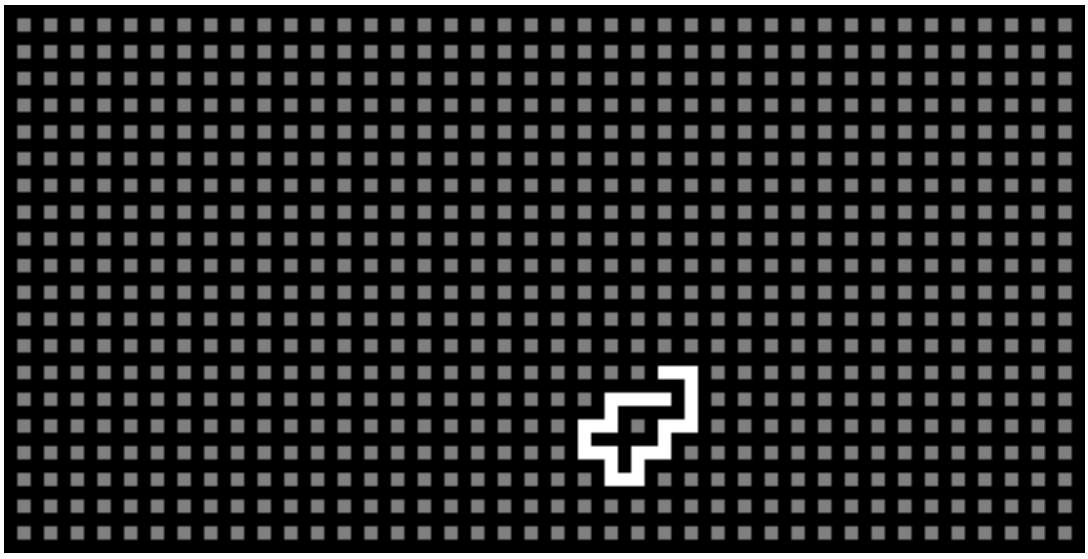
Pour cela, nous vous conseillons de concevoir une fonction Python qui prend en argument l'image et la position d'une pièce et qui retourne la position de toutes les pièces adjacentes de couleur grise. Attention à bien traiter le cas des positions en bordure du labyrinthe.

```
def pieces_adjacentes_non_connectees(image, x, y):
    pass
```

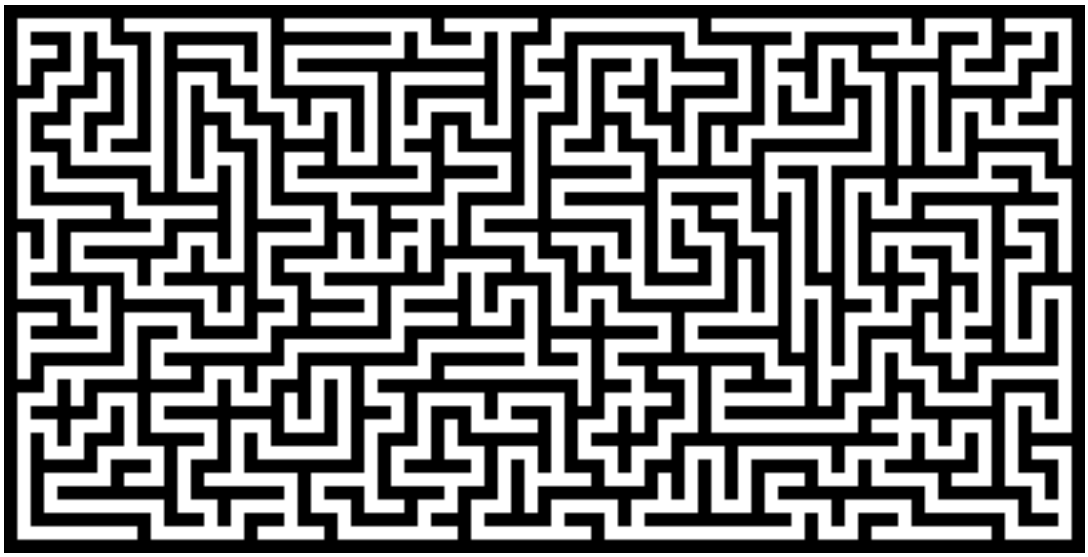
Une fois cette liste obtenue, la fonction `choice` du module `random` vous permettra de choisir une pièce adjacente au hasard pour la visiter. Le labyrinthe obtenu après un unique déplacement devrait ressembler à ceci :



Répétition Ensuite, on répète le processus de sélection d'une pièce adjacente non-connectée, cette fois depuis la pièce nouvellement connectée. On enchaîne les répétitions, choisissant à chaque fois une nouvelle pièce voisine depuis la dernière pièce connectée, et ce jusqu'à ce que l'on arrive à une pièce où toutes les pièces adjacentes sont connectées (un cul-de-sac), comme dans l'image ci-dessous.

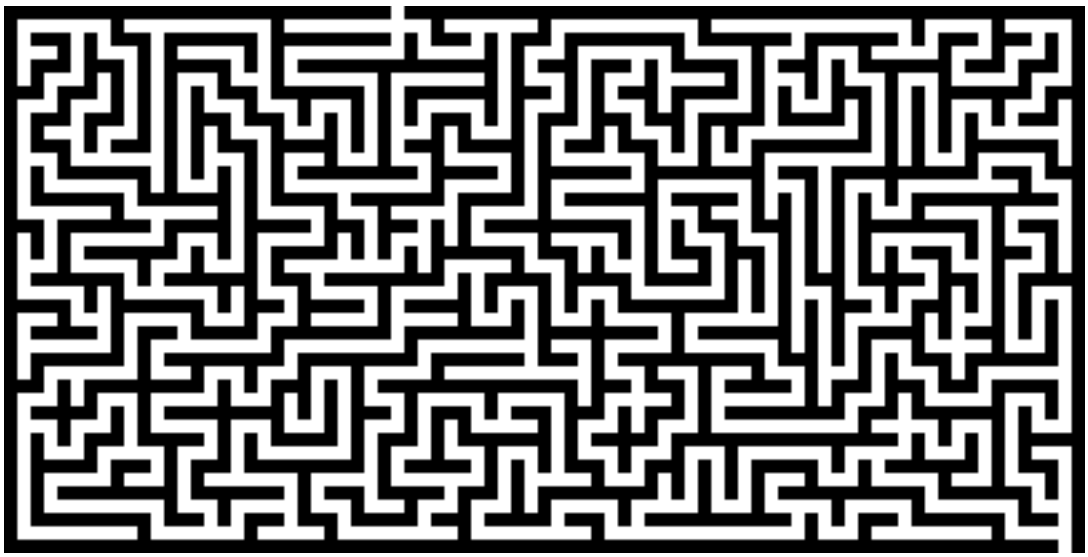


Rebrousser chemin Lorsque l'on arrive à un cul-de-sac, il faudra rebrousser chemin. Pour cela, il vous faudra maintenir un chemin depuis la position de départ, par exemple sous la forme d'une liste Python. Au départ, ce chemin sera vide, mais à chaque fois que l'on se déplace dans une nouvelle pièce, il faudra ajouter à la fin de ce chemin la position de la pièce que l'on quitte. Lorsque l'on arrive à un cul-de-sac, on se déplacera à la dernière pièce du chemin (la pièce d'où l'on était arrivé). Cette dernière sera alors retirée du chemin et l'on réessaiera le processus de sélection d'un voisin depuis cette pièce. La pièce aura certes déjà été visitée, mais elle aura peut-être encore des cellules adjacentes à visiter ! La génération du labyrinthe s'arrête lorsqu'il n'y a plus de pièces adjacentes non-connectées et que le chemin est vide. Le résultat à la fin de cette étape doit ressembler à ceci :



1.2.4 Ajouter une entrée et sortie

Finalement, on ajoutera une entrée sur le haut du labyrinthe ainsi qu'une sortie sur le bas du labyrinthe. La position sur l'axe horizontal est aléatoire mais doit permettre de connecter une pièce à l'extérieur du labyrinthe. Le résultat attendu est le suivant :

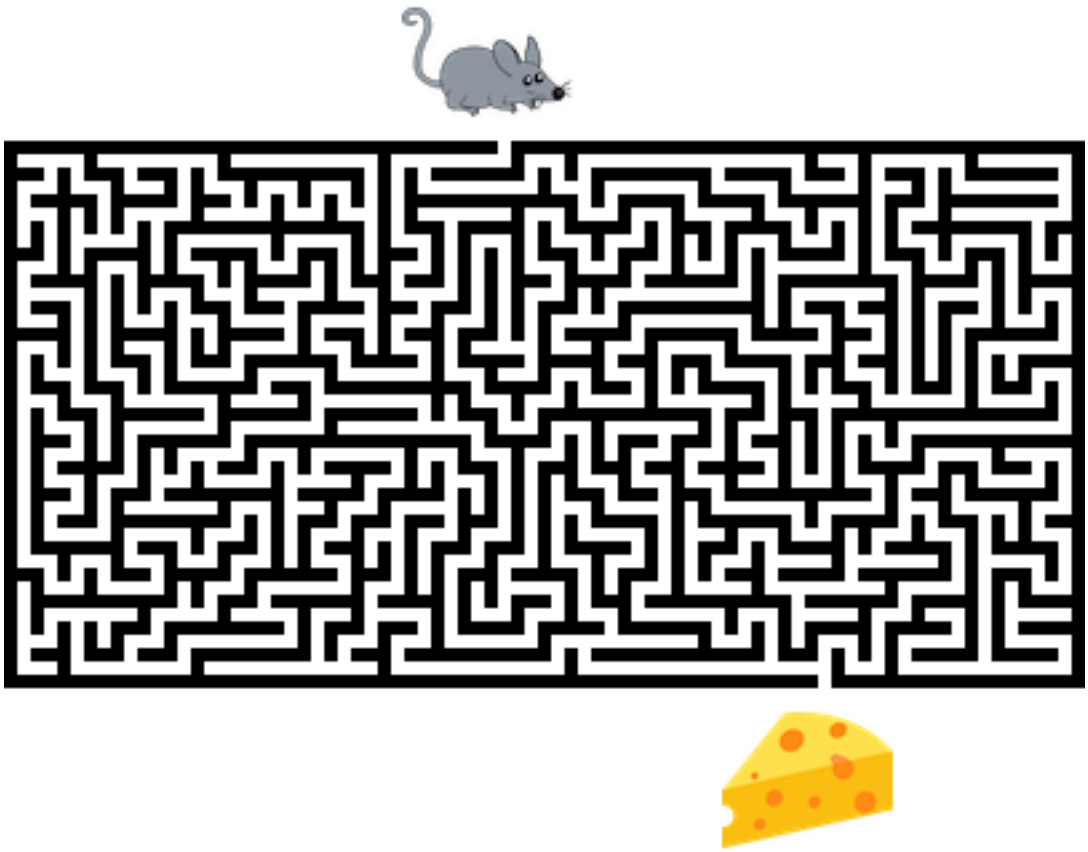


Une fois cette dernière étape implémentée, votre générateur de labyrinthe est terminé, félicitations !

1.3 Idées d'améliorations

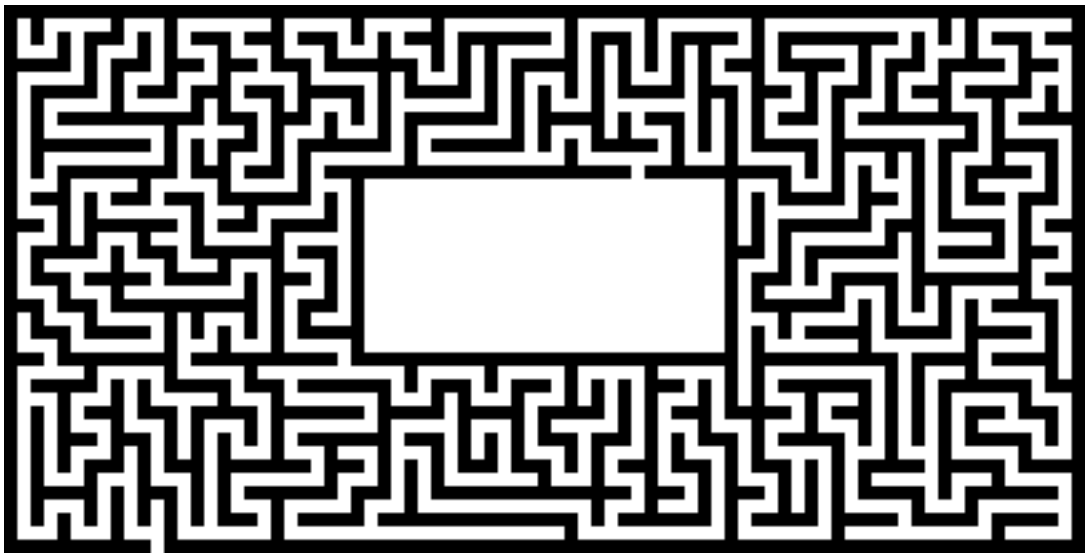
Si le coeur vous en dit, voici quelques idées d'améliorations pour votre générateur de labyrinthe :

Ajout d'images à l'entrée et à la sortie du labyrinthe Après avoir généré votre labyrinthe, vous pouvez y ajouter des éléments décoratifs à l'entrée et à la sortie. Par exemple, dans l'image ci-dessous, on a ajouté une souris à l'entrée et un fromage à la sortie.

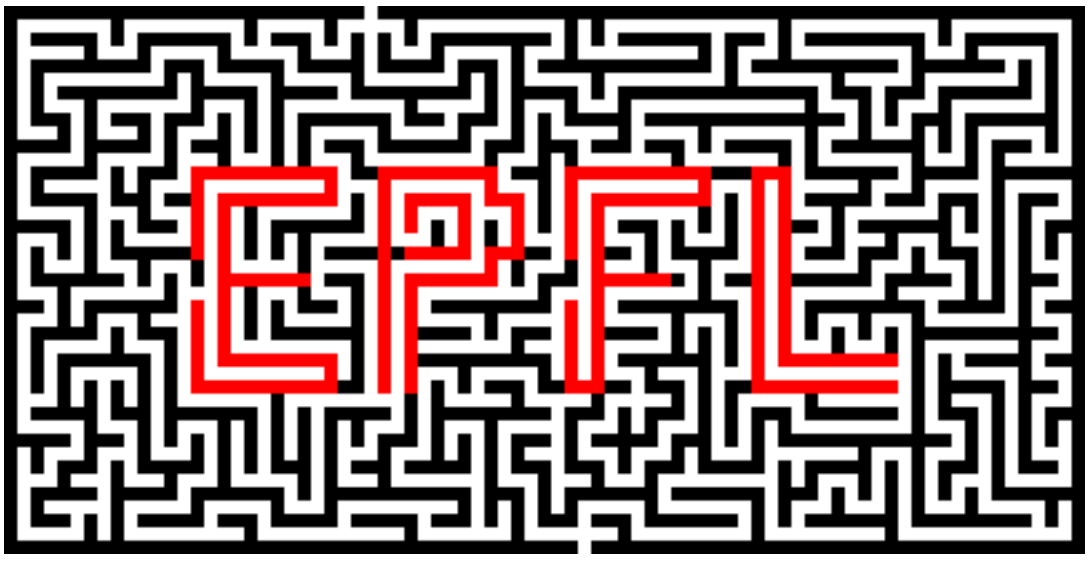


Notez que le module `PIL` dispose de fonctionnalités pour coller des images dans d'autres images. N'hésitez pas à consulter la documentation de la classe `Image`.

Zone vide et entrée au centre du labyrinthe Une variante pourrait être de faire commencer le labyrinthe en son centre, dans une plus grande zone vide, comme dans l'exemple ci-dessous.



Ajout de murs incassables De base, dans le labyrinthe, tous les murs internes sont susceptibles d'être supprimés. Une amélioration possible serait de pouvoir spécifier que certains murs sont incassables, en les colorant par exemple en rouge, comme dans l'exemple ci-dessous.



Reste le problème de savoir comment spécifier quels murs sont incassables !