Exercice 1. Création et copie d'une liste de listes

Rappelons qu'il est possible de créer une liste en utilisant l'opérateur de duplication « * ». Par exemple, l'expression `[1,2] * 4` s'évalue en `[1,2,1,2,1,2,1,2]`. Afin de créer une liste de listes de booléens, on propose le programme ci-contre.

```
L = [[False]*3]*5
L[0][1] = True
```

1. Demander à Python d'afficher `L` et expliquer pourquoi créer `L` de cette façon ne convient pas. Proposer une manière correcte de définir la liste `L`.
2. Expliquer pourquoi copier une liste de listes avec le programme ci-contre ne convient pas. Proposer une manière correcte de copier une liste de listes.

```
L = [[1,2,3], [4,7], [], [0,4]]
M = [L[i] for i in range(len(L))]
```

Exercice 2. Labyrinthe

Un labyrinthe est une grille rectangulaire avec $\texttt{n} \in \mathbb{N}^*$ lignes et $\texttt{m} \in \mathbb{N}^*$ colonnes où certaines cases sont des murs impossibles à franchir. Afin de simplifier ce qui suit, on suppose que toutes les cases sur les bords de la grille sont des murs. En Python, un labyrinthe est représenté par une variable `laby` contenant une liste de listes de booléens. Plus précisément, pour $\texttt{i} \in [\![0, \texttt{n}-1]\!]$ et $\texttt{j} \in [\![0, \texttt{m}-1]\!]$, le booléen `laby[i][j]` vaut `True` si la case d'indice `(i, j)` est un mur et `False` sinon. L'indice `i = 0` correspond à la ligne du haut, `i = n - 1` à la ligne du bas, `j = 0` à la colonne de gauche et `j = m - 1` à la colonne de droite.

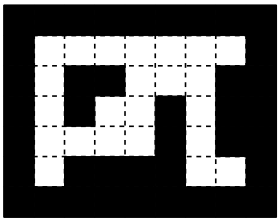

Figure 1

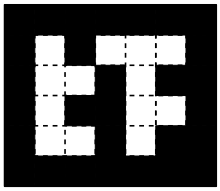

Figure 2

```
111111111
100000001
101100011
   ...
```

Figure 3

1. Définir une variable `laby1` représentant un labyrinthe avec 4 lignes et 7 colonnes où il n'y a pas de mur, mis à part sur les bords. Afin de traiter cette question rapidement, utilisez l'outil copier/coller de votre éditeur de texte avec les raccourcis clavier (`CTRL-C`/`CTRL-V`). Pour sélectionner le texte à copier, on pourra utiliser les touches du clavier en maintenant la touche `MAJ` enfoncée.
2. Définir deux variables `laby2` et `laby3` représentant les labyrinthes des figures 1 et 2. Afin de traiter cette question rapidement, merci de suivre les étapes suivantes :
   - → Commencer par dessiner le labyrinthe dans le fichier source en remplaçant les murs par des `1` et les cases libres par des `0`. Par exemple, pour le labyrinthe de la figure 1, on obtient la figure 3.
   - → Ouvrir la fonction "remplacer" de votre éditeur de texte. Par exemple sous Spyder : cliquer sur `Recherche` puis `Remplacer`.
   - → Remplacer `0` par « `False,` » (avec une virgule) et `1` par « `True,  ` » (avec une virgule et un espace).
   - → Ajouter les crochets `[` et `]` aux bons endroits pour obtenir la liste de listes. Pour vous déplacer dans le fichier, utilisez les flèches du clavier ainsi que les touches `début` et `fin`, vous serez plus rapides qu'en utilisant votre souris.
3. Écrire une fonction `nb_murs` qui prend en entrée la variable `laby` et renvoie le nombre de murs dans le labyrinthe.
4. Écrire une fonction `test_bords` qui prend en entrée la variable `laby` et renvoie `True` si tous les bords sont des murs ou `False` sinon. On rappelle que pour tester correctement la fonction, il faut nécessairement que certains tests renvoient `False`.

Dans la suite, on pourra supposer que la variable `laby` est telle que `test_bords(laby)` vaut `True`.
Afin d'afficher un labyrinthe dans la console, on va représenter un mur par `"# "` (un dièse suivi d'un espace) et une case sans mur par `"- "` (un tiret suivi d'un espace). On pourra utiliser la commande `print(s, end="")` qui affiche la chaîne de caractères `s` sans retour à la ligne.

5. Écrire une fonction `print_laby` qui prend en entrée la variable `laby` et affiche le labyrinthe dans la console. Lors des tests, le programme ne doit pas afficher `None`.

Exercice 3. Personnage dans le labyrinthe

On ajoute un personnage dans le labyrinthe qu'on affichera dans la console par `"O "`. Soit « `pos: (int, int)` » le couple d'entiers représentant la position du personnage dans le labyrinthe.

1. Écrire une fonction `print_laby_bis` qui prend en entrée `laby` ainsi que `pos`, et affiche le labyrinthe avec le personnage. On pourra supposer que le personnage ne se trouve pas sur un mur. Dans cette question, vous pouvez copier/coller le code de la fonction `print_laby` et le modifier.

Notre personnage peut se déplacer dans les quatre directions représentées par les chaînes de caractères `"haut"`, `"bas"`, `"gauche"` et `"droite"`. Lorsque le personnage essaie de se déplacer sur un mur, sa position n'est pas modifiée.

2. Écrire une fonction `nv_pos` qui prend en entrée `laby`, le couple `pos` ainsi qu'une chaîne de caractères `direction`, et renvoie la nouvelle position du personnage. On pourra supposer qu'initialement, le personnage ne se trouve pas sur un mur et que la variable `direction` vaut `"haut"`, `"bas"`, `"gauche"` ou `"droite"`. Lors des tests, vérifiez que votre personnage ne peut pas traverser les murs.

Lorsqu'on exécute la commande « `touche = input()` », Python attend que l'utilisateur saisisse un texte dans la console en terminant sa saisie par la touche `Entrée`. Son texte est stocké dans la variable « `touche: str` ».

3. Écrire une fonction qui prend en entrée `laby` ainsi qu'un couple `pos_ini` et déplace le personnage en fonction des saisies de l'utilisateur. Par exemple, on pourra utiliser la convention que la touche `z` représente un déplacement vers le haut, la touche `d` vers la droite, la touche `s` vers le bas, la touche `q` vers la gauche et que la touche espace met fin au programme. Testez votre fonction en vérifiant que votre personnage ne traverse pas les murs.

Exercice 4. Chemins dans le labyrinthe

Un "chemin" est une suite de cases dont aucune ne contient de mur et telles qu'il est possible de passer d'une case à la suivante en effectuant un pas vers le haut, vers le bas, vers la droite ou vers la gauche. En Python, on utilisera une variable « `chemin: list[int, int]` » contenant une liste de positions.

1. Écrire une fonction `pos_valide` qui prend en entrée `laby` ainsi qu'un couple noté `pos` et renvoie `True` si la position est valide (c'est à dire si elle est dans les bornes du labyrinthe et n'est pas un mur).
2. Écrire une fonction `pos_voisines` qui prend en entrée deux couples `pos1` et `pos2`, et renvoie `True` si les deux cases sont voisines. Dans le cas contraire, la fonction renverra `False`.
3. Écrire une fonction `chemin_valide` qui prend en entrée la variable `laby` ainsi que la variable `chemin`, et renvoie `True` lorsque le chemin est valide. Par exemple :

Labyrinthe	laby1	laby1	laby1	laby2	laby2	laby3
Chemins non valides	`[(2,-1)]`	`[(10,2)]`	`[(0,1), (1,1)]`	`[(1,1),(2,1), (2,2)]`	`[(1,1), (3,1)]`	`[(2,2), (2,3)]`

Labyrinthe	laby1	laby3
Chemins valides	`[(1,1),(1,2),(1,3), (1,4),(1,5),(2,5)]`	`[(2,2),(2,1),(3,1), (2,1),(1,1)]`

Exercice 5. Chemin le plus court

1. Écrire une fonction `trouver_chemin` qui prend en entrée la variable `laby` ainsi que deux couples `pos1`, `pos2`, et renvoie un chemin de taille minimum entre les deux positions. Votre fonction renverra `None` si aucun chemin n'existe. En particulier, dans le cas où l'une des deux positions est un mur, votre fonction devra renvoyer `None`. Si vous n'y arrivez pas, vous pouvez lire l'aide ci-dessous. Testez votre fonction avec :

laby	laby1	laby1	laby1	laby2	laby2	laby3	laby3
pos1	(0,0)	(1,1)	(1,1)	(5,1)	(-1,-1)	(1,1)	(1,1)
pos2	(1,1)	(1,1)	(2,5)	(5,7)	(1,1)	(4,2)	(4,4)

Indications (essayez de résoudre l'exercice sans lire ce qui suit). Dans un premier temps, supposons que `pos1` et `pos2` ne soient pas des murs et essayons de déterminer s'il existe un chemin entre ces deux positions. Pour cela, on construit une liste de listes d'entiers notée `dist` telle que `dist[i][j]` est la distance de la case (i, j) à la case `pos1`. En particulier, si `pos1` vaut `(i1, j1)` alors `dist[i][j]` vaut `0` si et seulement si `i = i1` et `j = j1`. Initialement, toutes les entrées de la variable `dist` contiennent `None` et on détermine successivement toutes les cases à distance `0`, puis `1`, puis `2`, et ainsi de suite jusqu'à ce que toutes les cases accessibles aient été visitées. Cette procédure est décomposée dans les questions qui suivent.

2. Soit `d` $\in \mathbb{N}$ un entier. On suppose que si la distance entre les positions `(i,j)` et `pos1` est inférieure à `d`, alors `dist[i][j]` a été calculée. Écrire une fonction `mise_a_jour` qui prend en entrée `laby`, `dist` ainsi que `d`, et calcule les valeurs `dist[i][j]` pour toutes les positions `(i,j)` à distance `d + 1` de `pos1`. En plus de modifier `dist`, votre fonction renverra `False` si aucune position n'est à distance `d + 1` de `pos1`, et `True` sinon.
3. Écrire une fonction `chemin_existe` qui prend en entrée la variable `laby` ainsi que deux couples `pos1`, `pos2`, et renvoie `True` s'il existe un chemin entre les deux positions et `False` sinon.

Dans le but d'écrire la fonction `trouver_chemin`, on va construire une liste de listes de couples notée `pred`. Étant donné un couple `(i,j)` et un chemin (noté `chemin`) de taille minimale entre `pos1` et `(i,j)` ; `pred[i][j]` contiendra la case qui précède `(i,j)` dans `chemin`. Notez en particulier que `len(chemin)` vaut `dist[i][j]+1`. Initialement toutes les cases de `chemin` contiennent `None`, puis ces cases sont remplies parallèlement à celles de la variable `dist`.

4. Supposons qu'il existe un chemin entre `pos1` et `pos2`, et que la variable `pred` ait été remplie. Écrire une fonction `pred_to_chemin` qui prend en entrée les variables `pos1`, `pos2`, `pred`, et renvoie un chemin entre les deux positions.
5. Soit `d` $\in \mathbb{N}$ un entier. On suppose que si la distance entre la position `(i,j)` et `pos1` est inférieure à `d`, alors `dist[i][j]` et `pred[i][j]` ont été calculées. Écrire une fonction `mise_a_jour_bis` qui prend en entrée `laby`, `dist`, `pred` ainsi que `d`, et calcule les valeurs `dist[i][j]` et `pred[i][j]` pour toutes les positions `(i,j)` à distance `d+1` de `pos1`. En plus de modifier `dist` et `pred`, votre fonction renverra `False` si aucune position n'est à distance `d+1` et `True` sinon. On pourra reprendre le code de la fonction `mise_a_jour` et le modifier.
6. Répondre à la question 1.

Un programme plus rapide Il est possible d'améliorer l'algorithme décrit dans la partie précédente. En effet, on peut se passer de la variable `dist` puisque la fonction `mise_a_jour` n'a besoin que de la liste des positions qui sont à distance `d` de `pos1`. On note donc `pos_dist_d` la liste des positions qui sont à distance `d` de `pos1`.

7. Soit `d` $\in \mathbb{N}$ un entier. On suppose que si la distance entre la position `(i,j)` et `pos1` est inférieure à `d`, alors `pred[i][j]` a déjà été calculée. Écrire une fonction `mise_a_jour_ter` qui prend en entrée `laby`, `pred` ainsi que `pos_dist_d`, et calcule les valeurs `pred[i][j]` pour toutes les case `(i,j)` à distance `d + 1` de `pos1`. En plus de modifier `pred`, votre fonction renverra la liste des positions qui sont à distance `d + 1` de `pos1`.
8. Écrire une fonction `trouver_chemin_bis` en utilisant `mise_a_jour_ter`.

Exercice 6. Dessin du labyrinthe

À l'aide du module `Tkinter` du TP 8, écrire une fonction `dessiner_laby` qui prend en entrée les variables `laby`, `chemin` ainsi qu'un entier $A \in \mathbb{N}^*$, et dessine le labyrinthe dans une fenêtre graphique. Chaque case du labyrinthe sera représentée par un carré de $A \times A$ pixels, les murs seront coloriés en noir, les cases vides en blanc et les cases du chemin en bleu. Dans le cas où le chemin n'est pas valide, votre fonction n'affichera rien.

Exercices à rendre au plus tard le 21/12/2025 à 20h

Comme d'habitude, vous devez créer vos propres tests et vous assurer que vos fonctions renvoient les bonnes valeurs sur ces tests. Les tests doivent être pertinents et suffisamment nombreux.

Exercice 7. Rock you

Contexte historique. Une des règles de base de la sécurité informatique est qu'un site web ne doit pas stocker directement les mots de passe de ses utilisateurs, il doit en stocker une version chiffrée. En décembre 2009, la société "RockYou" qui ne respectait pas cette règle a été victime d'une cyber-attaque. Plus de 10 000 000 de mots de passe ont été rendus publique. Le fichier `RockYou.txt` disponible sur la page du cours contient ces mots de passe :

https://informatique-lhp.fr/itc-mpsi.html

Téléchargez le fichier, et ouvrez le sous Python :

```
f = open("RockYou.txt", mode = 'r', encoding='utf8')
L_RockYou = [s.strip('\n') for s in f]
f.close()
```

Le programme ci-dessus crée une liste « `L_RockYou: list[str]` » contenant 14 344 392 chaînes de caractères. Les fonctions écrites dans cet exercice seront testées sur `L_RockYou`, mais doivent fonctionner pour une liste « `L: list[str]` » quelconque.

Remarque. Si vous utilisez un mot de passe présent dans `RockYou.txt`, vous devriez le changer.

1. Écrire une fonction « `len_maxi(L: list[str]) -> int` » qui renvoie la taille maximale parmi les éléments de `L`. Si `L` est vide, votre fonction déclenchera une erreur. Par exemple :

```
L = ["j@t", "love!", "?", "rj*"]
print(len_maxi(L))          # Affiche 5 qui est la taille de "love!"
print(len_maxi(L_RockYou)) # Affiche 285
```

2. Écrire une fonction « `makeM(L: list[str]) -> list[list[str]]` » qui renvoie une liste `M` telle que `M[n]` contient tous les éléments de `L` de taille `n`. Par exemple :

```
L = ["j@t", "love!", "?", "rj*"]
print(makeM(L)) # Affiche [[], ['?'], [], ['j@t', 'rj*'], [], ['love!']]
M_RockYou = makeM(L_RockYou)
print(M_RockYou[3][1444]) # Affiche "j@t"
print(M_RockYou[5][1678]) # Affiche "love!"
print(M_RockYou[1][38])   # Affiche "?"
print(M_RockYou[3][997])  # Affiche "rj*"
```

3. On considère qu'un mot de passe est "fort" lorsqu'il contient au moins 8 caractères, au moins trois lettres minuscules, au moins deux lettres majuscules et au moins un chiffre. À l'aide de la fonction `ord`, écrire une fonction « `est_fort(s: str) -> bool` » qui renvoie `True` si `s` est un mot de passe fort et `False` sinon.
4. Écrire une fonction « `nb_forts(M: list[list[str]]) -> list[int]` » qui prend en entrée la liste `M` construite à la question 2 et renvoie une liste « `N: list[int]` » telle que `N[n]` est le nombre de mots de passe forts dans `M[n]`.

```
L = ["j@t", "love!", "?", "rj*"]
print(nb_forts(makeM(L)))  # Affiche [0, 0, 0, 0, 0, 0]
M_RockYou = makeM(L_RockYou)
print(nb_forts(M_RockYou)) # Affiche [0, 0, 0, 0, 0, 0, 0, 0, 17172, 14356, ...
```

Exercice 8. Percolation (exercice facultatif)

Dans cet exercice on s'intéresse à des labyrinthes carrés avec n lignes et n colonnes. Soit $p \in [0, 1]$ une probabilité. Le labyrinthe est généré aléatoirement de la manière suivante : chaque case est un mur avec probabilité p et n'est pas un mur avec probabilité $1 - p$. Notre but est d'estimer la probabilité $P(p)$ qu'il existe un chemin entre une case située en haut du labyrinthe et une case située en bas du labyrinthe.

1. À l'aide du module `random`, écrire une fonction `rand_laby` qui prend en entrée n et p, et renvoie un labyrinthe généré aléatoirement comme décrit ci-dessus.
2. À l'aide du module `Tkinter`, écrire une fonction `dessiner_accessibles` qui prend en argument `laby` et affiche le labyrinthe en coloriant en bleu toutes les cases accessibles depuis une case située en haut. On pourra s'inspirer des exercices 5 et 6.
3. Écrire une fonction `approx_P` qui prend en entrée n et p, et donne une estimation de $P(p)$.

En réalité, il existe une probabilité p_0 dite ***critique*** telle que :

→ Si $p < p_0$, alors $P(p)$ tend vers 1 lorsque n tend vers l'infini.

→ Si $p > p_0$, alors $P(p)$ tend vers 0 lorsque n tend vers l'infini.

4. Vérifiez numériquement que p_0 vaut environ 0.4. Pour cela, on pourra exécuter la fonction de la question précédente avec différentes valeurs de p.