INFO-F-106 : PROJET D'INFORMATIQUE
PROJET 1 – DÉMINEUR

Anthony Cnudde Gwenaël Joret Abel Laval Tom Lenaerts
Robin Petit Cédric Simar

version du 4 novembre 2022

Présentation générale

Le projet en trois phrases

L'objectif du projet est de réaliser une implémentation en Python 3 du *Démineur*. C'est un jeu à 1 joueur sur une grille de taille variable et dont certaines cases contiennent des *mines*. Le but du jeu est de dévoiler petit-à-petit les cases vides de la grille jusqu'à localiser toutes les mines et les "désamorcer". Le joueur perd s'il dévoile une case contenant une mine.

Le Démineur

Au début du jeu, toutes les cases du plateau sont inexplorées. Le joueur doit alors dévoiler des cases une par une en faisant attention de ne pas dévoiler de mine. Sur chaque case déjà explorée est affiché un chiffre entre 1 et 8 : c'est le nombre de mines présentes dans le voisinage immédiat de cette case (s'il n'y a aucune mine dans ce voisinage, on peut soit afficher 0 soit laisser la case vide).
C'est la version disponible par défaut sur les systèmes d'exploitation Windows qui a donné sa notoriété au jeu (voir figures 1 et 2). Pour ce projet, nous nous contenterons cependant de coder le jeu dans un terminal, c'est-à-dire sans interface graphique.

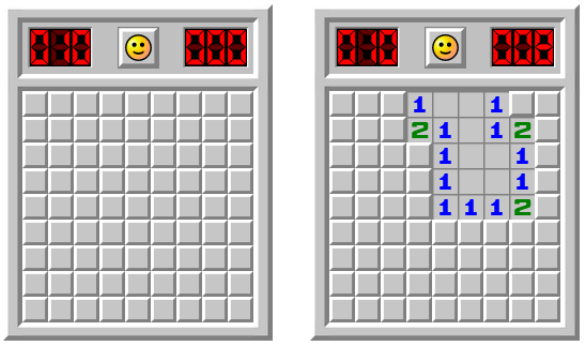


FIGURE 1 – À gauche, le plateau de jeu au début de la partie. À droite, ce même plateau après que le joueur ait cliqué sur une première case au hasard.

Le but du jeu est de localiser toutes les mines présentes sur le plateau (il y en a 10 sur la figure 2). Afin de marquer la position d'une mine, le joueur peut placer un drapeau sur la case correspondante.
La partie est gagnée lorsqu'un drapeau est posé sur chaque case contenant une mine (et qu'aucun drapeau n'est placé sur une case sans mine). Au contraire, on perd si l'on dévoile une case contenant une mine, au lieu d'y placer un drapeau.

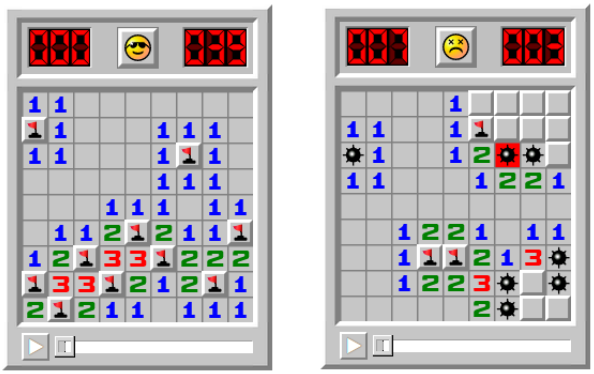


FIGURE 2 – À gauche, une partie gagnée : les 10 mines sont localisées par des drapeaux. À droite, une partie perdue : trois mines ont bien été localisées mais le joueur a dévoilé une case qui contenait une mine.

Consignes générales

- L'ensemble du projet est à réaliser en `Python 3`.
- Le module INFO-F106 est constitué de deux projets de programmation distincts, un par quadrimestre.
- En plus de ces projets à remettre, chaque étudiant(e) devra remettre un rapport final au second quadrimestre (Q2).
- La répartition des points est la suivante :
  - Projet 1 (Q1, Démineur) : 40 points
  - Projet 2 (Q2) : 40 points
  - Rapport (Q2) : 20 points
  - **Total : 100 points**
- Les deux projets et le rapport devront être remis sur GitHub Classroom.

— Les « Consignes de réalisation des projets » (cf. `http://www.ulb.ac.be/di/consignes_projets_INFO1.pdf`) sont d'application pour ce projet individuel. *(Exception : Ne tenez pas compte des consignes de soumission des fichiers, des consignes précises pour la soumission via GitHub Classroom seront données).* Vous lirez ces consignes en considérant chaque partie de ce projet d'année comme un projet à part entière. Relisez-les régulièrement !
— **Il n'y aura pas de seconde session pour ces projets !**

Veuillez noter également que le projet vaudra **zéro** sans exception si :
— le projet ne peut être exécuté correctement via les commandes décrites dans l'énoncé ;
— les noms de fonctions (et de vos scripts) sont différents de ceux décrits dans cet énoncé, ou ont des paramètres différents ;
— à l'aide d'outils automatiques spécialisés, nous avons détecté un plagiat manifeste (entre les projets de plusieurs étudiant(e)s, ou avec des éléments trouvés sur Internet). Nous insistons sur ce dernier point car l'expérience montre que chaque année une poignée d'étudiant(e)s pensent qu'un petit copier-coller d'une fonction, suivi d'une réorganisation du code et de quelques renommages de variables passera inaperçu... Ceci sera sanctionné d'une note nulle pour l'entièreté du projet pour toutes les personnes impliquées, ainsi que d'éventuelles autres sanctions. Afin d'éviter cette situation, veillez en particulier à ne pas partager de bouts de codes sur des forums, Facebook, etc.

Soumission de fichiers

La soumission des fichiers se fait via un repository individuel par projet sur la plateforme GitHub Classroom. Pour chaque projet, le lien GitHub Classroom à utiliser est donné dans l'énoncé (voir consignes de remise).

Une remarque concernant les retards : Contrairement à d'autres cours d'informatique, il n'est ici pas possible de remettre un projet en retard. Le système de versioning offert par Git vous permet de constamment mettre à jour la version de votre code sur le serveur de GitHub Classroom. Vous êtes d'ailleurs **fortement encouragés** à le faire régulièrement lorsque vous travaillez sur une partie. Ceci vous permet de garder une copie de chaque version intermédiaire de votre travail, et nous permet à nous, en tant qu'enseignants, d'avoir une idée de la régularité de votre travail. Pour l'évaluation d'une partie, vous ne devez pas indiquer quelle est la version "finale" de votre code, nous prendrons simplement la dernière version uploadée sur le repository avant la date et heure limite (toute version uploadée après sera ignorée).

Communication avec l'équipe des enseignants

Remarque préliminaire : Il y a 614 étudiant(e)s inscrits à ce cours (chiffres du 28 octobre 2022), il nous est donc simplement impossible de répondre individuellement aux questions par email. Ceci nous amène à la règle suivante :

Règle d'or : ne jamais envoyer d'email !

Des séances de questions / réponses seront organisées régulièrement, vous aurez donc de nombreuses occasions pour poser vos questions. De cette façon, tout le monde en profite ! Et cela nous évite de répondre 25x à la même question.

Afin d'insister sur l'importance de cette règle, voici quelques exemples de situations dans lesquelles un(e) étudiant(e) souhaiterait contacter un assistant ou le titulaire. Dans chacune de

ces situations, écrire un email (ou message individuel Teams) est proscrit :

- « j'ai une question concernant l'énoncé du projet » → question à poser lors des séances Q/R organisées par les assistants ;
- « j'éprouve des difficultés à utiliser git » → allez aux guidances en informatique, ils pourront vous aider ;
- « je ne comprends pas la note reçue pour mon projet, j'aimerais pouvoir en discuter » → allez à la séance de visite des copies organisée par les assistants en charge ;
- « j'ai eu un problème technique avec git juste avant la remise, je n'ai pas réussi à uploader le fichier à temps sur GitHub Classroom, le voici en pièce jointe » → n'envoyez jamais de fichiers par email ; nous vous demandons d'uploader **très régulièrement** votre code sur GitHub Classroom, la dernière version présente sur le serveur sera donc corrigée (si vous n'avez pas uploadé de versions intermédiaires, vous aurez donc une note nulle) ;
- « j'ai eu zéro à mon projet pour non-respect des consignes à cause d'une faute de frappe dans le nom de mon fichier, je trouve cela trop sévère » → nous appliquons les consignes annoncées dans l'énoncé à la lettre, sans aucune exception. Apprendre à respecter les consignes est un des objectifs pédagogiques de ce projet d'année.

Notez que nous ne répondrons simplement pas aux emails concernant le projet d'année si jamais vous nous en envoyez. Deux exceptions à cette règle : (1) emails d'ordre administratif adressés aux titulaires (certificat maladie, réorientation, EBS, etc.), et (2) email aux titulaires si jamais votre nom n'apparaît pas dans liste des étudiant(e)s inscrit(e)s sur Github Classroom.

Objectifs pédagogiques

Ce projet *transdisciplinaire* permettra de solliciter diverses compétences.

- *Des connaissances vues aux cours de programmation, langages, algorithmique ou mathématiques.* L'ampleur du projet requerra une analyse plus stricte et poussée que celle nécessaire à l'écriture d'un projet d'une page, ainsi qu'une utilisation rigoureuse des différents concepts vus aux cours.
- *Des connaissances non vues aux cours.* Les étudiant(e)s seront invité(e)s à les étudier par eux-mêmes, aiguillé(e)s par les *tuyaux* fournis par l'équipe encadrant le cours. Il s'agit entre autres d'une connaissance de base des interfaces graphiques en `Python 3`.
- *Des compétences de communication.* Après la partie 4, les étudiant(e)s remettront un rapport expliquant leur analyse, les difficultés rencontrées et les solutions proposées. Le rapport devra être rédigé en LaTeX. Ce sera l'occasion pour les étudiant(e)s de se familiariser avec ce langage, utilisé pour la rédaction de documents scientifiques. Une orthographe correcte sera exigée.

En résumé, vous devrez démontrer que vous êtes capables d'appliquer des concepts vus aux cours, de découvrir par vous-même des nouvelles matières, et enfin de communiquer de façon scientifique le résultat de votre travail.

Bon travail !

1 Démineur

Il vous est demandé d'implémenter le jeu du Démineur sur terminal, en donnant au joueur la possibilité de faire varier la taille du plateau et le nombre de mines présentes sur celui-ci.
Le jeu doit être lancé via la commande

```
python3 demineur.py n m number_of_mines
```

où `n`, `m` et `number_of_mines` devront être remplacés par des entiers, lequels sont les paramètres du jeu définissant la taille du plateau et le nombre de mines présentes sur celui-ci. Par exemple, pour jouer sur un plateau (standard) de 8x8 contenant 10 mines, il faudra entrer :

```
python3 demineur.py 8 8 10
```

Afin d'utiliser des paramètres entrés en ligne de commande dans votre code, vous pouvez utiliser la méthode `argv` du module `sys`. Avec l'exemple ci-dessus, la méthode `sys.argv` renvoit une liste de chaînes de caractères : `['demineur.py', '8', '8', '10']`. Le premier élément de la liste correspond au nom du script python, tandis que `sys.argv[1]`, `sys.argv[2]` et `sys.argv[3]` correspondent aux paramètres (Attention : ici les paramètres de `sys.argv` sont des chaînes de caractères, pas des entiers !).
Vous n'avez alors qu'à récupérer ces paramètres dans votre fonction `main` afin de les utiliser en jeu.

Veillez à ce que votre fichier puisse être importé, mettez donc l'appel à la fonction `main` dans un test conditionnel `if __name__ == '__main__':`

1.1 Structure générale du programme

Pour ce projet, on utilisera deux versions du plateau de jeu :

— Le *plateau de référence* nommé `reference_board` dans le code :
Ce plateau contient la position des mines et le nombre de mines au voisinage de chaque case. Il est généré au début du jeu et n'est plus modifié par la suite. Il n'est pas connu du joueur et servira comme argument pour plusieurs fonctions qu'il faudra implémenter.

— Le *plateau de jeu* nommé `game_board` dans le code :
C'est le plateau que voit le joueur pendant la partie. Il ne contient que des cases non-explorées au début de la partie et sera modifié petit-à-petit au cours de celle-ci.

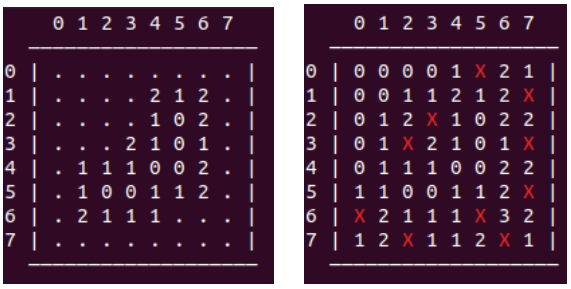


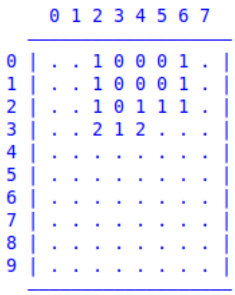
FIGURE 3 – À gauche, le plateau de jeu au début de la partie, après propagation du premier clic. À droite, le plateau de référence. Notez que l'affichage sur terminal permet de colorer les mines pour les rendre plus faciles à distinguer.

FIGURE 4 – La plateau de jeu après plusieurs tours.

1.2 Affichage des plateaux

Le plateau de jeu est affiché sous forme d'une matrice de n lignes et m colonnes, avec n et m supérieurs ou égaux à 4. En pratique, n et m seront souvent égaux mais il doit être possible de créer des plateaux rectangulaires. Le mode *facile* du démineur utilise typiquement une grille de taille 8×8 et contenant 10 mines.

Voici un exemple d'affichage à l'écran d'un plateau de taille 10×8 après le premier clic du joueur.



On peut ici noter que la numérotation commence à 0 et termine à $n - 1$ et $m - 1$, respectivement. Cela permet de simplifier l'écriture et la lecture du code puisque alors, l'élément

`game_board[i][j]` correspond bien à la case (i, j), et non à la case $(i + 1, j + 1)$, comme ce serait le cas si l'on décidait de faire démarrer la numérotation des cases au chiffre 1. Notez aussi qu'avec cette convention, la première variable correspond à l'axe des ordonnées et la seconde à l'axe des abscisses.
Par exemple, dans l'image ci-dessus, la case tout en haut à droite est la case $(0, 7)$, et non $(7, 0)$.

Les cases non-explorées sont représentées par le caractère `.` tandis que les cases avec un chiffre correspondent aux cases sans mines.

Affichage d'un plateau pour n ou m supérieur à 10

Lorsque le nombre de lignes ou de colonnes d'un plateau dépasse 10, l'affichage du plateau de jeu est modifié par la présence d'indices à deux chiffres. Il faut alors modifier la fonction d'affichage pour tenir compte de ce problème. (NB : nous ferons l'hypothèse que $n \leq 100$ et $m \leq 100$, il n'y aura donc pas d'indice à trois chiffres).

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13
  ┌──────────────────────────┐
0 | . . . . . . . . . . . . . |
1 | . . . . . . . . . . . . . |
2 | . 2 2 1 2 . . . . . . . . |
3 | . 1 0 0 1 2 2 1 1 . . . . |
4 | . 2 1 1 0 0 0 0 2 . . . . |
5 | . . . 2 1 0 0 0 1 . . . . |
6 | . . . . 1 0 0 0 1 1 2 . . |
7 | . . . . 1 0 0 0 0 0 1 . . |
8 | . . . . 1 1 1 0 0 0 2 . . |
9 | . . . . . . 1 0 0 0 1 . . |
10| . . . . . . 2 0 0 0 1 3 . |
11| . . . . . . 1 0 0 0 0 1 . |
12| . . . . 2 1 1 0 0 0 0 1 1 |
13| . . . . 1 0 0 0 0 0 0 0 0 |
  └──────────────────────────┘
```

```
                  1 1 1 1
  0 1 2 3 4 5 6 7 8 9 0 1 2 3
  ┌──────────────────────────┐
0 | . . . . . . . . . . . . . |
1 | . . . . . . . . . . . . . |
2 | . 2 2 1 2 . . . . . . . . |
3 | . 1 0 0 1 2 2 1 1 . . . . |
4 | . 2 1 1 0 0 0 0 2 . . . . |
5 | . . . 2 1 0 0 0 1 . . . . |
6 | . . . . 1 0 0 0 1 1 2 . . |
7 | . . . . 1 0 0 0 0 0 1 . . |
8 | . . . . 1 1 1 0 0 0 2 . . |
9 | . . . . . . 1 0 0 0 1 . . |
10| . . . . . . 2 0 0 0 1 3 . |
11| . . . . . . 1 0 0 0 0 1 . |
12| . . . . 2 1 1 0 0 0 0 1 1 |
13| . . . . 1 0 0 0 0 0 0 0 0 |
  └──────────────────────────┘
```

FIGURE 5 – À gauche, une version de `print_board` où les problème d'affichage des indices n'ont pas été anticipés. À droite, une version modifiée de la fonction qui résout le problème d'affichage.

Ici, la solution choisie consiste à ajouter un espace devant l'indice des lignes lorsque celui-ci est inférieur à 10. En effet, les chaînes de caractères `␣0`, `␣9` ou `10` ont alors la même longueur ce qui corrige le décalage qui apparaissait en bas du plateau.
Pour l'affichage des indices des colonnes, on affiche l'indice sur deux lignes : la première contient le chiffre des dizaines et la seconde celui des unités. Lorsque le chiffre des dizaines vaut 0, on remplace ce 0 par un espace `␣`. Dans l'exemple ci-dessus, les deux chaînes de caractères sont construites comme suit :

```
␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ 1 ␣ 1 ␣ 1 ␣ 1
0 ␣ 1 ␣ 2 ␣ 3 ␣ 4 ␣ 5 ␣ 6 ␣ 7 ␣ 8 ␣ 9 ␣ 0 ␣ 1 ␣ 2 ␣ 3
```

En plus de cela, il est évidemment nécessaire d'ajouter un certain nombre d'espaces afin d'aligner ces deux lignes avec le reste du plateau.

1.3 Encodage des plateaux

Un plateau (que ce soit le plateau de jeu ou celui de référence) est encodé sous forme d'une matrice `board` de taille $n \times m$, c'est-à-dire une liste de n listes, chacune composée de m caractères, c'est-à-dire un par case : `.` pour une case non-explorée, `0`, ..., `8` pour les cases sans mines, `F` pour les cases avec un drapeau et enfin `X` pour celles avec une mine.

1.4 Format des mouvements

Lors d'une partie de démineur, deux actions sont possibles à chaque moment : cliquer sur une case pour la dévoiler (en espérant qu'elle ne contienne pas de mine) ou placer un drapeau sur une case que l'on pense être occupée par une mine. Notez que les drapeaux ne peuvent être placés que sur les cases encore inexplorées.
Les coups sont alors encodés de la manière suivante :

— Pour dévoiler une case non explorée, le format est "`c i j`" pour dévoiler la case (i, j). Par exemple, pour dévoiler la case $(1, 3)$, on tapera simplement "`c 1 3`".
— Pour placer un drapeau, le format est "`f i j`". Par exemple, pour placer un drapeau en $(6, 0)$, on tapera "`f 6 0`".

Ainsi, lorsque le joueur choisit un coup encodé par une chaîne de caractères, cette dernière contient deux informations : d'abord la nature du coup ("c" pour cliquer sur la case ; "f" pour placer un drapeau (*"flag"*)), puis la case sur laquelle effectuer l'action.

```
Choix d'une case :f 0 4

    0 1 2 3 4 5 6 7
  ┌─────────────────┐
0 | 0 0 0 1 F . . . |
1 | 0 0 0 1 3 . . . |
2 | 0 0 0 0 2 . . . |
3 | 0 0 1 1 2 . . . |
4 | 0 0 1 . . . . . |
5 | 0 0 1 . . . . . |
6 | 0 0 1 . . . . . |
7 | 0 0 1 . . . . . |
  └─────────────────┘
```

FIGURE 6 – Ici, le joueur vient de placer un drapeau sur la case $(0, 4)$

1.5 Conditions de victoire

Après chaque action du joueur, le programme doit vérifier les conditions de victoire et mettre fin au jeu lorsque au moins l'une de celle-ci est remplie. Ces conditions sont les suivantes :

1. Le joueur gagne si toutes les cases contenant une mine – et uniquement ces cases – sont marquées d'un drapeau.
2. Le joueur gagne s'il reste exactement autant de cases non-dévoilées sur le plateau qu'il n'y a de mines.

Autrement dit, pour gagner il faut soit que chaque drapeau placé sur le plateau de jeu corresponde exactement à une mine sur le plateau de référence, [1] ou alors que les dernières cases

1. En particulier, placer un drapeau sur chaque case du plateau de jeu ne doit pas valider la condition de victoire.

encore non-dévoilées soient exactement celles qui contiennent des mines.
Notez qu'une case avec un drapeau est comptée comme non-dévoilée (la présence d'un drapeau sur une case signifie que le joueur *pense* qu'il y a une mine à cet endroit, ça ne dit pas que la case contient *effectivement* une mine.).
Notez aussi que dans la plupart des situations, les deux conditions de victoire seront réunies en même temps.

1.6 Propagation des clics

Lorsque le joueur choisit de dévoiler une case du plateau de jeu, elle est dévoilée. Le joueur voit alors combien de mines sont présentes dans le voisinage immédiat de cette case.
Dans l'éventualité où la case choisie n'a aucune mine dans son voisinage immédiat (c'est-à-dire que dans le plateau de référence, cette case contient un `0`), alors non seulement on doit la dévoiler, mais on doit aussi dévoiler son voisinage.
Ce processus est récursif : si les nouvelles cases dévoilées sont aussi des `0`, on continue à dévoiler leur voisinage. La récursion contient trois étapes :

- On dévoile la case courante, laquelle contient un `0`.
- Pour chaque case du voisinage encore non dévoilée et qui contient un `0`, on applique la récursion (retour à la première étape).
- Si aucune case non-dévoilée du voisinage n'est un `0`, on dévoile le voisinage de la case courante.

1.7 Gestion de l'aléatoire

Certaines fonctions du programme nécessiteront de faire appel à des processus pseudo-aléatoires (le placement des mines sur la grille, par exemple).
Le module `random` de Python fournit les outils basiques dont vous aurez besoin pour ce projet. Il contient en particulier deux méthodes importantes :

— La méthode `random.randint` prend en entrée deux entiers `a` et `b` avec $a \leq b$ et renvoie un entier aléatoire dans l'intervalle $[a, b]$.
— La méthode `random.seed` prend en argument un entier [2] appelé *graine* et qui sert à "fixer" la façon dont les nombres sont générés par la méthode `random.randint`. Cette méthode ne sera pas nécessaire pour le jeu lui-même mais pourra faciliter le débogage du programme.

Voici quelques exemples d'utilisation de ces méthodes :

Le code

```
import random

for i in range(5) :
        print(random.randint(1,100))
```

Affiche par exemple les cinq entiers suivants :

```
62
96
25
98
```

2. Pas nécessairement un entier, en fait. Mais nous n'aurons pas besoin d'utiliser `seed` avec autre chose que des entiers, dans le cadre de ce projet.

```
48
```

Mais si on relance le programme une seconde fois, les nombres générés seront différents.
À l'inverse, le code

```
import random

random.seed(420)
for i in range(5) :
        print(random.randint(1,100))
```

Affichera toujours les même cinq entiers :

```
4
87
47
35
52
```

et ce, même si l'on relance le programme plusieurs fois ou même si l'on change d'ordinateur ! (vérifiez ça chez vous)

1.8 Fonctions à implémenter

Chaque fonction est présentée avec le nom, les paramètres de la fonction, dont chaque type est spécifié après les deux points, et le type de ce que renvoie la fonction après le `->`. Pour les connaisseurs, il s'agit d'une adaptation française de la convention de typage de PEP python.
La notation `List[int]` correspond par exemple à une liste dont les éléments sont des variables de type `int`. La notation `Tuple[int,int]` correspond à un tuple contenant exactement deux éléments, chacun étant un entier.

Pour la description des fonctions ci-après, nous utiliserons les noms de variable `game_board` ou `reference_board` lorsque la fonction s'applique spécifiquement au plateau de jeu ou à celui de référence et nous utiliserons simplement `board` lorsque la fonction peut être utilisée pour l'un ou l'autre des types de plateau (la fonction `print_board` est dans ce cas de figure).
Pour une entrée `board[i][j]` nous appelerons la paire (i, j) la *paire d'indices correspondant* à cette entrée.

— `create_board(n : int, m : int) -> board : List[List[str]]`
Construit un tableau de taille $n \times m$ où chaque case contient le caractère correspondant à une case inexporée.

— `print_board(board : List[List[str]]) -> None`
Affiche le plateau de jeu de taille minimale 4x4 et maximale 100x100, comme spécifié dans la section 1.2.
Dans un premier temps, il est recommandé de se contenter d'un affichage pour des plateaux de taille 10x10 maximum. Cela vous permet de passer plus rapidement à l'implémentation des autres fonctions du jeu. Vous pourrez ensuite revenir à cette fonction et l'adapter pour des plateaux plus grands.

— `get_size(board : List[List[str]]) -> Tuple[int,int]`
Renvoie un tuple de deux entiers (n, m) correspondant aux dimensions du plateau donné en entrée.

— `get_neighbors(board : List[List[str]], pos_x : int, pos_y : int) -> List[Tuple[int,int]]`
Renvoie une liste de tuples $[(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)]$ où chaque (x_i, y_i) correspond à une case voisine de la case `(pos_x, pos_y)`. Notez qu'une case a au maximum 8 cases voisines mais que certaines cases en ont moins !
Par exemple, la case $(0, 0)$ qui se trouve dans l'angle en haut à gauche du tableau n'a que trois cases voisines : $(1, 0), (0, 1)$ et $(1, 1)$.

— `place_mines(reference_board : List[List[str]], number_of_mines : int, first_pos_x : int, first_pos_y : int) -> List[Tuple[int,int]]]`
Cette fonction place aléatoirement les mines sur le plateau `reference_board` après que le joueur ait choisi une première case à dévoiler. Parce que les règles du Démineur stipulent qu'aucune mine ne peut être générée dans le voisinage de la première case dévoilée, les coordonnées de ladite case doivent être entrées en paramètre de cette fonction.
La fonction renvoie la liste des cases contenant une mine. Cette liste sera plus tard utilisée afin de vérifier la condition de victoire du jeu.

— `fill_in_board(reference_board : List[List[str]]) -> None`
Une fois les mines placées sur le plateau de référence, cette fonction doit permettre de calculer le nombre de mines présentes dans le voisinage de chaque case. Cette fonction modifie ensuite le plateau de référence `reference_board` pour y faire apparaître ces nombres.

— `propagate_click(game_board : List[List[str]], reference_board : List[List[str]], pos_x : int, pos_y : int) -> None`
Cette fonction est *a priori* la plus complexe du programme. Elle permet de mettre à jour le plateau de jeu en dévoilant d'un coup toutes les cases adjacentes à la case dévoilée après un clic, lorsque celles-ci n'ont aucune mine dans leur voisinage. Cette fonction doit être appelée à chaque fois qu'une case contenant un `0` est dévoilée (typiquement, la toute première case choisie par le joueur est nécessairement un `0`).
Une description plus détaillée de cette fonction est donnée dans la section 1.6.

— `parse_input(n : int, m : int) -> Tuple[str, int, int]`
Permet au joueur de rentrer une chaîne de caractères selon le format décrit dans la section 1.4, lequel sera ensuite interprété et découpé en un tuple `[action, pos_x, pos_y]`, où `action` peut être soit "`c`" ou "`f`" et où `pos_x` et `pos_y` sont des entiers correspondant à la case `[pos_x, pos_y]` visée par l'action.

— `check_win(game_board : List[List[str]], reference_board : List[List[str]], mines_list : List[tuple], total_flags : int) -> Bool`
Renvoie `True` si la condition de victoire est remplie. Renvoie `False` sinon. Voir la section 1.5 pour une description détaillée des conditions de victoire.

— `init_game(n : int, m : int, number_of_mines : int) -> game_board : List[List[str]], reference_board : List[List[str]], mines_list : List[Tuple[int,int]]`
Initialise les paramètres du jeu en faisant appel aux fonctions définies précédemment. À partir uniquement des dimensions du plateau et du nombre de mines souhaité, la fonction doit
- Générer les plateaux `game_board` et `reference_board`.

- Demander au joueur de choisir la première case à dévoiler.
- Placer les mines sur le plateau de référence en fonction de la première case dévoilée et calculer la liste donnant la position de chaque mine.
- Calculer le nombre de mines dans le voisinage de chaque case et modifier le plateau de référence en conséquence.
- Dévoiler la case choisie précédemment sur le plateau de jeu.
- Propager le premier clic et ainsi dévoiler toutes les cases autour de la première case, tant qu'aucune mine n'est rencontrée.
- Renvoyer toutes les données nécessaires à la suite du jeu : les deux plateaux ainsi que la liste donnant la position de chaque mine.

— `main() -> None`
La fonction principale du programme. Elle commence par récupérer les valeurs `n`, `m` et `number_of_mines` de `sys.argv` (voir 1), puis initialisera toutes les données du jeu. Enfin, une boucle permettra au joueur de rentrer chaque nouvelle action afin de faire avancer le jeu, jusqu'à ce qu'une des conditions de victoire ou que la condition de défaite soit satisfaite. Cette fonction doit aussi afficher l'état courant du plateau de jeu à chaque tour.

Remarque : vous pouvez supposer que les paramètres n et m donnés seront bien compris entre 4 et 100, il n'est pas nécessaire de les tester.

Vous êtes naturellement invité(e)s à implémenter des fonctions supplémentaires si vous le désirez, mais vous devez obligatoirement implémenter telles que demandées les fonctions ci-dessus. Ces fonctions sont celles qui seront testées via un script python et ne laisseront aucune place à l'erreur[3]. Faites également attention à ne pas utiliser de variables globales, celles-ci pourraient vous faire échouer les tests.

1.9 Consignes de remise

Tout d'abord, insistons sur l'importance de respecter toutes les consignes ci-dessus et ci-dessous : **tout manquement aux consignes sera sanctionné d'une note nulle.** Il n'y aura pas d'exception, même si c'est une "bête erreur" d'inattention (ex : nommer son fichier `Demineur.py` au lieu de `demineur.py`, mettre son fichier dans un sous-répertoire au lieu du répertoire racine du repository, etc).

Afin de créer votre repository pour votre projet sur GitHub Classroom, vous devrez utiliser le lien suivant :

`https://classroom.github.com/a/Q0cbj8gw`

Sélectionnez votre Prénom Nom dans la liste. Si vous ne le trouvez pas, contactez le titulaire Gwenaël Joret par email (`gwenael.joret@ulb.be`). Attention, la liste est ordonnée selon le prénom (et non le nom) !

L'entièreté de votre code doit se trouver dans un fichier `demineur.py` (avec une première lettre minuscule). **Votre nom, numéro de matricule et section doivent être indiqués en commentaire au début du fichier `demineur.py`. avec le format suivant :**

3. La fonction `print_board` sera la seule à ne pas être testée de cette façon. Vous recevrez simplement les points si l'affichage fonctionne correctement et est suffisamment clair.

Prenom : <Votre prénom>
Nom : < Votre nom>
Matricule : <Votre matricule>

Le jeu doit être lancé via la commande

```
python3 demineur.py n m number_of_mines
```

depuis le dossier racine de votre repository. Veillez à ce que votre fichier puisse être importé, mettez donc l'appel à la fonction `main()` dans un test conditionnel `if __name__ == '__main__':`.

À travers l'entièreté de votre projet, veillez à respecter à la lettre la signature des fonctions (nom + paramètres + type de la valeur retournée) ainsi que la casse des caractères (respect des majuscules et minuscules).

Uploadez régulièrement votre code sur le serveur ! Une bonne habitude est de le faire systématiquement à la fin de chaque session de travail, par exemple :

```
git commit -am 'courte description des modifications'
git push
```

Échéance pour la remise du fichier `demineur.py` sur GitHub Classroom : **dimanche 18 décembre à 22h**.

Remarques importantes :

— Il n'y a aucun retard possible, GitHub Classroom n'acceptera aucun *commit* passé cette heure. La version de votre code qui sera évaluée sera la dernière version sur GitHub Classroom.

— Si vous rencontrez des problèmes avec l'utilisation de git, nous vous invitons à relire les slides d'introduction à git disponibles sur l'UV, et à poser vos éventuelles questions qui subsisteraient aux guidances. Dans tous les cas, posez vos questions **bien à l'avance**, pas deux-trois jours avant l'échéance.

— Chaque année, quelques étudiant(e)s attendent le dernier moment pour apprendre à utiliser git, et rencontrent des problèmes avec git le weekend de la remise. Il est inutile d'envoyer le projet par email et d'expliquer les problèmes rencontrés, seuls les projets remis via GitHub Classroom seront évalués.

— Faites bien attention à utiliser le lien GitHub Classroom ci-dessus pour créer votre repository, et à ne pas créer vous-même un repository GitHub standard de votre côté. Ces derniers sont publics (!) par défaut, vous partageriez donc votre code avec tous les internautes, ce qui serait sanctionné d'une note nulle. Nous insistons sur ce dernier aspect car quelques cas se sont présentés les années précédentes.