Licences Mathématiques et Informatique Semestre 2 septembre 2025

Université d'Artois
Faculté des Sciences Jean Perrin

U.E. ALGO3

TP1

Exercice 1 Spécifiez puis écrivez une fonction `pair_impair` qui prend en paramètre une pile d'entiers. Elle modifie le contenu de cette pile qui doit, au sortir de la fonction, contenir les mêmes éléments, mais en alternant un entier pair et un entier impair. S'il y a plus d'entiers pairs ou impairs, le *'trop-plein'* se retrouvera au sommet de la pile.

Exemple : considérons la pile `pileA`. Le sommet de cette pile est l'élément le plus à droite.

`pileA`	12	-46	123	85	-111	37	18	13	25	-12	48

Après l'appel `pair_impair(pileA)`, l'état de cette pile pourrait être :

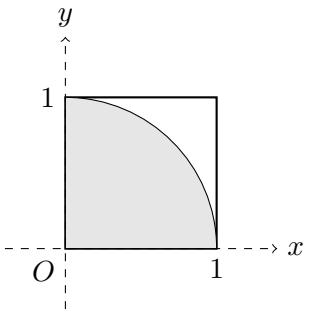
`pileA`	12	123	-46	85	18	-111	-12	37	48	13	25

Vous considérerez que la classe `Pile` est déjà écrite. Vous ne pouvez utiliser que les méthodes de la classe `Pile` que nous avons étudiées en cours. Vous pouvez utiliser d'autres variables locales de type simple ou de type `Pile` ou de type `File`. Les autres structures de données de groupes d'éléments sont interdites (listes, tuples, dictionnaires, ...).

Exercice 2 Approximation de Pi

Vous allez écrire une fonction qui calcule une approximation de π par une méthode statistique qui vous est décrite.

L'idée est la suivante. Considérons un carré de côté 1 : son aire est donc de $1^2 = 1$. Considérons maintenant le quart de cercle de rayon 1 inscrit dans ce carré : son aire est $1/4$ de l'aire du cercle complet de rayon 1, et donc l'aire du quart de cercle est $\pi/4$.



Si on tire aléatoirement un couple de coordonnées x, y dans l'intervalle $[0; 1]$, la probabilité qu'il corresponde à un point à l'intérieur du quart de cercle est *aire du quart de cercle / aire du carré* soit $\pi/4$.

Si maintenant on tire aléatoirement un grand nombre de points dont les coordonnées sont dans l'intervalle $[0; 1]$, on peut regarder pour chaque point s'il est dans le quart de cercle ou non (sa distance au point $(0, 0)$ est inférieure à 1). Il suffit ensuite de calculer $4\times$*(nombre de points dans le quart de cercle / nombre de points tirés)* pour obtenir une approximation de π.

On appelle méthode de Monte-Carlo la famille des algorithmes basés sur un grand nombre d'expériences aléatoires pour calculer une approximation (et donc qui utilisent des techniques probabilistes).

Spécifier puis écrire une fonction `approx_pi(n)` dont le paramètre `n` est un entier qui indique un nombre de points à choisir aléatoirement, et qui affiche, à chaque tir, la nouvelle approximation de π calculée.

Exemple d'affichage :

```
>>> approx_pi(1500000)
...    ## vous n'avez pas tous les affichages ...
Tir : 1499997 - Approximation de Pi : 3.1418649503965677
Tir : 1499998 - Approximation de Pi : 3.1418655224873633
Tir : 1499999 - Approximation de Pi : 3.1418660945773964
Tir : 1500000 - Approximation de Pi : 3.1418666666666666
```

Aides :

1. la fonction `random()` du module `random` retourne un flottant dans l'intervalle $[0; 1]$
2. la fonction `sqrt(x)` du module `math` retourne la racine carrée de `x` (pour `x` $\geq$ `0`).

Exercice 3 Sudoku

On s'intéresse ici à une version simplifiée des sudoku : ces jeux de logique où on doit remplir une grille carrée de dimension 9×9 de tous les nombres de 1 à 9 de telle manière que chaque ligne et chaque colonne contienne tous ces nombres [1].

Pour les questions suivantes, lorsque vous en avez besoin, vous pouvez toujours utiliser une fonction définie dans une question précédente, même si vous n'avez pas su l'écrire.

Q 1 . Lorsqu'un jeu de Sudoku démarre, il contient quelques cases qui sont initialisées. Le jeu consiste à compléter les cases vides.

Écrivez le constructeur de la classe `Sudoku`. Il prend en paramètre une liste de triplets `(lig, col, valeur)`. Le constructeur initialise une matrice carrée d'entiers de dimension 9. Toutes les cases sont initialisées à 0 (qui sera la marque d'une case vide), à l'exclusion des cases indiquées par les triplets `(lig, col, valeur)` : la case aux coordonnées `(lig,col)` contiendra la valeur `val`.

Le constructeur devra également initialiser une seconde matrice de même dimension, mais dont le contenu des cases sera une liste vide. Cette seconde matrice sera nommée `__aide` (et nous nous en servirons plus tard).

```
>>> jeu = Sudoku([(0,0,3),(1,1,1),
  (2,2,5),(1,3,8),(2,3,7),(2,5,1),
  (0,6,2),(1,7,6),(0,8,7),(3,4,7),
  (3,7,2),(3,8,9),(4,1,7),(6,4,3),
  (5,1,8),(6,0,8),(6,2,9),(4,3,3),
  (4,5,2),(5,7,7),(5,8,3),(7,2,4),
  (7,4,8),(8,5,9),(8,6,1)])
```

```
>>> print(jeu)
   0 1 2 3 4 5 6 7 8
  +-+-+-+-+-+-+-+-+-+
0 |3| | | | | |2| |7|
  +-+-+-+-+-+-+-+-+-+
1 | |1| |8| | | |6| |
  +-+-+-+-+-+-+-+-+-+
2 | | |5|7| |1| | | |
  +-+-+-+-+-+-+-+-+-+
3 | | | | |7| | |2|9|
  +-+-+-+-+-+-+-+-+-+
4 | |7| |3| |2| | | |
  +-+-+-+-+-+-+-+-+-+
5 | |8| | | | | |7|3|
  +-+-+-+-+-+-+-+-+-+
6 |8| |9| |3| | | | |
  +-+-+-+-+-+-+-+-+-+
7 | | |4| |8| | | | |
  +-+-+-+-+-+-+-+-+-+
8 | | | | | |9|1| | |
  +-+-+-+-+-+-+-+-+-+
```

Q 2 . Écrivez une méthode `ligne_correcte(self,lig)` qui vérifie qu'il n'y a pas de doublons dans la ligne numéro `lig` (mais il peut y avoir plusieurs zéros puisque le zéro symbolise une case vide).

Écrivez une méthode `colonne_correcte(self,col)` qui vérifie qu'il n'y a pas de doublons dans la colonne numéro `col` (mais il peut y avoir plusieurs zéros puisque le zéro symbolise une case vide).

1. Dans le jeu originel, on décompose en plus le sudoku en sous-grilles de 3×3 qui doivent vérifier cette condition. Nous ne considérons pas cette condition ici.

Écrivez une méthode `matrice_correcte(self)` qui vérifie que la matrice complète est correcte (elle ne contient pas de ligne ou de colonne avec des doublons).

Exemples :

```
>>> jeu.ligne_correcte(5)
True
>>> jeu.colonne_correcte(2)
True
>>> jeu.matrice_correcte()
True
```

Q 3 . Écrivez une méthode `pose_val_sans_aide(self,lig,col,val)` qui pose la valeur `val` dans la case de coordonnées `(lig,col)` si c'est possible sans créer un doublon. La méthode retourne vrai si la valeur a pu être posée, faux sinon.

Exemples :

```
>>> jeu.pose_valeur_sans_aide(0,1,1)
False
>>> jeu.pose_valeur_sans_aide(0,1,6)
True
```

```
>>> print(jeu)
   0 1 2 3 4 5 6 7 8
  +-+-+-+-+-+-+-+-+-+
0 |3|6| | | | |2| |7|
  +-+-+-+-+-+-+-+-+-+
1 | |1| |8| | | |6| |
  +-+-+-+-+-+-+-+-+-+
2 | | |5|7| |1| | | |
  +-+-+-+-+-+-+-+-+-+
3 | | | | |7| | |2|9|
  +-+-+-+-+-+-+-+-+-+
4 | |7| |3| |2| | | |
  +-+-+-+-+-+-+-+-+-+
5 | |8| | | | | |7|3|
  +-+-+-+-+-+-+-+-+-+
6 |8| |9| |3| | | | |
  +-+-+-+-+-+-+-+-+-+
7 | | |4| |8| | | | |
  +-+-+-+-+-+-+-+-+-+
8 | | | | | |9|1| | |
  +-+-+-+-+-+-+-+-+-+
```

Q 4 . Écrivez une méthode `val_manquantes_ligne(self,lig)` qui retourne la liste des valeurs qui ne sont pas présentes dans la ligne numéro `lig`.

Écrivez la méthode `est_dans_colonne(self,col,val)` qui teste si `val` est présent dans la colonne `col`.

Écrivez la méthode `val_manquantes_colonne(self,col,vals)` qui retourne la liste des valeurs dans `vals` qui sont manquantes dans la colonne `col`.

Écrivez la méthode `val_possibles(self,lig,col)` qui retourne la liste des valeurs possibles pour la case de coordonnées `lig,col`.

Q 5 . Écrivez la méthode `construit_aide(self)` qui affecte à chaque case de `self.__aide` la liste des valeurs possibles pour la case de la matrice aux mêmes coordonnées.

Écrivez une méthode `aide(self,lig,col)` qui retourne le contenu de l'aide pour la case en `lig,col`.

Exemples :

```
>>> jeu.aide(0,2)
[1, 8]
>>> jeu.aide(8,3)
[2, 4, 5, 6]
```

```
>>> jeu.aide(8,1)
[2, 3, 4, 5]
>>> jeu.aide(8,0)
[2, 4, 5, 6, 7]
```

Q 6 . Écrivez une méthode `pose_val_avec_aide(self,lig,col,val)` qui pose la valeur `val` dans la case de coordonnées `(lig,col)` si c'est possible (en fonction de ce que contient `self.__aide`). Si la valeur peut être posée, le contenu de `__aide` doit être mis-à-jour. La méthode retourne vrai si la valeur a pu être posée, faux sinon.

Exemples :

```
>>> jeu.pose_valeur_avec_aide(8,3,4)
True
>>> jeu.aide(8,0)
[2, 5, 6, 7]
>>> jeu.aide(8,1)
[2, 3, 5]
>>> jeu.aide(8,3)
[]
```

Aide : la méthode `remove(self, val)` qui s'applique sur une liste supprime la première occurrence de `val` dans la liste `self`. Attention, `val` doit être présent dans la liste (sinon, il y a une erreur).

```
>>> l = [1,2,3,2,4,1]
>>> l.remove(2)
>>> l
[1, 3, 2, 4, 1]
>>> l.remove(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```
