TP : RÉSOLUTION DE SUDOKU

1. Enregistrez votre travail dans un dossier `TP_sudoku`.
Le but du TP de résoudre des grilles de sudoku.

Généralités

Une grille de sudoku sera représentée par une liste de neuf listes python. Chaque sous-liste représentera les cœfficient situés sur une ligne. Les cœfficients inconnus seront codés par l'entier 0.

Par exemple, la grille suivante, sera codée par la liste de listes :

·	·	3	·	2	·	6	·	·
9	·	·	3	·	5	·	·	1
·	·	1	8	·	6	4	·	·
·	·	8	1	·	2	9	·	·
7	·	·	·	·	·	·	·	8
·	·	6	7	·	8	2	·	·
·	·	2	6	·	9	5	·	·
8	·	·	2	·	3	·	·	9
·	·	5	·	1	·	3	·	·

```
grille = [[0,0,3,0,2,0,6,0,0],
          [9,0,0,3,0,5,0,0,1],
          [0,0,1,8,0,6,4,0,0],
          [0,0,8,1,0,2,9,0,0],
          [7,0,0,0,0,0,0,0,8],
          [0,0,6,7,0,8,2,0,0],
          [0,0,2,6,0,9,5,0,0],
          [8,0,0,2,0,3,0,0,9],
          [0,0,5,0,1,0,3,0,0]]
```

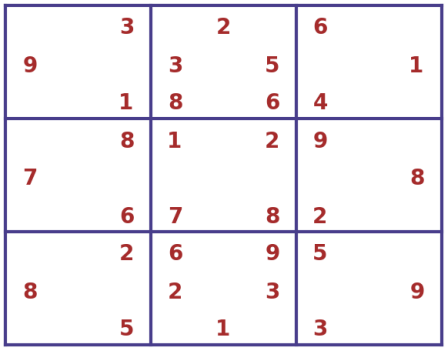
On donne aussi dans le fichier `affichage_sudoku.py` (ouvrez-le !) une fonction `affiche_tout(grille)` permettant d'obtenir l'affichage complet d'une grille, une fonction `affiche(v, i, j)` permettant d'afficher seulement la valeur v en position (i, j) et renvoyant l'objet correspondant à cet affichage, et enfin `efface(chiffre)` prenant en argument cet objet et l'effaçant.

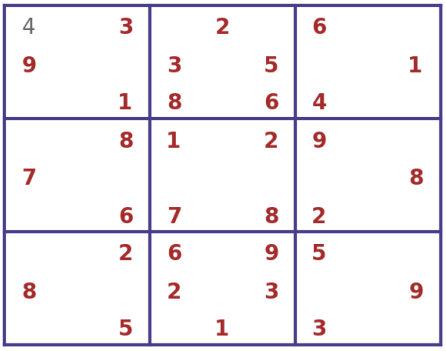
On placera dans le dossier `TP_sudoku` le fichier nommé `affichage.py` et on insérera dans le fichier de travail la ligne :

```
from affichage_sudoku import *
```

Et l'appel de `affiche_tout(grille)` où `grille` est la grille précédente permet d'obtenir l'affichage ci-dessous dans une fenêtre indépendante, avec l'avantage de pouvoir mettre à jour cet affichage à loisir.

Puis `chiffre = affiche(4, 0, 0)` modifie l'affichage en ajoutant le chiffre 4.





Et enfin `efface(chiffre)` permet de revenir à l'état précédent.

2. Première lecture dans un fichier

2.a) Écrire une fonction `transforme(chaine)` qui permet d'obtenir une grille de sudoku (liste de listes) à partir d'une chaîne de caractères de la forme ci-dessous :

```
003020600900305001001806400008102900700000008006708200002609500800203009005010300
```

2.b) Écrire une fonction `lire(fichier)` permettant de lire un fichier comme le fichier `sudoku17.txt` et renvoyant la liste des grilles contenues dedans.

3. Deuxième lecture dans un fichier

3.a) Écrire une fonction `transforme2(chaine)` qui permet d'obtenir une grille de sudoku (liste de listes) à partir d'une chaîne de caractères de la forme ci-contre. *(Utiliser la méthode `split` des chaînes de caractères.)*

```
003020600
900305001
001806400
008102900
700000008
006708200
002609500
800203009
005010300
```

3.b) Écrire une fonction `lire2(fichier)` permettant de lire un fichier comme le fichier `p096_sudoku.txt` et renvoyant la liste des grilles contenues dedans. Ce fichier provient du problème 96 du project Euler :

https://projecteuler.net/problem=96

Principe de résolution

On va résoudre informatiquement les sudokus en appliquant une méthode dite de **backtracking**. Le principe est le suivant : on va tester toutes les possibilités

jusqu'à en trouver une qui fonctionne. Lorsque l'on se trouve dans une impasse (grille impossible), on revient en arrière (backtracking) et on teste une autre possibilité. La récursivité nous permettra de mettre ce procédé en pratique (assez) facilement.

Le couple des coordonnées d'un élément de la grille de sudoku étant un couple (i, j) où i est le numéro de la ligne et j le numéro de la colonne, on obtient cet élément grâce à `grille[i][j]`.

La grille est alors décomposée en 9 sous-carrés dont les éléments ont des coordonnées de la forme $(i+3m, j+3n)$ où i et j varient entre 0 et 2 et où m et n sont fixés (pour un sous-carré) entre 0 et 2 également.

4. Créer une liste `sous_carres` contenant les 9 sous-listes contenant elles-mêmes les coordonnées (x,y) des éléments de chacun des 9 sous-carrés de la grille de sudoku. Il s'agit donc d'une liste de listes de couples.

On pourra utiliser une liste en compréhension de la forme

```
[[( , ) for in for in ] for in for in ]
```

On obtiendra une numérotation des sous-carrés de la forme :

0	1	2
3	4	5
6	7	8

Par exemple, `sous_carres[1]` est la liste

```
[(0,3),(0,4),(0,5),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5)]
```

car le deuxième sous-carré contient dans les trois premières lignes, les éléments des colonnes 3, 4, 5.

5. Écrire une fonction `quel_sous_carre(i,j)` renvoyant, pour un couple d'entiers $(i, j) \in [\![0,8]\!]^2$ le numéro du (seul) sous-carré dans lequel il se trouve. Ce numéro s'obtient très simplement à l'aide de i et j et en utilisant la division entière `//`.

6. On appelle voisin de (i, j) tout couple de coordonnées correspondant à une case, différente de (i, j), se trouvant sur la même ligne, sur la même colonne ou dans le même sous-carré.

Écrire une fonction `voisins(i,j)` qui renvoie la liste des voisins de (i, j). On fera en particulier attention à ce que chaque couple apparaissent une et une seule fois.

7. En utilisant cette fonction construire une liste de listes `voisinage` telle que `voisinage[i][j]` soit la liste des voisins de (i, j). Cette liste sera calculée une fois pour toutes, et pourra être utilisée telle quelle dans les algorithmes (variable globale).

8. Écrire une fonction `valeurs(grille,i,j)` renvoyant une liste vide si la case (i, j) est déjà remplie, la liste des valeurs possible en position (i, j) en fonction de ses voisins sinon.

On pourra commencer par remplir un tableau de booléens permettant de savoir si oui ou non la valeur k est possible en (i, j) pour chaque k entre 1 et 9 puis après le parcours des voisins, renvoyer la liste demandée.

Une première version naïve

9. On va écrire une fonction récursive `remplir(grille,n)` permettant de faire effectivement la résolution par backtracking en modifiant directement `grille`. n est le numéro (entre 0 et 81) de la prochaine case à remplir. Le fonction renverra `True` si la résolution est terminée, `False` si la grille est impossible. Pour cela, on propose d'utiliser l'algorithme suivant :

- Si $n = 81$, renvoyer Vrai, la résolution est terminée.
- Sinon, rechercher les coordonnées (i, j) de la case à remplir correspondant au numéro n.
- Si elle est déjà remplie, remplir la suivante, renvoyer le résultat.
- Sinon pour chaque valeur v valide en (i, j),
  - ⋆ Placer la valeur v en position (i, j) dans la grille.
  - ⋆ Afficher la nouvelle valeur de la grille.
  - ⋆ Remplir la valeur $n + 1$ de la grille.
  - ⋆ Si la résolution a été possible, renvoyer Vrai : la résolution est terminée.
  - ⋆ Sinon, effacer le chiffre et passer à la valeur suivante.
- Si la résolution n'a été possible pour aucune valeur v, alors réinitialiser la case (i, j) à 0 et renvoyer Faux : c'est une impasse (backtrack).

10. Écrire enfin une fonction `sudoku(grille)` qui affiche la grille initiale et effectue la résolution de celle-ci.

Déjà fini ?

Plusieurs possibilités. Vous pouvez, au choix :

- Écrire une fonction `teste(grille)` permettant de tester si une grille de sudoku (pas nécessairement pleine) est cohérente.
- Résoudre le problème 96 du Project Euler. On conseille de désactiver l'affichage qui ralentit beaucoup la résolution. *(Chez moi, cela prend une quinzaine de secondes avec la version force brute et moins d'une seconde avec l'algorithme amélioré ci-après.)*
- Implémenter la version plus efficace suivante.
- Modifier le code pour obtenir toutes les solutions possibles en cas de solutions multiples.
- Écrire une fonction permettant à un utilisateur de remplir de façon interactive une grille de sudoku.

Une version un peu plus efficace

Naturellement, lorsque l'on résout une grille, on commence par chercher des valeurs dans les emplacements pour lesquels la ligne, et la colonne et le sous-carré sont le plus remplis.

On se propose de mettre en œuvre cette amélioration.

L'idée est que l'on va manipuler et faire évoluer une liste de listes de listes de valeurs possibles dans chaque case de la grille. On désignera par `liste_val` cette liste de liste de listes.

11. Écrire une fonction `nb_non_rempli(grille)` renvoyant le nombre de case non remplies (de zéros, donc) dans la grille.

12. Écrire une fonction `min_non_rempli(liste_val)` renvoyant, à partir d'une liste de listes de listes de valeurs possibles, les coordonnées `(imin,jmin)` d'une case non remplie (au moins une valeur possible) ayant un minimum de valeurs possibles. On suppose ici qu'il y a au moins une valeur possible pour au moins une case de la grille.

13. Écrire une fonction `supprime(L, v)` renvoyant une copie de la liste `L` débarrassée de toute occurrence de `v`.

14. Écrire une fonction `mise_a_jour(liste_val, i, j, v)` prenant en argument une liste de listes de listes de valeurs possibles, des coordonnées (i, j) et une valeur v (que l'on veut placer en (i, j)) et renvoyant :

- une copie de `liste_val` (**attention : copie de liste de liste de liste !**) dans laquelle on aura vidé la liste des valeurs possibles en (i, j) et supprimé toutes les apparitions de v dans les valeurs possibles pour ses voisins **s'il y a d'autres valeurs possibles**,
- la liste vide si l'un des voisins de (i, j) ne peut prendre que la valeur v initialement (ce qui correspondra à une impasse).

15. Écrire une fonction récursive `resolution_mieux(grille, liste_val, n)` prenant en argument une grille, la liste de listes de listes des valeurs possibles correspondante, et un entier n égal au nombre de cases non encore remplies dans la grille utilisant les fonctions précédentes pour résoudre (récursivement) le problème. Cette fois, c'est une case ayant le minimum de valeurs possibles que l'on tentera de remplir avec les différentes valeurs possibles données par `liste_val`, et pour chacune de ces valeurs en (i, j), calculant une copie mise à jour de cette liste et rappelant récursivement la fonction si l'on n'est pas dans une impasse.

De nouveau, on renverra vrai si la résolution est terminée, faux si elle n'est pas possible (impasse).

16. Écrire enfin la fonction `sudoku_mieux(grille)`.

17. Constater empiriquement un gain de facteur environ 20 (en moyenne) sur le temps d'exécution.

On pourrait encore affiner notre résolution et ce n'est pas la seule technique permettant de venir à bout de ces petites grilles : beaucoup de techniques classiques (et passionnantes) peuvent être appliquées à ce casse-tête. Cependant, nous ne connaissons pas d'algorithme vraiment efficace pour résoudre les grilles de sudoku (lorsque leur taille devient grande) car il s'agit d'un problème NP-complet.

(Voir par exemple, Centrale 2014, option informatique. Beaucoup de ressources disponibles sur le web. Articles de JP Delahaye de décembre 2005 et janvier 2015 dans Pour La Science.)