TP guidé: La bataille navale

La bataille navale, appelée aussi touché-coulé, est un jeu de société dans lequel deux joueurs doivent placer des « navires » sur une grille tenue secrète et tenter de « toucher » les navires adverses. Le gagnant est celui qui parvient à couler (c'est-à-dire toucher toutes les cases) tous les navires de l'adversaire avant que tous les siens ne le soient.

La bataille navale oppose deux joueurs. Chaque joueur dispose de deux grilles carrées de côté 10, dont les colonnes sont numérotées de 1 à 10 et les lignes de A à J, ainsi que d'une flotte composée de quelques bateaux de deux à cinq cases de long.

L'une des grilles représente la zone contenant sa propre flotte. Au début du jeu, chaque joueur place ses bateaux sur sa grille, en s'assurant que deux bateaux ne sont pas adjacents. L'autre représente la zone adverse, où il cherchera à couler les bateaux de son adversaire.

Chaque joueur, à son tour, annonce une case (par exemple « B6 »), et son adversaire lui répond si le tir tombe à l'eau ou au contraire s'il touche un bateau. Dans ce dernier cas, il annonce « touché » s'il reste des cases intactes au bateau ciblé, et « touché-coulé » si non.

Chaque joueur possède les mêmes navires, dont le nombre et le type dépendent des règles du jeu choisies.

Liste des navires :

- 1 Porte-avions (5 cases) ;
- 1 Croiseur (4 cases) ;
- 2 Contre-torpilleurs (3 cases) ;
- 1 Sous-marin (2 cases).

1 TP 1

L'élément fondamental de ce jeu est une grille. Et plus précisément plusieurs grilles.

Combien de grilles faudra t-il à votre avis ?

Dans une grille il faudra représenter plusieurs éléments (bateaux/eau) mais aussi plusieurs états (touché/raté). Au total combien de valeurs différentes peut prendre chaque case de la grille ? Doit-on tenir compte du type de bateaux dans la grille ? Si oui combien d'états différents faut-il alors ? Comment ces données vont-elles être représentées informatiquement ?

Prenez-le temps de réfléchir à ces questionnements avant de lire les réponses dans le cadre ci-dessous.

Il faudra 4 grilles (2 par joueurs). En effet chaque joueur possède sa propre grille, ainsi que la grille des tirs effectués vis à vis de la grille de l'adversaire.

L'eau peut avoir deux états : libre / tir raté
Il y a 5 bateaux, dans deux états chacun : libre / touché ; soit 5x2 = 10 états (coulé n'est qu'un cas particuliers où tout est touché)
Nous avons un total de 12 états.
Pour ce TP ces états seront représentés par des valeurs entières, dans une liste 2D :

- l'eau : 1
- sous-marin : 2
- contre-torpilleur n°1 : 3
- contre-torpilleur n°2 : 4
- croiseur : 5
- porte-avion : 6
- Toute case touchée correspondra au négatif des valeurs ci-dessus.

Partie 1

Créez manuellement une petite grille qui vous servira à tester le travail qui suit.

```
# exemple avec 4x4
grille_test = [[1,2, 3, 4], [4, 3, 2, 1], [1, 1, 1, 1], [7, 1, 6, 2]]
```

1. Écrire une fonction/procédure **`Afficher_Grille()`** qui prend en entrée une grille (liste 2D) de taille quelconque et affiche son contenu par un simple print(). Testez votre fonction en l'appelant pour vérifier le résultat (pensez à faire cela à chaque étape).

2. Ensuite modifiez le code pour afficher chaque case de la liste externe, à la ligne les unes des autres. Cela doit donner :

```
[1, 2, 3, 4]
[4, 3, 2, 1]
[1, 1, 1, 1]
[7, 1, 6, 2]
```

3. puis, en concaténant chaque éléments de chaque sous-liste, faites en sorte d'afficher :

```
1234
4321
1111
7162
```

4.

5. Modifier cette fonction pour que l'eau soit affichée par le caractère espace.

6. Modifier cette fonction pour que les bateaux soient affichés avec le caractère unicode U+2588

7. Continuer de modifier la fonction pour que ”raté” (tir dans l'eau) s'affiche avec le caractère unicode U+2591
8. Et finalement, qu'un bateau touché s'affiche avec U+2573
9. En cas de valeur inconnue vous afficherez un arobase @ (mais en principe ce ne sera jamais le cas) Voici un exemple de résultat :

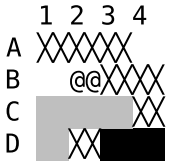
10. Testez votre code sur Moodle.

Pensez à commenter/docstringuer votre code avant de le soumettre !

Partie 2

Nous avons un premier jet d'affichage, mais il n'est pas encore très esthétique. Nous allons encore modifier cette fonction pour améliorer cela.

1. L'affichage semble étiré verticalement, modifier votre code pour que chaque case s'affiche avec deux caractères identiques.
2. Modifiez la fonction pour qu'elle affiche en première ligne le numéro de colonne suivi d'un espace.
3. Modifiez la fonction pour qu'elle affiche en première colonne la lettre de chaque ligne (A, B, C, ...) suivie d'un espace.
4. Voici un exemple de résultat :



5. Testez votre code sur Moodle.

Pensez à commenter/docstringuer votre code avant de le soumettre !

Partie 3

Écrire une fonction `Generer_Grille()` qui prend en entrée une taille et qui génère et renvoie une liste 2D carrée ayant un côté de cette taille et dont chaque case est remplie d'eau.

Testez votre fonction en l'appelant, puis si cela vous semble bon, testez-la sur Moodle.

Pensez à commenter/docstringuer votre code avant de le soumettre !

2 TP2

L'objectif de ce TP est mettre en place la boucle de jeu. Nous supposons ici que les bateaux ont déjà été placés par les deux joueurs. Le placement des bateaux sera abordé en TP3.

> Ce TP étant la suite du projet guidé, vous écrirez le code dans le même fichier de script Python que celui entamé en dernière séance.

Partie 1 - Saisie des coordonnées

Afin de tirer un missile, il faut tout d'abord donner les coordonnées de la case visée. Pour cela il faut effectuer une saisie utilisateur. Mais un utilisateur peut se tromper. Il est donc fondamental de vérifier sa saisie.

Voici une fonction (à copier dans votre code) dont le rôle est simplement d'afficher un message (elle semble peut utile à ce stade, mais prendra de l'importance plus tard dans le TP guidé)

```
def Afficher_msg(msg):
    '''
    Affiche un message (chaine donnée en entrée) à l'utilisateur.
    '''
    print(msg)
```

Consignes:

Compléter, docstringuez et commentez la fonction suivante qui demande à l'utilisateur de saisir des coordonnées sous la forme d'une chaîne "ligne colonne" (ex: B6, sans espace), et qui renvoie cette saisie uniquement lorsqu'elle est valide. Pensez que l'utilisateur peut saisir des minuscules, dans ce cas le renvoi sera malgré tout en majuscules. Lorsque la saisie est invalide, on pense à afficher un message d'erreur à l'utilisateur, et on recommence une nouvelle saisie.

```
def Saisie_Coords():
    Afficher_msg(______)
    saisie = input()
    while True:
        saisie = saisie.upper()
        if len(saisie) ______ and len(saisie) ______:
            if saisie[0] in ______:
                if saisie[1:].isdigit() and ______ and ______:
                    return saisie
        Afficher_msg(______)
        saisie = ______
```

Tester votre fonction sur votre ordinateur, puis sur Moodle lorsque cela vous semble OK.

Partie 2 - Tir de missile

Dans ce TP, la liste des bateaux de chaque joueur est une liste 2D dont les listes internes contiennent dans l'ordre : le nom du bateau (indice 0) ; sa taille (indice 1) ; et le nombre de fois qu'il a été touché (indice 2).

Voici la liste pour un joueur, à copier dans votre code.

```
bateaux_joueur = [["porte-avion",5,0], \
                  ["croiseur",4,0], \
                  ["contre-torpilleur 1 ",3,0], \
                  ["contre-torpilleur 2",3,0], \
                  ["sous-marin",2,0]]
```

Générez la liste pour le second joueur (qui sera l'ordinateur).

Voici également deux grilles (océan) préremplies pour vos tests :

```
grille_joueur=[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 2, 1, 1, 1, 1, 1, 1, 1, 1], \
[1, 2, 1, 5, 5, 5, 1, 1, 1, 1], [1, 2, 1, 1, 1, 1, 1, 1, 1, 1], \
[1, 2, 1, 1, 1, 1, 1, 1, 1, 1], [1, 2, 1, 1, 1, 6, 6, 1, 1, 1], \
[1, 1, 1, 1, 1, 1, 1, 1, 1, 4], [1, 1, 1, 1, 3, 3, 3, 3, 1, 4], \
[1, 1, 1, 1, 1, 1, 1, 1, 1, 4], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

grille_ordi=[[1, 1, 1, 5, 5, 5, 1, 1, 1, 1], [3, 1, 1, 1, 1, 1, 1, 1, 1, 1], \
[3, 1, 1, 1, 1, 1, 1, 1, 1, 1], [3, 1, 1, 1, 1, 1, 6, 1, 1, 1], \
[3, 1, 1, 1, 1, 1, 6, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1], \
[1, 2, 2, 2, 2, 2, 1, 4, 1, 1], [1, 1, 1, 1, 1, 1, 1, 4, 1, 1], \
[1, 1, 1, 1, 1, 1, 1, 4, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
```

Et enfin, une fonction (à copier dans votre code) qui transforme des coordonnées de tir en numéro de colonne et de ligne de la liste 2D (grille de jeu).

```
def Coords2Nums(pos):
    '''
    Entrée : une chaine représentant des coordonnées : exemple "B6"
    Sortie : deux entiers représentants respectivement les numéros
             de ligne et colonne : 1, 5 (indicé à partir de 0)
    '''
    return ord(pos[0])-65, int(pos[1:])-1
```

Consignes:

Écrire une fonction `Tir()` qui prend en entrée, et dans cet ordre : la grille de placement de l'adversaire (avec les bateaux), la grille des états connus (tirs tentés par le joueur en cour), les coordonnées du tir (chaine du genre "B6"), et la liste des bateaux de l'adversaire. Elle renvoie un entier dont la signification est : 0 pour raté, 1 pour touché, 2 pour coulé, et -1 pour touché mais le bateau avait déjà été touché à cet endroit

Voici un décomposé des étapes effectuées par cette fonction :

- Transformer et conserver les coordonnées de tir, en numéros de ligne et colonne de grille (utilisez un appel à la fonction donnée ci-dessus)
- Récupérer et conserver la valeur de la case correspondante à partir de la grille de placement de l'adversaire.
- Mettre à jour la case des deux grilles. On rappelle que lorsque un tir a été effectué, les valeurs des grilles deviennent négatives. La case de la grille d'états prend la même valeur (négative) que la grille de placement. Vérifiez votre fonction avec plusieurs tirs, en affichant les grilles résultantes.
- Lorsque la case contient de l'eau, votre fonction doit renvoyer 0. Testez.
- Lorsque la case contient un bateau, il faut vérifier plusieurs situations :
  - La case avait déjà été ciblée, dans ce cas on renvoie -1. Dans les variables que vous avez à ce stade, laquelle permet de savoir que la case avait déjà été ciblée ? Codez et vérifiez.
  - La case n'avait pas encore été ciblée. Dans ce cas il faut incrémenter la valeur "touché" dans la liste des bateaux, pour le bon bateau (comment déterminer quel est le bon bateau dans la liste ?). Ensuite, là encore il reste plusieurs situations :
    * Il faut déterminer si le bateau est juste touché, ou bien touché et coulé : comment déterminer qu'un bateau est simplement touché ou coulé ? Les informations contenues dans la liste des bateaux vont vous mettre sur la piste.
    * Lorsque le bateau est coulé renvoyer 2, ou juste touché renvoyer 1.
    * Codez et vérifiez.
- Quand cela vous semble OK, validez sur Moodle.

Pensez à commenter/docstringuer votre code avant de le soumettre !

Partie 3 - Boucle de jeu

Dans cette partie, nous allons (enfin) imbriquer l'ensemble des fonctions pour commencer à obtenir un programme jouable.

Commencez par nettoyer toutes les instructions vous permettant de tester vos codes (les appels de fonction, les print, ...)

Voici deux fonctions, à recopier:

```
from random import randint # importation de bibliothèque, sera vue dans un autre TP

def Ordi_Coords(grille):
    '''
    Obtention des coordonnées de tir choisies par l'ordinateur.
    Entrée : la grille à analyser pour faire le choix (grille d'état du joueur)
    Sortie : coordonnées, genre "B6"
    '''
    return chr(randint(65, 74))+str(randint(1, 10))

def Gagne(bateaux):
    '''
    Détermine si tous les bateaux ont été coulés
    Entrée : une liste de bateaux
    Sortie : un booléen, True s'ils ont tous été coulés, False sinon
    '''
    for bateau in bateaux:
        if bateau[1]!=bateau[2]: # bateau non coulé ? (taille différent de nb fois touché)
            return False
    return True
```

Le choix des coordonnées par l'ordinateur est ici totalement aléatoire. Au TP5 nous vous donnerons une fonction plus efficace. D'ici là, ceux qui le souhaitent peuvent s'amuser à réfléchir et implémenter une ou plusieurs statégies de tir. La seule contrainte est que l'entrée et la sortie de la fonction soit identique (E: une grille, S: des coordonnées).

Sur la page suivante, vous trouverez un algorigramme représentatif de la fonction principale, nous l'appellerons `Boucle_Jeu()`. Nous verrons plus tard comment permettre à un joueur qui a touché de rejouer, pour l'instant c'est strictement l'un après l'autre.

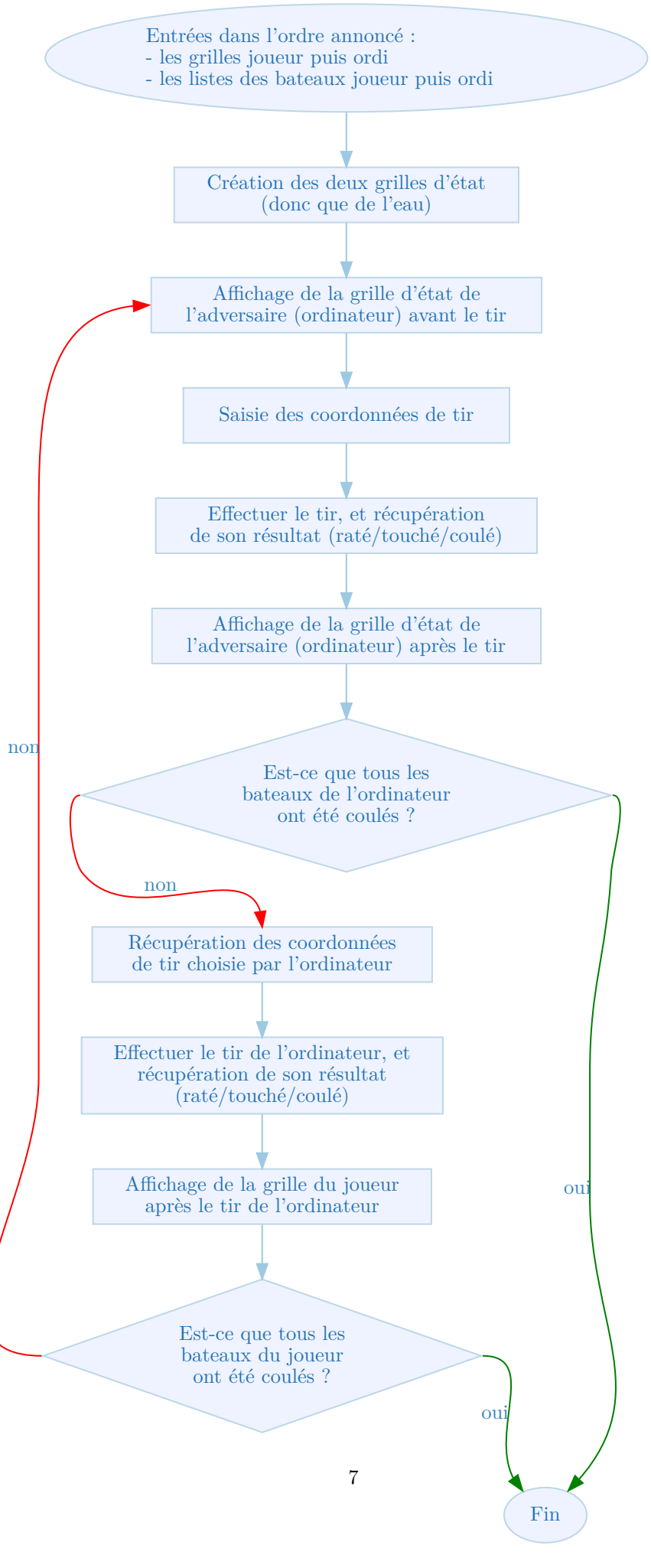
Transcrivez ce dernier en fonction Python.

Allez-y par étapes. Et à chaque étape vérifiez que le comportement de la fonction est cohérent avec vos attentes.

Remarque : lorsqu'une flèche de l'algorigramme remonte en arrière, cela signifie qu'il y a une boucle. De quelle bouche va t-il s'agir ? for ? while ? Si c'est un while quel est de test de continuité ? Cette boucle doit-elle être interrompue prématurément, autrement que par son test ? Si oui quand ?

Quand cela vous semble OK, validez sur Moodle.

Pensez à commenter/docstringuer votre code avant de le soumettre !


Entrées dans l'ordre annoncé :
- les grilles joueur puis ordi
- les listes des bateaux joueur puis ordi
Création des deux grilles d'état
(donc que de l'eau)
Affichage de la grille d'état de
l'adversaire (ordinateur) avant le tir
Saisie des coordonnées de tir
Effectuer le tir, et récupération
de son résultat (raté/touché/coulé)
Affichage de la grille d'état de
l'adversaire (ordinateur) après le tir
Est-ce que tous les
bateaux de l'ordinateur
ont été coulés ?
non
oui
Récupération des coordonnées
de tir choisie par l'ordinateur
Effectuer le tir de l'ordinateur, et
récupération de son résultat
(raté/touché/coulé)
Affichage de la grille du joueur
après le tir de l'ordinateur
Est-ce que tous les
bateaux du joueur
ont été coulés ?
non
oui
Fin

3 TP3

Les objectifs de ce TP sont d'améliorer les structures de données et revoir l'affichage.

Partie 1 - Les bateaux

Dans le TP précédent les bateaux ont été représentés par des listes, dont l'indice avait une signification (0: nom, 1: taille, 2: touché (nombre de fois))

Maintenant que vous avez vu le concept de dictionnaire dans le Notebook, on comprends que ce type de structure sera plus pertinent.

Chaque bateau va être représenté par un **dictionnaire** dont les clés sont le ***nom***, la ***taille***, et le nombre de fois qu'il a été ***touché***.

Quelles seront les valeurs pour la clé ***touché*** ?

Écrire une fonction **gen1bat()** qui prend en entrée un nom et une taille, puis renvoie un dictionnaire représentant un bateau.

Testez cette fonction sur votre ordinateur.

Pour chaque bateau, les valeurs associées aux deux premières clés seront :

nom	taille
porte-avion	5
croiseur	4
contre-torpilleur 1	3
contre-torpilleur 2	3
sous-marin	2

Les cinq bateaux seront stockés dans une liste, dont l'ordre est important !

Écrire une fonction **Generer_Bateaux()** qui utilise la fonction **gen1bat()** pour générer puis renvoyer une liste avec les cinq bateaux. Les noms des bateaux devront être identiques aux spécifications ci-dessus, au caractère près.

Testez cette fonction sur votre ordinateur. Puis validez les deux fonctions sur Moodle lorsque cela vous semble OK.

Pensez à docstringuer votre code avant de le soumettre !

Ces fonctions servirons dans un prochain TP

Partie 2 - Les joueurs

À ce stade les données de chaque joueurs sont stockées dans des objets différents (grille_joueur, grille_ordi, bateaux_joueur, etc...)

Maintenant que nous avons connaissance des dictionnaires, il sera plus pertinent d'enregistrer toutes les données de chaque utilisateur dans un dictionnaire.

Écrire une fonction **Generer_Joueur()** qui ne prend rien en entrée, et renvoie les données initiales d'un joueur sous forme de dictionnaire :

clé	contenu
grille	la grille de jeu de départ (remplie d'eau)
tirs	la grille des tirs tentés contre l'adversaire
bateaux	la liste des bateaux

Pensez à utiliser les différentes fonctions de génération de données du TP1 et TP3.

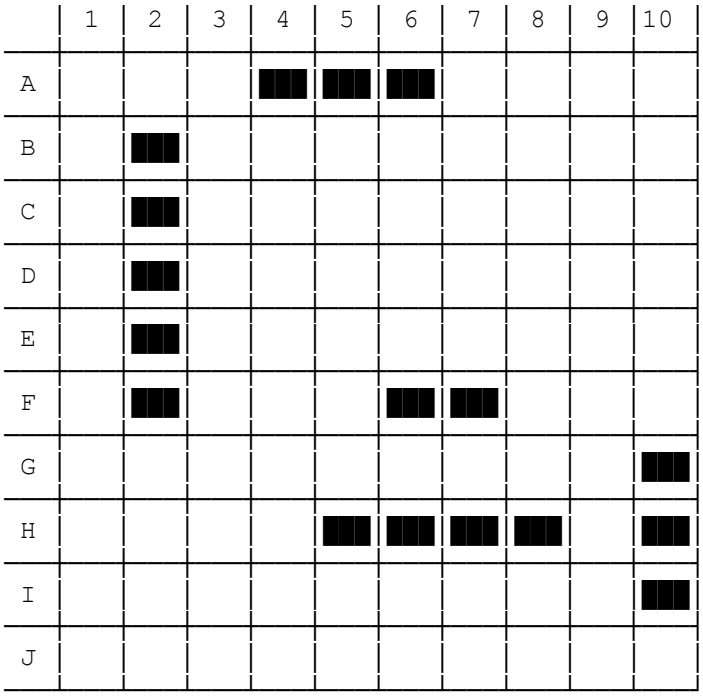
Testez cette fonction sur votre ordinateur, puis validez-la sur Moodle lorsque cela vous semble OK.

Pensez à docstringuer votre code avant de le soumettre !

Cette fonction servira dans un prochain TP

Partie 3 - Un affichage de grille encore plus beau

Voici un exemple de ce que l'on souhaite obtenir :



Vous allez partir de la fonction Afficher_Grille() effectuée en TP1, et la modifier.
Comme tout grand problème, il faut le décomposer en petits problèmes :

- Première ligne : `   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |`
  elle débute par 3 espaces, puis '│'. Ensuite pour chaque valeur on observe : qu'elle est centrée dans une zone de 3 caractères, puis suivie d'un '│'. Modifiez le début de votre fonction, en utilisant le formattage de chaine vu dans le Notebook. Testez.
- Seconde ligne : `---┼---┼---┼---┼---┼---┼---┼---┼---┼---┤`
  on duplique plusieurs fois la suite de caractères : 3 '─' et un '┼'. Souvenez-vous qu'il est possible de "multiplier" une chaîne de caractères pour la dupliquer plusieurs fois (TP1). Sauf à la fin, où on a 3 '─' et un '┤' Modifier votre fonction et testez.
- Troisième ligne : `A |   |   |   |███|███|███|   |   |   |   |`
  on débute par la lettre centrée dans une zone de 3 caractères suivie d'un '│'. Puis chaque case de la grille est affichée sur 3 caractères suivie par un '│'. Modifier votre fonction et testez.
- Jusqu'à la ligne 'J', le processus "Seconde ligne puis Troisième ligne" (les deux points ci-dessus) se répètent. Modifier votre fonction et testez.
- Dernière ligne : `---┴---┴---┴---┴---┴---┴---┴---┴---┴---┘`
  enfin on a une duplication de 3 '─' et un '┴', et à la fin 3 '─' et un '┘'. Modifier votre fonction et testez.

Testez cette fonction sur votre ordinateur, puis validez-la sur Moodle lorsque cela vous semble OK.

Pensez à docstringuer et commenter votre code avant de le soumettre !

4 TP4

Partie 1 - Sauvegarde

En pleine partie, vous devez tout stopper pour attaquer une autre tache. Vous aimeriez pouvoir enregistrer votre partie, afin de la reprendre plus tard.

Voici une fonction "code à trou" dont le rôle est de sauvegarder les données de la partie dans un fichier. Ce code est à compléter, commenter et docstringuer.

Son rôle est de sauvegarder les informations dans un fichier *sauvegarde.json*, lui-même enregistré dans un sous-dossier *data* du dossier courant du projet :

***dossier_du_projet*/data/sauvegarde.json**

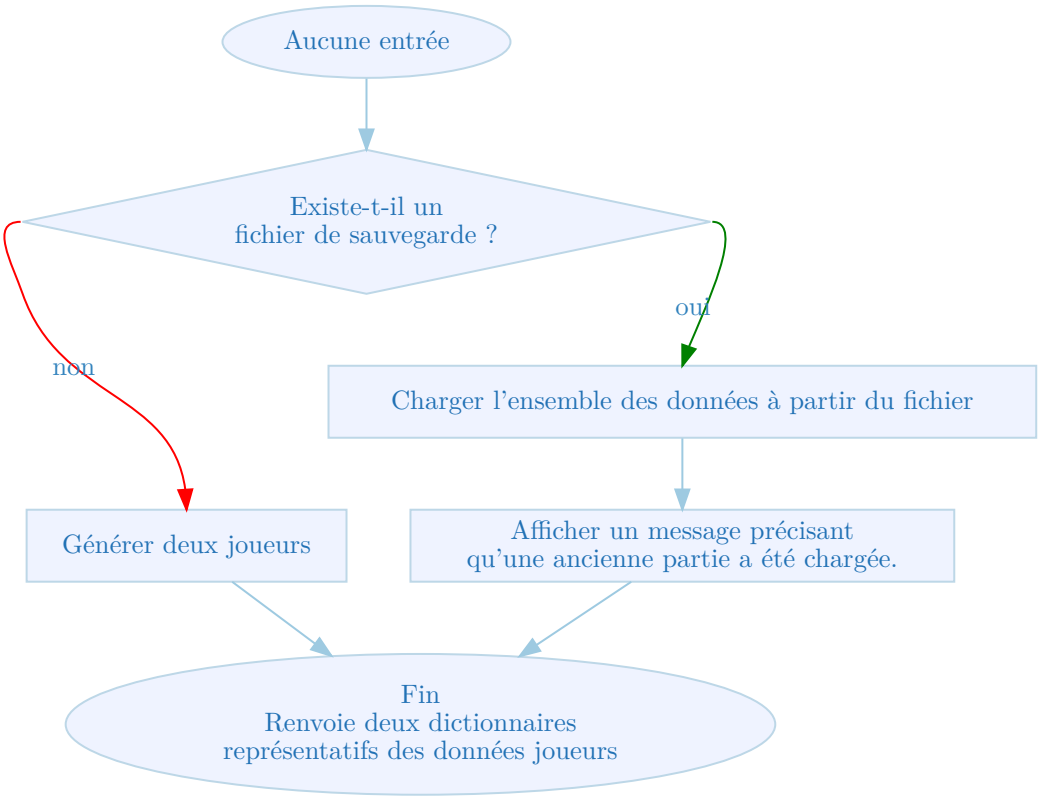
```
def Sauvegarde(dico_joueur, dico_ordi):
    dossier_projet = os.getcwd()
    if not os.path.exists(______):
        os.______(______)
    data = [dico_joueur, dico_ordi]
    fichier = open(______, ______)
    json.dump(______, ______)
    fichier.close()
```

Testez votre code sur votre ordinateur, puis effectuez la validation sur Moodle.

Pensez à commenter/docstringuer votre code avant de le soumettre !

Partie 2 - Chargement

Qui dit "sauvegarde" dit "chargement". Vous allez implémenter la fonction **Chargement()**, basée sur l'algorigramme suivant. Vous penserez à utiliser les fonctions déjà développées lors des TP précédents.



Testez votre code sur votre ordinateur, puis effectuez la validation sur Moodle.

Pensez à commenter/docstringuer votre code avant de le soumettre !

Partie 3.1 - Saisie_Coods() - v2

Le joueur doit pouvoir indiquer qu'il souhaite sauvegarder sa partie. Le seul moment où il intervient c'est lorsqu'il doit saisir les coordonnées de tir.

Modifier la fonction Saisie_Coords() afin qu'elle accepte et renvoie la lettre 'Q'

Testez votre code sur votre ordinateur, puis effectuez la validation sur Moodle.

Pensez à commenter/docstringuer votre code avant de le soumettre !

Partie 3.2 - Tir() - v2

En TP3 nous avons modifié les structures de données des bateaux, pour passer d'une liste à un dictionnaire. Cela va donc avoir des implications sur les fonctions qui utilisent les bateaux.

Voici par exemple la fonction gagne() après modification, pour tenir compte de cette nouvelle structure de données :

```
def Gagne(bateaux):
    '''
    Détermine si tous les bateaux ont été coulés
    '''
    for bateau in bateaux:
        if bateau['taille']!=bateau['touché']: # bateau non coulé ? (taille différent de
            return False
    return True
```

Modifiez la fonction Tir(), sachant que maintenant un bateau n'est plus une liste mais un dictionnaire.

Autre modification à effectuer : la validation précédente oubliait un test important, et de ce fait validait à tord certaines fonctions incomplètes : on nous disait qu'en cas de tir dans l'eau, elle doit renvoyer 0. La validation oubliait le cas où l'eau vaut -1 (raté). Cette nouvelle validation en tient compte !

Testez votre code sur votre ordinateur, puis effectuez la validation sur Moodle.

Pensez à commenter/docstringuer votre code avant de le soumettre !

Partie 3.3 - Boucle_Jeu() - v2

Les modifications du TP3 et de ce TP nécessitent d'adapter la fonction **Boucle_Jeu()**.

- La fonction ne prend plus aucun argument d'entrée.
- À la place, elle fait appel dès le début à la fonction Chargement() afin de récupérer les données des deux joueurs (l'humain et l'ordinateur).
- À chaque fois que vous accédiez à une grille ou à des bateaux, il faut maintenant effectuer ces accès à partir du dictionnaire du joueur adéaquat.
- Ne pas oublier que depuis le TP précédent, chaque bateau est un dictionnaire.
- Lorsque la saisie du joueur humain vaut 'Q', il faut sauvegarder la partie en cours, et stopper la boucle de jeu.

Testez votre code sur votre ordinateur, puis effectuez la validation sur Moodle.

Pensez à commenter/docstringuer votre code avant de le soumettre !

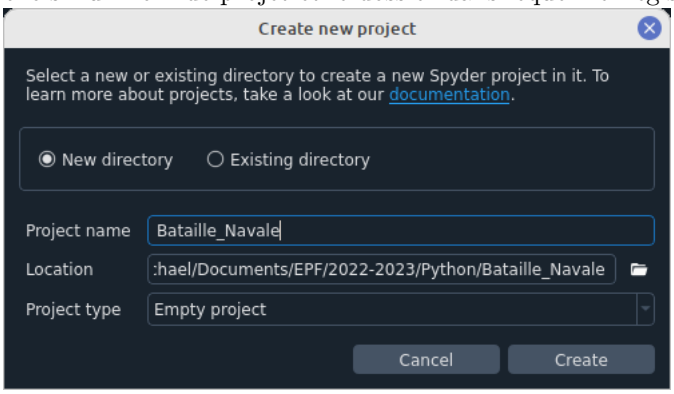
5 TP5

Partie 1 - Le projet

Lisez toute la partie 1 avant d'effectuer les manipulations.

Jusqu'à présent, vous avez géré vos scripts python chacun à votre manière, afin d'implémenter et tester vos différentes fonctions.
Dans Spyder nous allons maintenant créer un projet, afin de bien organiser le travail.

- Menu Projects > New Project...
- choisir un nom de projet et le dossier dans lequel l'enregistrer :



Dans ce dossier copier/collez le fichier TP5.py fourni sur Moodle, ainsi que le fichier Corrige.pyc

- puis vous allez créer des scripts/modules dans votre projet :
  - **CLI.py** qui va contenir toutes les fonctions de saisie et d'affichage (CLI signifie : Command Line Interface)
  - **Partie.py** qui va contenir les fonctions de génération/chargement/sauvegarde
  - **BNlib.py** qui va contenir les fonctions utilitaires du jeu (c'est à dire les "petites" fonctions qui ne sont pas dans les modules précédents)
- **Jeu.py** qui va contenir les fonctions majeures du Jeu, ainsi qu'un appel à la fonction $Boucle_Jeu()$

Débutez TOUS vos scripts par un import du corrigé, ainsi même s'il vous manque des fonctions, vous aurez un jeu fonctionnel :

```
from Corrige import *
```

Puis copiez/collez dans les différent modules les fonctions que vous avez développées (uniquement celles pour lesquelles vous avez eu 10/10 lors des validations Moodle). Voici également deux nouvelles fonctions à ajouter/modifier :

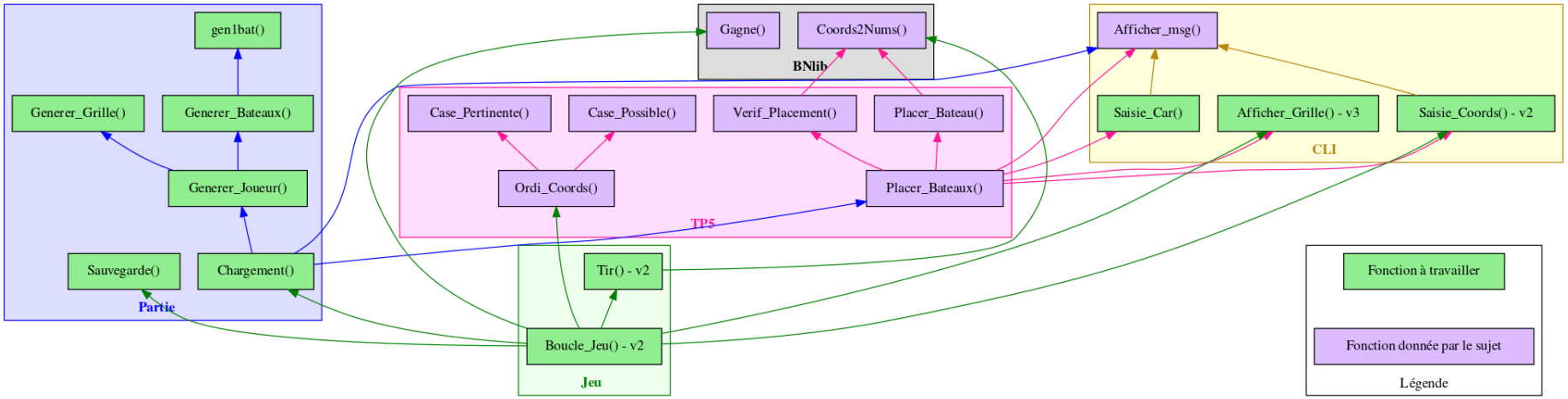
```
def Saisie_Car(msg, choix):
    '''
    Restreint la saisie d'un caractère parmis ceux proposé par "choix"
    Le renvoie en majuscule quand ok
    '''
    Afficher_msg(msg+" "+choix)
    saisie = input().upper()
    while len(saisie)!=1 and saisie not in choix:
        Afficher_msg("Erreur, "+msg+" "+choix)
        saisie = input().upper()
    return saisie
```

Une nouvelle version de Generer_Joueur() :

```
def Generer_Joueur(humain):
    '''
    Génère les informations de jeu d'un joueur
    Entrée : booléen qui spécifie si l'on crée le joueur humain ou l'ordi
    Sortie : un dictionnaire
    '''
    batx = Generer_Bateaux()
    return {"grille": Placer_Bateaux(Generer_Grille(10), batx, humain), \
        "bateaux" : batx, \
        "tirs": Generer_Grille(10), \
        "score": 0
    }
```

–> il faudra donc modifier la fonction de Chargement() pour qu'elle génère un joueur humain, puis un joueur ordi.

Et enfin une représentation visuelle de l'agencement souhaité des fonctions (voir page suivante) :


gen1bat()
Generer_Grille()
Generer_Bateaux()
Generer_Joueur()
Sauvegarde()
Chargement()
Partie
Gagne()
Coords2Nums()
BNlib
Case_Pertinente()
Case_Possible()
Verif_Placement()
Placer_Bateau()
Ordi_Coords()
Placer_Bateaux()
TP5
Tir() - v2
Boucle_Jeu() - v2
Jeu
Afficher_msg()
Saisie_Car()
Afficher_Grille() - v3
Saisie_Coords() - v2
CLI
Fonction à travailler
Fonction donnée par le sujet
Légende

Partie 2 - Débogage

Le fichier TP5.py contient les fonctionnalités permettant :

- de placer les bateaux
- le tir statégique de l'ordinateur

Mais il est boggué ! Vous devez trouver les 13 bugs de ce module. Pour cela utilisez les fonctionnalités de Spyder (détections d'erreur de syntaxe par l'éditeur, débogueur intégré pour avancer pas à pas).

Vous avez à trouver et corriger (pas forcément dans cet ordre !):

- Faciles :
  - 6 erreurs de syntaxe
  - 2 crashs simples (l'erreur est sur la ligne concernée)
- Moyens :
  - 2 crash moyennement difficile (l'erreur est sur une autre ligne que le moment du crash –> la conséquence de l'erreur survient plus tard dans le déroulement de l'algorithme)
- Difficiles :
  - 1 fonction qui joue mal son rôle de façon silencieuse (observez attentivement la console)
  - 1 boucle infinie
  - 1 crash qui ne se produit jamais dans le cadre des tests du TP5.

Ayez du bon sens ! Observez ! Prenez des notes pendant la séances sur la manière dont vous résolvez chaque bug.

Travail à rendre : vous choisirez un bug de la catégorie "moyennes à difficiles", et vous ferez une vidéo expliquant comment vous avez débusqué le bug et comment l'utilisation du débogueur intégré vous a aidé à le corriger. (Si vous avez pu le débusquer/corriger sans faire appel au débogueur, alors il vous faudra tout de même montrer comment un usage pertinent du débogueur permet de voir la cause de l'erreur). Plus d'informations concernant ce rendu vous seront fournies sur Moodle.

Dernière modification : l'ancienne fonction Ordi_Coords() prenait un seul argument d'entrée ; mais la nouvelle données dans le TP5 en prend 2. Modifiez la Boucle_Jeu() en ce sens.

Félicitations, vous avez un jeu 100% fonctionnel ! (il suffit de lancer le module principal Jeu.py)

Ci-après, quelques explications sur le fonctionnement d'une partie des fonctions données.

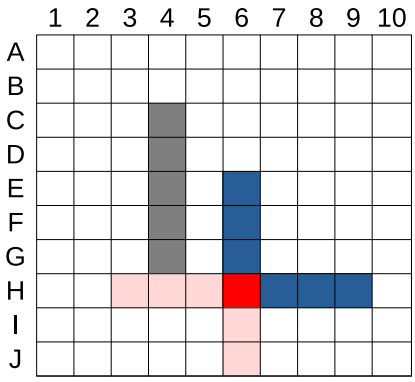
Placer les bateaux :

À partir de la première case d'un bateau, il va falloir vérifier toutes les orientations possibles, en tenant compte des bords de la grille et des bateaux déjà placés (sachant que les bateaux ne doivent jamais se toucher).

Voici une explication du fonctionnement de fonction **Verif_Placement()**

Elle prend en entrée - dans cet ordre - une grille, une coordonnée de départ et une taille de bateau ; et renvoie en sortie une chaine de caractères indiquant les directions possibles parmi N, E, S, O.

Représentation du problème :



- En gris le porte-avion est déjà placé, et on souhaite placer le croiseur.
- Le joueur a choisi la case H6 comme point de départ (rouge vif).
- À l'ouest on ne peut pas placer le bateau car il touche le porte-avion
- Au sud, on sort de la grille.
- La fonction va donc renvoyer les orientations possibles (en bleu), Nord et Est, soit la chaîne "NE" (peu importe l'ordre).

Stratégie de tir :

La stratégie de tir actuelle comprend 3 grandes situations :

```
Recherche d'un bateau déjà touché mais non coulé, si trouvé :
    L'as-t-on déjà touché sur deux case adjacentes ? Si oui :
        alors je connais son orientation, je recherche
        la première case non tirée aux extrémités du bateau.
    Sinon
        je choisi au hasard l'une des 4 cases orthogonales qui n'aurait pas déjà été tir
Sinon
    Tir aléatoire :
    - toutefois non adjacent à un bateau
    - et si orthogonalement je n'ai pas déjà tiré les deux cases autour de la case ciblé
      (ce serait inutile car il n'existe pas de bateau de 1 case)
```

Pour tous les traitements sur des "morceaux" de grille, il faut systématiquement penser à vérifier qu'on ne tente pas d'accéder à des cases inexistantes (sortie de grille)

À noter qu'il est encore possible d'améliorer cette stratégie, par exemple en effectuant les tirs aléatoires espacés les uns des autres par au maximum la longueur du plus grand bateau non coulé.

TP 6 (Partie 1): Interface Graphique avec PySimpleGUI

Ce TP a pour but de vous initier au développement d'une interface graphique (GUI : Graphical User Interface).

PySimpleGUI est une boite à outils permettant de gérer une interface graphique sous Python de manière relativement simple (comparativement à d'autres boites à outils graphiques).

6 Installation

Dans Anaconda, Suivre les étapes suivantes pour installer PySimpleGUI :

1. Accédez à l'onglet Environnements.
2. Cliquez sur le bouton Channels (Canaux).
3. Cliquez sur le bouton Add (Ajouter) et saisissez l'URL du canal : https://conda.anaconda.org/conda-forge/
   Appuyez sur la touche Entrée de votre clavier.
4. Cliquez sur le bouton Update Channels (Mettre à jour les canaux).
5. Cliquez sur le bouton Update indexes (Mettre à jour les index)
6. Dans le champ de filtrage en haut à gauche, passez de "installed" à "Not installed" (car on ne souhaite voir que les bibliothèques pas encore installées !)
7. Dans le champ de recherche en haut à droite, cherchez PySimpleGUI
8. Vérifiez qu'il est installé (déjà coché), ou le cas échéant installez-le (cocher, appliquer)

Source pour davantage informations : https://conda-forge.org/docs/user/introduction.html

Problème occasionnel : malgré les bonnes manipulations, chez certains PySimpleGUI n'apparaît pas dans la liste. Dans ce cas une désintallation complète de Anaconda suivie d'une réinstallation résoud le problème.

7 Utilisation

Comme pour tout module python, si l'on souhaite l'utiliser il faut l'importer dans le script courant :

```
import PySimpleGui
```

Mais vous vous souvenez certainement qu'à l'usage il faudra systématiquement précéder les fonctions par le nom du module :

```
# exemples d'appels :
PySimpleGui.fonction1()
PySimpleGui.fonction2()
...
```

On va donc renommer le module avec une version courte, pour gagner du temps :

```
import PySimpleGUI as sg  # le module s'appelle maintenant 'sg'

# exemples d'appels :
sg.fonction1()
sg.fonction2()
...
```

8 Première interface graphique

8.1 Code

Copiez/collez le code ci-dessous dans un script python (et prenez le temps de lire les commentaires !):

```
import PySimpleGUI as sg

# Déclaration du 'layout' (mise en page de la fenêtre)
layout=[[sg.Button('Cliquez ici')]]

# Création de la fenêtre
window=sg.Window('Ma première fenêtre', layout=layout)

# Boucle de gestion des évènements
while True:
    event, values = window.read() # lecture du dernier évènement
    if event == sg.WIN_CLOSED: # si l'évènement est "quitter", on casse la boucle
        break

window.close() # Destruction de la fenêtre
```

Testez le code. Vous avez beau cliquer sur le bouton il ne se passe rien. En revanche il est possible de fermer la fenêtre.

8.2 Comprendre les évènements

Un évènement est une action produite par l'utilisateur. Sur le code ci-dessus l'évènement "WIN_CLOSED" (quitter la fenêtre) est géré par notre code. En revanche on ne gère pas l'évènement "cliquer sur un bouton".

Après la ligne "window.read()", ajoutez cette ligne :

```
print(event, values)
```

Testez votre code. Vous constatez que l'évènement correspondant au clic s'appelle comme le texte du bouton. Dans la boucle, après la gestion de l'évènement "quitter" ajoutez le code suivant :

```
elif event == "Cliquez ici":
    print("Bravo !")
```

Testez votre code.

Et ensuite...

Ajoutons un bouton, modifiez le 'layout' en :

```
layout=[[sg.Button('Cliquez ici'), sg.Button("Cliquez ici")]]
```

On a donc deux boutons avec le même texte dessus. Comment PySimpleGUI gère ce cas au niveau des évènements ? Testez le code.

Donc PySimpleGUI nomme automatiquement le second évènement avec un numéro pour le différencier du premier bouton. C'est pratique en apparence, mais cela peut devenir compliqué à gérer :

- si j'ai plein de boutons avec le même texte, je vais me perdre avec une simple numérotation automatique.
- si je veux changer le texte d'un bouton, il faudra changer également la gestion de l'évènement : deux portions de code à changer.

Pour résoudre ce problème on va identifier nos boutons par une clé, modifiez le layout ainsi :

```
layout=[[sg.Button('Cliquez ici', key='BTNgauche'), sg.Button("Cliquez ici", key="BTNdro
```

Testez votre code.

Ajustez la gestion des évènements pour afficher deux messages différents selon que l'utilisateur clique l'un ou l'autre des boutons.

8.3 le 'layout'

Comme vous le constatez la mise en page correspond à un tableau de tableaux :

- Le tableau englobant correspond à la fenêtre.
- le tableau intérieur correspond à une ligne.
- et dans notre exemple, cette ligne contient deux boutons en colonne.

Positionner les éléments dans une fenêtre correspond donc à déterminer dans quel tableau on doit le placer. Modifiez le layout ainsi (exemple tiré de la documentation de PySimpleGUI, et modifié) :

```
layout = [
            [sg.Text('1. '), sg.Input(key='i1')],
            [sg.Text('2. '), sg.Input(key='i2')],
            [sg.Text('3. '), sg.Input(key='i3')],
            [sg.Text('4. '), sg.Input(key='i4')],
            [sg.Text('5. '), sg.Input(key='i5')],
            [sg.Text(' ', key="MonTexte")],
            [sg.Button('Afficher'), sg.Button('Quitter')]
        ]
```

Vous découvrez deux nouveaux éléments en plus du bouton : Text (pour écrire un simple texte) et Input (pour afficher une boite de saisie utilisateur).

Testez le code.

8.4 Évènements (suite) et premier retour visuel

En testant le code ci-dessus, vous avez dû constater que lors de chaque évènement l'ensemble des valeurs des éléments 'Input' avec leur clé sont retournés dans la variable 'values' sous forme de **dictionnaire**.

On va donc pouvoir effectuer différents traitements avec ces valeurs, et notamment ce qu'on appelle un **retour visuel** : c'est à dire que l'interface est modifiée en fonction des actions de l'utilisateur.

Par exemple, si je veux qu'un clic sur "Afficher" montre le contenu de la boite de saisie n°3, on va ajouter dans la boucle de gestion évènementielle :

```
    elif event == "Afficher": # bouton 'Afficher'
        window['MonTexte'].update(values['i3']) # Modification de l'élément "MonTexte"
```

Testez votre code, bien évidemment pensez à remplir le champ input n°3.

Cela semble fonctionner, mais on ne voit que le premier caractère. C'est parce que notre élément de texte est trop petit pour montrer tout son nouveau contenu. En fait, dans le layout il fallait prévoir un espace plus grand. Modifiez la ligne ainsi :

```
    [sg.Text(' ', key="MonTexte", size=(50,1))], # 50 caractères max sur 1 ligne
```

Remarque : avec PySimpleGUI, l'unité de taille des éléments se fait en nombre de caractères (horizontaux et verticaux).

8.5 Récapitulatif des concepts importants à maîtriser

- Le 'layout' est un tableau (fenêtre) de tableaux (lignes) d'élements.
- Chaque élément peut avoir une clé, passée grâce à l'argument **key='nom_de_la_clé'**
- La fenêtre est créée par un appel à la fonction **Window()** (du module PySimpleGUI); le premier argument de cette fonction correspond au titre affiché sur la fenêtre, l'argument **layout=** doit contenir un 'layout' valable (tableau de tableaux d'éléments). Cette fonction renvoie un objet dictionnaire dont tous les élements créés sont accessibles via leur clé.
- Les évènements sont récupérés via un appel à la fonction **read()** de l'objet fenêtre créé précédemment. Cette fonction renvoie deux arguments : la clé de l'élément qui a causé l'évènement, ainsi qu'un dictionnaire des valeurs des élements dynamiques.

Remarque : on parle ici d'objet, vous n'avez pas besoin de comprendre ce que cela signifie, ce sera abordé en seconde année. Vous avez déjà régulièrment utilisé cette notation particulière lorsqu'une fonction n'existe que pour certaines variables :

- var_chaine.split()
- var_fichier.read()
- var_liste.append()
- etc
- et donc maintenant : var_fenetre.read()

8.6 Exercice

1. Nettoyez la boucle évènementielle des évènements qui ne sont plus utiles pour le code actuel.
2. En bas de fenêtre, ajoutez des boutons pour en avoir 6.
3. Les 5 premiers affichent l'input correspondant.
4. Le dernier sert à quitter (en plus du clic sur la croix)

9 Les éléments

9.1 Les réglages fins via les arguments

La plupart des éléments peuvent être finement configurés, notament pour modifier leur aspect :

key='...'	Définir la clé pour cet élément.
size=(largeur, hauteur)	Imposer la taille pour cet élément, les unités sont en nombre de cacactères.
border_width=(largeur)	Définir l'épaisseur de bordure, l'unité est le pixel.
pad=(horizontal, vertical)	Définir la taille des marges autour de l'élément. L'unité est le pixel.

9.2 Button

```
sg.Button('texte du bouton', ...)
```

Il est recommandé de prévoir une clé pour cet élément.

Il est également possible d'utiliser l'argument **button_color=(avant, arrière)** pour définir la couleur du texte (avant) et du fond (arrière). Les couleurs sont représentées par des chaîne de caractères :

- soit par leur nom en anglais ('blue', 'red', ...)
- soit par leur code couleur en hexadécimal '#RRGGBB' où RR, GG et BB sont respectivment les valeurs du rouge, du vert et du bleu en synthèse additive, sur 8 bits. Exemple '#00D99E' est un vert turquoise (pas de rouge, et plus de vert que de bleu (D9 > 9E))

9.3 Texte

```
sg.Text('Texte à afficher', ...)
```

Si le texte doit être modifié par le code, penser à définir une taille suffisante pour afficher le futur texte.

9.4 Boîte à cocher

```
sg.Checkbox('Texte à afficher', ...)
```

Cet élément ne génère pas d'évènement, c'est normal : lorsqu'on l'on coche ou décoche une boite, on souhaite généralement prendre en compte la nouvelle valeur lors d'un clic sur un bouton de validation (ou ignorer sur un bouton d'annulation). Il est toutefois possible de forcer un évènement au clic avec l'attribut **enable_event=True**. En cas de forçage d'évènement il est recommandé de prévoir une clé pour cet élément.

9.5 Choix multiples

```
sg.Radio('Texte à afficher', 'Groupe d'appartenance', ...)
```

Les éléments à choix multiple servent à faire une sélection parmi des choix mutuellement exclusifs. Les boutons radio doivent donc appartenir à un groupe : dès que l'utilisateur clique sur un des choix, les autres choix du même groupe sont désélectionnés.

Cet élément ne génère pas d'évènement, c'est normal : lorsqu'on l'on coche un bouton radio, on souhaite généralement prendre en compte la nouvelle valeur lors d'un clic sur un bouton de validation (ou ignorer sur un bouton d'annulation). Il est toutefois possible de forcer un évènement au clic avec l'attribut **enable_event=True**. En cas de forçage d'évènement il est recommandé de prévoir une clé pour cet élément.

9.6 Colonnes et "Frame"

```
sg.Column([[...]])
sg.Frame('Texte à afficher', [[]])
```

Rappel : la mise en page est conçue comme un tableau de lignes (tableaux) contenant des élements. Dans certains cas on peut vouloir affiner la présentation, il sera alors possible d'utiliser l'élément Column : Cet élement se subdivise à son tour en un tableau de lignes (tableaux) d'éléments.

Frame est comme un Column, avec une bordure et un titre visible en plus.

Vous avez un exemple d'utilisation de colonne ici : https://pysimplegui.readthedocs.io/en/latest/cookbook/recipe-multiple-columns

9.7 Tous les élements possibles

La liste de tous les éléments possible est disponible dans la documentation de PysimpleGUI :

https://pysimplegui.readthedocs.io/en/latest/cookbook/recipe-nearly-all-elements-with-color-theme-menus-the-everything-bagel

9.8 Exercice

Maintenant, nous pouvons améliorer encore notre projet Bataille Navale avec une interface graphique (Partie 2 du TP 6 et le TP7) !