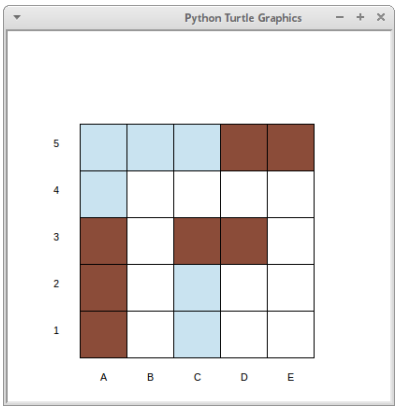
Travaux Pratiques
Initiation à la programmation avec Python
Feuille n.5
Jeu de la bataille navale

1 Définition du jeu

Wikipedia [1] : 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 torpiller complètement les navires de l'adversaire avant que tous les siens ne le soient. **Règles** : La bataille navale oppose deux joueurs qui s'affrontent. Chacun a une flotte composée de 5 bateaux, qui sont, en général, les suivants : 1 porte-avion (5 cases), 1 croiseur (4 cases), 1 contre torpilleur (3 cases), 1 sous-marin (3 cases), 1 torpilleur (2 cases). Au début du jeu, chaque joueur place ses bateaux sur sa grille. Celle-ci est toujours numérotée de A à J verticalement et de 1 à 10 horizontalement. Un à un, les joueurs vont "tirer" sur une case de l'adversaire. Par exemple B3 ou encore H8. Le but est donc de couler les bateaux adverses.

2 Cahier des charges

Écrire un jeu de bataille navale où seul l'utilisateur (humain) doit deviner la grille cachée par l'ordinateur. On simplifiera aussi, au moins dans un premier temps, le nombre et la taille des navires. Exemple d'exécution du programme `bataille_navale.py` à écrire :




```
$ ipython
import turtle
import bataille_navale
bataille_navale.jouer()
Coordonées (ex: C3)? A5
Coordonées (ex: C3)? A4
Coordonées (ex: C3)? A3
Coordonées (ex: C3)? A2
Coordonées (ex: C3)? A1
Navire de taille 3 coulé!
Coordonées (ex: C3)? B5
Coordonées (ex: C3)? C1
Coordonées (ex: C3)? C2
Coordonées (ex: C3)? C3
Coordonées (ex: C3)? D3
Navire de taille 2 coulé!
Coordonées (ex: C3)? C5
Coordonées (ex: C3)? D5
Coordonées (ex: C3)? E5
Navire de taille 2 coulé!
Bravo, vous avez coulé tous les navires!
```

1. `https://fr.wikipedia.org/wiki/Bataille_navale_(jeu)`
(Texte sous Licence Creative Commons paternité partage à l'identique)

3 Guide de développement du jeu

Vous pouvez ignorer ce guide et développer le jeu à votre façon, si vous pensez pouvoir le faire en autonomie. La fonction principale du programme `jouer()` peut être ou peut ressembler à la suivante :

```
1.  def jouer():
2.    # ---
3.    entetes_colonnes = "ABCDE"            # on pourra les changer, en rajouter, les renverser, etc
4.    entetes_lignes   = "12345"            # idem
5.    origine          = (-200,-200)        # coordonnées "turtle" de l'angle en bas à gauche du carré A1
6.    taille           = 50                 # taille en pixels d'une case (côté du carré)
7.    # ---
8.    traducteur_positions = fabrique_traducteur_positions(origine, taille, entetes_colonnes, entetes_lignes)
9.    (navires, grille)    = fabrique_navires_et_grille(entetes_colonnes, entetes_lignes)
10.   # ---
11.   nb_colonnes = len(entetes_colonnes)   # nombre de colonnes
12.   nb_lignes   = len(entetes_lignes)     # nombre de lignes
13.   # ---
14.   trace_grille_carres(origine, taille, False, "white", nb_lignes, nb_colonnes)
15.   trace_entetes_colonnes(origine, taille, entetes_colonnes)
16.   trace_entetes_lignes(origine, taille, entetes_lignes)
17.   # ---
18.   devinees = {}                         # dictionnaire des cases devinées
19.   navires_en_vie = navires              # liste des navires pas encore coulés
20.   while navires_en_vie!= [] :
21.     (X,Y) = saisie_coordonnees_joueur(entetes_colonnes, entetes_lignes)
22.     pos = traducteur_positions[(X,Y)]              # traduit en position "tortue"
23.     if grille[(X,Y)]:                              # grille est un dictionnaires à valeurs booléennes
24.       trace_carre(pos, taille, True, "salmon4")    # couleur des navires
25.       devinees[(X,Y)] = True                       # la case (X,Y) est devinée
26.       navires_en_vie = annoncer_si_navire_vient_de_couler(navires_en_vie, devinees)
27.     else:
28.       trace_carre(pos, taille, True, "azure")      # couleur de l'eau
29.   # ---
30.   print "Bravo, vous avez coulé tous les navires!"
```

Idées principales

1. Il y a des coordonnées (ou positions) "joueur", comme `('A','1')`, et des coordonnées "tortue", comme `(-200,-200)`. Les deux sont des **couples** (n-uplets de 2 éléments). Un simple **dictionnaire** (`traducteur_position`) nous permettra de traduire une coordonnée "joueur" en coordonnée "tortue" .
2. Un navire est représenté par une **liste** de positions "joueur". Par exemple, dans le schéma de jeu précédent, le navire de trois cases est représenté par la liste `[('A','1'), ('A','2'), ('A','3')]`.
3. La grille de jeu, où sont positionnés les navires, est représentée par un dictionnaire (`grille`) associant à toute position "joueur" un booléen représentant la **présence** (`True`) ou l'**absence** (`False`) d'une case de navire.

Fonctions principales

On commence par coder le premier effet observable par l'utilisateur (en début de la boucle, ligne 21).

(3.1) `def saisie_coordonnees_joueur(entetes_colonnes, entetes_lignes):`

Cette fonction demande à l'utilisateur une chaîne de caractère **en boucle**, tant que cette chaîne ne correspond pas à une coordonnée **valide**, par exemple `"C3"`. Pour être valide, il faut que la chaîne soit constituée de 2 caractères, le premier doit appartenir à la chaîne `entetes_colonnes`, et le second à la chaîne `entetes_lignes`, qui sont données en argument. La fonction renvoie le couple des deux caractères choisis (d'où l'affectation en retour à la ligne 22).

Une fois saisie, les coordonnées "joueur" (comme `('A','1')`) sont traduites en coordonées "tortue" (comme `(-200,-200)`) à la ligne 23 par le dictionnaire `traducteur_position` qui est construit une fois pour toutes à la ligne 8, en faisant appel et en récupérant le résultat fourni par la fonction suivante.

(3.2) `def fabrique_traducteur_positions(origine, taille, entetes_colonnes, entetes_lignes):`

Étant donné une coordonnée joueur `(X,Y)` on appellera coordonnées "matricielle" le couple `(i,j)` des **indices** respectivement de `X` dans `entetes_colonnes` et de `Y` dans et `entetes_lignes`. Par exemple, `('A','4')` correspondra aux coordonnées matricielles `(0,3)` (en supposant `entetes_colonnes="ABCDE"` et `entetes_lignes="12345"`). Si on appelle `origine = (x0,y0)` la coordonnée "tortue" (écran graphique) de l'**angle bas à gauche** de la case `('A','1')`, on constate que :

— la coordonnée (clef) `('A','1')` (matricielle `(0, 0)`) doit être associé à `(x0, y0)`
— la coordonnée (clef) `('A','2')` (matricielle `(0, 1)`) doit être associé à `(x0, y0+taille)`
— la coordonnée (clef) `('B','3')` (matricielle `(1, 2)`) doit être associé à `(x0+taille, y0+2*taille)`
— pigé ?

À la ligne 23 se trouve le test permettant de savoir si le coup du joueur est bon (navire touché, peut-être même coulé), ou s'il est mauvais (dans l'eau) grâce au dictionnaire `grille`. Un second dictionnaire, `devinees`, enregistre les **cases devinées** par l'utilisateur de façon à pouvoir déduire qu'un navire a été coulé. Le dictionnaire `grille` est construit en même temps que la liste des navires (l'information est la même !) à la ligne 9. Chaque navire étant une liste de coordonnées "joueur", les **navires** seront une **liste de listes de couples**. Les deux structures sont construites à la ligne 9, en faisant appel et en récupérant le résultat fourni par la fonction suivante.

(3.3) `def fabrique_navires_et_grille(entetes_colonnes, entetes_lignes):`

Théoriquement, la position des navires devrait être générée de façon aléatoire. Dans une première version, nous nous contenterons d'un positionnement fixé dans notre code. L'important est que les deux structures soient **cohérentes**. Pour cela, vous devrez en définir une, et calculer l'autre à partir de la première. Par exemple, si vous définissez les navires de cette façon : `navires = [ [('A','1'), ('A','2'), ('A','3')], [('C','3'), ('D','3')], [('D','5'), ('E','5')]]`, vous devrez définir le dictionnaire `grille` de façon que `grille[(X,Y)]` soit `True` si et seulement si `(X,Y)` apparaît dans une des listes précédentes. (**Suggestion** : calculer la grille à partir des navires). La fonction doit retourner le couple `(navires, grille)`.

Un outil fondamental est celui qui permet de mettre à jour l'ensemble des navires "en vie", c'est-à-dire non encore coulés. C'est le rôle de la fonction suivante.

(3.4) `def annoncer_si_navire_vient_de_couler(navires, devinees):`

Cette fonction a un objectif **double**, celui d'indiquer (par un `print`) à l'utilisateur qu'un navire a été coulé et, dans ce cas, d'éliminer ce navire de la liste. La liste actualisée des navires encore en vie sera retournée à l'appelant (permettant ainsi la mise à jour de la variable correspondante, ligne 26).

Dessiner la grille case par case

Il est avant tout intéressant d'avoir un outil de déplacement "propre" de la tortue, c'est-à-dire un équivalent de la fonction prédéfinie `turtle.goto()`, mais qui ne laisse aucune trace du déplacement. Cet outil peut être programmé de la manière suivante :

```
def deplace_proprement(pos):
  turtle.up()
  turtle.goto(pos)
  turtle.down()
```

Aussi, pour rendre la tortue plus **rapide** et **invisible**, vous pouvez rajouter à votre programme les instructions suivantes :

```
turtle.speed(10)
turtle.hideturtle()
```

Pendant le jeu, qu'un navire soit touché (lignes 24-26) ou pas (ligne 28), un **carré colorié** devra être tracé sur l'écran graphique à une certaine position "tortue". Cependant, en début du jeu, il faudra aussi avoir dessiné la grille, qu'on peut concevoir comme une composition de carrés cette fois **vides** (non coloriés). Dans cette approche, il nous faudra un outil générique permettant de tracer des carrés coloriés ou pas.

(3.5) `def trace_carre(pos, taille, plein, couleur):`

Le paramètre `plein` est un booléen qui indique si le carré doit être rempli (colorié) ou pas. Le paramètre couleur est utilisé seulement si `plein` est égal à `True`. Utilisez les fonctions `turtle.fillcolor()`, `turtle.begin_fill()` et `turtle.end_fill()` pour cet éventuel remplissage.

Une fois `trace_carre()` disponible, on peut plus facilement définir les fonctions suivantes.

(3.6) `def trace_barre_horizontale(pos, taille, plein, couleur, nb_colonnes):`

Trace une barre horizontale de `nb_colonnes` carrés contigus de même `taille`, remplis éventuellement (`plein`) de la même `couleur`.

(3.7) `def trace_barre_verticale(pos, taille, plein, couleur, nb_lignes):`

Trace une barre verticale de `nb_lignes` carrés contigus de même `taille`, remplis éventuellement (`plein`) de la même `couleur`.

(3.8) `def trace_grille_carres(pos, taille, plein, couleur, nb_lignes, nb_colonnes)`

Trace une grille de `nb_colonnes` $\times$ `nb_lignes` carrés contigus de même `taille`, remplis éventuellement (`plein`) de la même `couleur`. On pourra soit utiliser une boucle d'appels à `trace_barre_horizontale()` (la grille comme *superposition* de barres horizontales), soit une boucle d'appels à `trace_barre_verticales()` (la grille comme *juxtaposition* de barres verticales).

Dessiner les en-têtes

On peut imaginer que les caractères des en-têtes soient écrit à l'intérieur de carrés **invisibles** situés autour des carrés (visibles) de la grille, comme si la grille avait une **ligne** de carrés supplémentaire (en bas) invisibles pour accueillir les en-têtes de colonnes (par exemple `"ABCDE"`), et une colonne de carrés supplémentaires (à gauche) invisibles pour accueillir les en-têtes de lignes (par exemple `"12345"`). Dans cette approche, on définira les fonctions suivantes :

(3.9) `def ecrire_dans_carre(pos, taille, caractere):`

On utilisera la fonction `turtle.write()`.

(3.10) `def trace_entetes_colonnes(pos, taille, entetes_colonnes):`

Similaire à `trace_barre_horizontale()`, on utilisera la fonction `ecrire_dans_carre()`.

(3.11) `def trace_entetes_lignes(pos, taille, entetes_lignes):`

Similaire à `trace_barre_verticale()`, on utilisera la fonction `ecrire_dans_carre()`.

Tout devrait être prêt, à présent, pour jouer et profiter d'un petit moment de détente, après avoir tant bossé...