SAE 1.01 : Implémentation d'un besoin client

Projet: Puissance 4 en Python

Réalisé par TRAN Minh-Tue - Groupe B2 (Binôme BB21)

Version 1

Explication des fonctions, des variables

Variable globale

```
COLUMNS = 7
ROWS = 6
```

COLUMNS: le nombre de colonne de la grille
ROWS: le nombre de ligne de la grille

Fonction create_grid()

```
def create_grid() -> list[list[str]]:
    return [[" " for _ in range(COLUMNS)] for _ in range(ROWS)]
```

- Créer une grille de 6 lignes et 7 colonnes

- En utilisant **for … in range**, la fonction retourne une liste de sous liste (type **string**)
- Voici le résultat

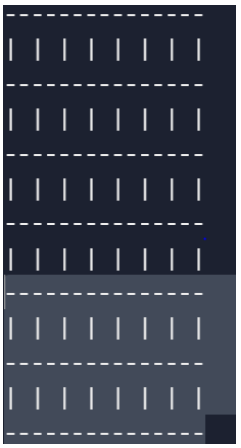
```
[[' ', ' ', ' ', ' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ', ' ', ' ',
    ' '], [' ', ' ', ' ', ' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ',
    ' ', ' ', ' '], [' ', ' ', ' ', ' ', ' ', ' ', ' '], [' ', ' ',
    ' ', ' ', ' ', ' ', ' ']]
```

- On peut voir une **liste** de **longueur 6 (~lignes)** et une **sous liste** de **longueur 7 (~ colonnes)**

Fonction print_grid(grid: list[list[str]])

```
7   def print_grid(grid: list[list[str]]):
8       print('-'*15)
9       print(grid)
10      for row in grid:
11          print("|" + "|".join(row) + "|")
12          print("-" * 15)
13
```

- Afficher la grille selon le forme du sujet.
- Paramètre **grid** est une liste de sous-listes de **str**
- Voici le résultat:



Fonction check_victory(grid: list[list[str]]), piece: str):

```python
# Fonction pour vérifier si un joueur a gagné
def check_victory(grid: list[list[str]], piece: str)-> bool:
    # Vérification des lignes
    for row in grid:
        if f"{piece * 4}" in "".join(row):
            return True

    # Vérification des colonnes
    for col in range(len(grid[0])):
        column = "".join([grid[i][col] for i in range(len(grid))])
        if f"{piece * 4}" in column:
            return True

    # Vérification des diagonales ascendantes
    for i in range(3, len(grid)):
        for j in range(len(grid[0]) - 3):
            if grid[i][j] == piece and grid[i - 1][j + 1] == piece and grid[i - 2][j + 2] == piece and grid[i - 3][j
                return True

    # Vérification des diagonales descendantes
    for i in range(3, len(grid)):
        for j in range(3, len(grid[0])):
            if grid[i][j] == piece and grid[i - 1][j - 1] == piece and grid[i - 2][j - 2] == piece and grid[i - 3][j
                return True

    return False
```

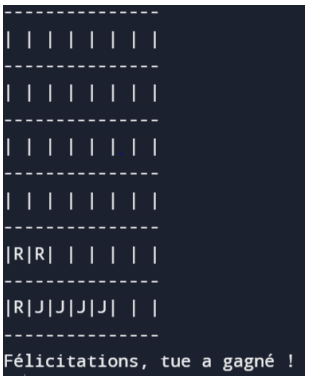
- Vérifier les cas gagnants en utilisant la liste de sous-liste, le pion de joueur (**piece: str)**
- Pour le paramètre **piece: str**, Il existe 2 valeurs : soit **"J"** - jaune ou soit **"R"** - rouge
- Si le joueur gagne, la fonction retourne **True**. Sinon, elle retourne **False**
- Voici les cas gagnants :

Vérification des lignes

```
# Vérification des lignes
for row in grid:
    if f"{piece * 4}" in "".join(row):
        return True
```

- J'ai utilisé **for row in grid** pour lancer la liste de la grille
- Je vérifie si je vois des pions de joueur consécutif (soit **"J" - Jaune** ou soit **"R" - Rouge**) dans la même ligne. La fonction retourne **True**

Par exemple:



Vérification des colonnes

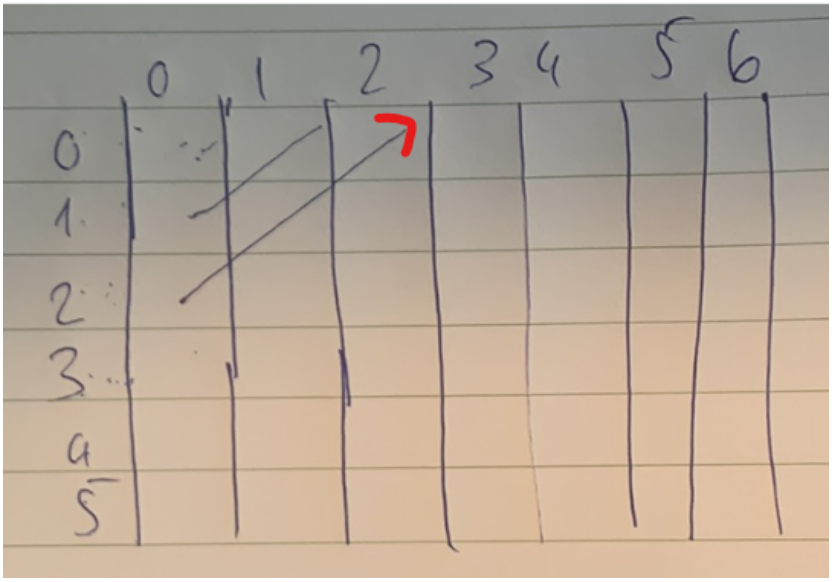
```
21      # Vérification des colonnes
22      for col in range(len(grid[0])):
23          column = "".join([grid[i][col] for i in range(len(grid))])
24          print('column', column)
25          if f"{piece * 4}" in column:
26              return True
27
```

- Pour vérifier des colonnes, j'ai utilisé **for col in range(len(grid[0])**.
- **Len(grid[0])** est la longueur de la sous liste, qui est équivalente à la colonne de la grille (~ 7 colonnes).
- **column** est une collection de pions de la colonne (type **str**). Pour les collecter, j'utilise le **join()** et la boucle.
- La fonction **join()** permet de joindre tous les éléments de la boucle qui retourne.
- La boucle **([grid[i][col] for i in range len(grid))])** retourne les valeurs (~ pions) de colonne selon la valeur **col**. **len(grid)** est le nombre de la ligne. La variable **i** est la position de ligne. La boucle exécutera chaque ligne de la colonne (selon la variable **col**). Exemple: **col = 1** => **column** est une chaîne de la colonne 1.
- Enfin, je vérifie si les pions sont consécutifs dans la même colonne. On retourne **True**

Vérification des diagonales ascendantes

```
28      # Vérification des diagonales ascendantes
29      for i in range(3, len(grid)):
30          for j in range(len(grid[0]) - 3):
31              if grid[i][j] == piece and grid[i - 1][j + 1] == piece and grid[i - 2][j + 2] == piece and grid[i - 3][j + 3] == piece:
32                  return True
33
```

- **for i in range(3,6)**: **La variable i** est équivalent à la ligne. Elle commence à partir du 3. Si c'est 0, 1, 2 alors dans le sens de la diagonale, on ne peut vérifier que jusqu'à 3 valeurs. Voici l'illustration



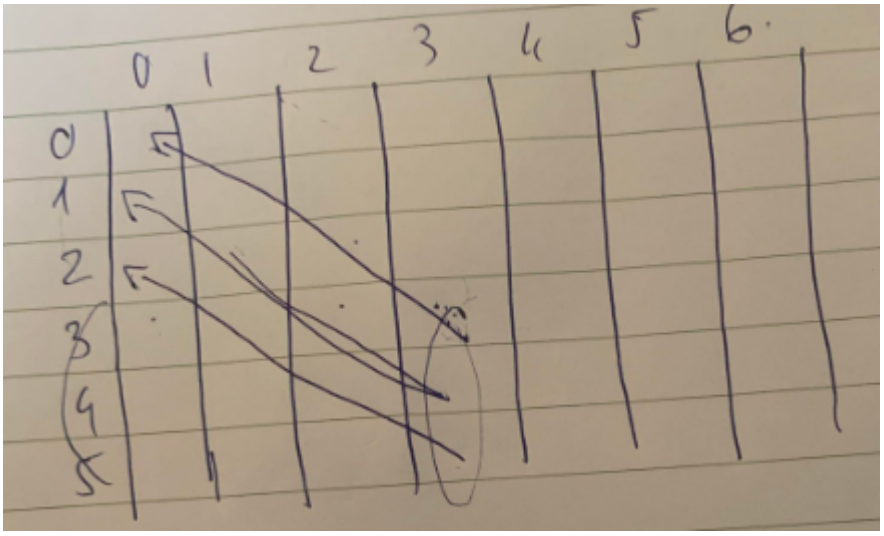
- **for j in range(grid[0]-3) = for j in range(4)**: **j** est équivalent à la colonne. La boucle lance jusqu'à la colonne 4. Dans le sens de la diagonale ascendante, on doit vérifier **[j+3]**. Donc, si le nombre de la boucle est supérieure à 4 => **out of range**
- Vérifier la grille si les pions de joueur sont consécutifs dans le sens diagonale ascendante. La fonction retourne **True**
- Exemple:

```
---------------
| | | | | | | |
---------------
| | | | | | | |
---------------
| | | |J| | | |
---------------
| | |J|J| | | |
---------------
| |J|R|R| | | |
---------------
|J|R|R|J| | |R|
---------------
Félicitations, tue a gagné !
```

Vérification des diagonales descendantes

```
34      # Vérification des diagonales descendantes
35      for i in range(3, len(grid)):
36          for j in range(3, len(grid[0])):
37              if grid[i][j] == piece and grid[i - 1][j - 1] == piece and grid[i - 2][j - 2] == piece and grid[i - 3][j - 3] == piece:
38                  return True
39
```

- **for i in range(3,6)**: **La variable i** est équivalent à la ligne. Elle commence à partir du 3. Si c'est 0, 1, 2 alors dans le sens de la diagonale, on ne peut vérifier que jusqu'à 3 valeurs. Voici l'illustration



-
-

- **for j in range(3, len(grid[0]) = for j in range(3, 7)**: **j** est équivalent à la colonne. La boucle lance jusqu'à la colonne 4. Dans le sens de la diagonale descendante, on doit vérifier **[i-3][j-3]**. Donc, si le nombre de boucles commence à moins de 3 **=> out of range.** Exemple: avec la colonne 2, on ne peut pas vérifier.
- Vérifier la grille si les pions de joueur sont consécutifs dans le sens diagonale ascendante. La fonction retourne **True**
- Exemple:

```
---------------
| | | | | | | |
---------------
| | | | | | | |
---------------
|R| | | | | | |
---------------
|J|R| | | | | |
---------------
|R|R|R|J| | | |
---------------
|J|J|J|R|J| | |
---------------
Félicitations, ok a gagné !
```

Par défaut

- Enfin, si aucun des cas ci-dessus n'est vrai (retourne **True**), la fonction retournera **False**

Fonction start_game()

```
# Fonction principale pour jouer
def start_game():
    player1=input('Nom du 1er joueur (J pion): ')
    player2=input('Nom du 2ème joueur (R pion): ')
    grid = create_grid()
    players = [(player1, "J"), (player2, "R")]
    turn = 0
```

- Une principale fonction pour lancer le jeu
- **player1:** nom du premier joueur
- **player2:** nom du 2ème joueur
- **grid**: grille de 6 lignes et 7 colonnes.
- **players:** liste qui contient deux éléments: nom de joueur, leur pion (soit “j” ou soit “R”)
- **turn**: tour de joueur

Boucle while True

```
while True:
        player, piece = players[turn % 2]
        print_grid(grid)
        try:
            column = int(input(f"{player}, choisissez la colonne (0-6)
pour placer votre pion : "))
            if column < 0 or column > 6:
                print("Choix de colonne invalide. Veuillez choisir une
colonne entre 0 et 6.")
                continue
        except ValueError:
            print("Veuillez entrer un numéro de colonne valide.")
            continue
```

- **while true**: la boucle s’arrête quand on lance **break** => le jeu s'arrête
- **player, piece**: recevoir la valeur de liste **players** en fonction du **tour**. Exemple: **tour = 0** => le tour du premier joueur, **piece = “J”**. **tour = 1** => le tour du deuxième joueur, **piece = “R”**.
- **print_grid(grid):** afficher la grille qui a été définie

- **try et except**: permet de saisir la colonne valide (1 < colonne < 6). Dans cas exceptionnel, l'input de la colonne n'est pas sous forme **integer** => **except ValueError**: recommence la boucle pour saisir la variable valide
- **column:** le numéro de la colonne est saisi par le joueur

Remplissage des pions

```
for i in range(5, -1, -1):
            if grid[i][column] == " ":
                grid[i][column] = piece
                break
        else:
            print("Colonne pleine. Choisissez une autre colonne.")
            continue

        if check_victory(grid, piece):
            print_grid(grid)
            print(f"Félicitations, {player} a gagné !")
            break

        turn += 1

        if turn == 42:
            print_grid(grid)
            print("Match nul !")
            break
```

- **for i in range(5,-1,-1)** : remplir le pion dans la grille dans le cas où cette position est **empty**. La boucle commence de 5 à 0. De bas en haut, si on voit l'espace est vide (" "), on remplit le pion du joueur.
- Dans le cas où la boucle termine. La colonne est saisie, elle n'a plus d'espace. On recommence la boucle pour choisir l'autre colonne
- Dans le cas où la colonne est valide, on vérifie si le joueur a gagné ou non en utilisant **check_victory(grid, piece)** (grille mise à jour)

Check_victory (grid, piece)

```
if check_victory(grid, piece):
    print_grid(grid)
    print(f"Félicitations, {player} a gagné !")
    break
```

- Permet de vérifier de victoire dans plusieurs cas.
- Si elle retourne **True,** le programme affichera un message en fonction du joueur gagnant. On quitte la boucle, **start_game**() s'arrête
- Si elle retourne **False.** Le programme continue

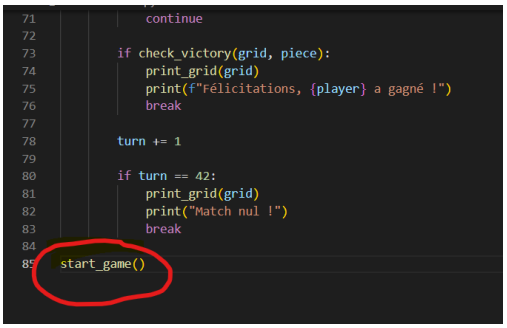
- On incrémente la valeur du **turn** de 1 (**Turn +=1**)

Match nul

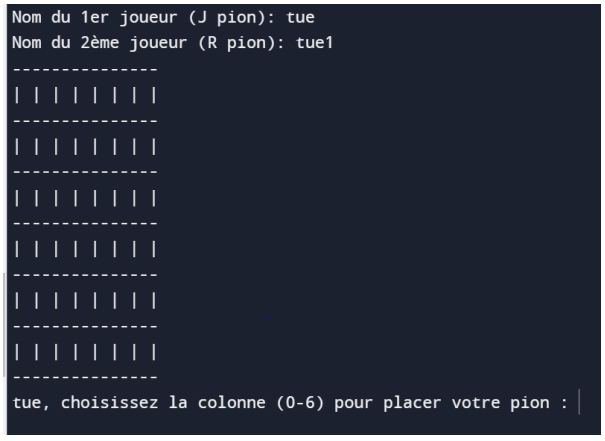
```
78          turn += 1
79
80          if turn == 42:
81              print_grid(grid)
82              print("Match nul !")
83              break
84
```

- Le cas **turn == 42.** La grille est tout remplie, on ne trouve pas le gagnant => **Match Nul**
- On quitte la boucle, **start_game**() s'arrête.

Démarrer le jeu



- Lancer la fonction **start_game()** pour démarrer le jeu
- On va voir l'écran de terminal comme suivant
- Chaque joueur devra inscrire son nom
- Entrez ensuite le nombre de colonnes où le joueur souhaite déposer le pion.
- La grille sera mise à jour après chaque dépose de pion



Version 2

Mise en place les outils nécessaires

Thonny

- Lien de téléchargement: https://thonny.org/

Pygame

- Documentation d'installation: https://www.zonensi.fr/Miscellanees/Pygame/InstallationPygame/InstallationPygame.pdf
- Documentation de librairie: https://www.pygame.org/docs/

Explication des fonctions, des variables

Import librairie

```
2   import pygame
3   import sys
4   import math
5
```

- import pygame: librairie pygame qui permet d'avoir le graphique
- import sys: accéder au système de l'ordinateur.
- import math: calculatrice

Variable globale

```
6   COLUMNS = 7
7   ROWS = 6
8
9   BLUE = (0,0,255)
10  BLACK = (0,0,0)
11  RED = (255,0,0)
12  YELLOW = (255,255,0)
13  YELLOW_PIECE = "J"
14  RED_PIECE = "R"
15  SQUARESIZE = 100
16  RADIUS = int(SQUARESIZE/2 - 5)
17
```

- **COLUMNS**: le nombre de colonne de la grille
- **ROWS**: le nombre de ligne de la grille
- **BLUE**: couleur bleu
- **YELLOW**: couleur jaune
- **RED**: couleur rouge
- **YELLOW_PIECE**: définir pion sous type **str**
- **RED_PIECE:** définir pion sous type **str**

Fonction create_grid()

```
def create_grid() -> list[list[str]]:
    return [[" " for _ in range(COLUMNS)] for _ in range(ROWS)]
```

- Je réutilise la même méthodologie que la version 1

Veuillez trouver l'explication dans la version 1

Fonction is_valid_location(grid: list[list[str]], col: int) -> bool:

```
29  # Vérifier la position empty
30  def is_valid_location(grid: list[list[str]], col: int) -> bool:
31      return grid[ROWS-1][col] == " "
```

- **grid:** grille sous forme d'une liste de sous-liste
- **col**: colonne est sélectionnée
- La fonction permet de vérifier que la dernière ligne de la colonne sélectionnée est toujours vide. ça veut dire la colonnes est pleine ou non

- Si elle est pleine, la fonction retourne **False**. Sinon elle retourne **TRUE**

Fonction get_next_open_row(grid: list[list[str]], col: int) -> (int | None):

```
33  def get_next_open_row(grid: list[list[str]], col: int) -> (int | None):
34      for r in range(ROWS):
35          if grid[r][col] == " ":
36              return r
37
```

- **grid:** grille sous forme d'une liste de sous-liste
- **col**: colonne est sélectionnée
- **for r in range(ROWS):** la boucle permet de chercher la première ligne de la colonne sélectionnée. Si la fonction la trouve, elle retourne la position de cette ligne. Sinon, elle retourne **None**

Fonction check_victory(grid: list[list[str]], piece: str):

```
38  def check_victory(grid: list[list[str]], piece: str):
39      # Vérification des lignes
40      for row in grid:
41          if f"{piece * 4}" in "".join(row):
42              return True
43
44      # Vérification des colonnes
45      for col in range(len(grid[0])):
46          column = "".join([grid[i][col] for i in range(len(grid))])
47          if f"{piece * 4}" in column:
48              return True
49
50      # Vérification des diagonales ascendantes
51      for i in range(3, len(grid)):
52          for j in range(len(grid[0]) - 3):
53              if grid[i][j] == piece and grid[i - 1][j + 1] == piece and grid[i - 2][j + 2] == piece and grid[i - 3][j + 3] == piece:
54                  return True
55
56      # Vérification des diagonales descendantes
57      for i in range(3, len(grid)):
58          for j in range(3, len(grid[0])):
59              if grid[i][j] == piece and grid[i - 1][j - 1] == piece and grid[i - 2][j - 2] == piece and grid[i - 3][j - 3] == piece:
60                  return True
61
62      return False
63
```

- Je réutilise la même méthodologie que la version 1

Veuillez trouver l'explication dans la version 1

Fonction draw_grid(grid: list[list[str]])

Mis en place le background bleu avec le cercle noir

```
64  def draw_grid(grid: list[list[str]]):
65      # Dessiner le rectangle bleu avec le cercle noir
66      for c in range(COLUMNS):
67          for r in range(ROWS):
68              pygame.draw.rect(screen, BLUE, (c*SQUARESIZE, r*SQUARESIZE+SQUARESIZE, SQUARESIZE, SQUARESIZE))
69              pygame.draw.circle(screen, BLACK, (int(c*SQUARESIZE+SQUARESIZE/2), int(r*SQUARESIZE+SQUARESIZE+SQUARESIZE/2)), RADIUS)
```

- cette fonction pour dessiner la grille en mettant la rectangle bleu ,
- **for c in range(col=7), for r in range(row=6)**: pour dessiner le background en rectangle bleu avec le cercle noir au-dessus

- **pygame.draw.rect(surface, color,(top, left, width, height))**: dessiner le rectangle de la taille 100x100px selon la position (ligne et colonne)
- **pygame.draw.circle(surface, color, center, radius)**: le paramètre **center** est sous forme (x,y). **Radius** (=45px) est le rayon du cercle

Mise à jour la grille avec le pion

```
71      # mettre à jour le tableau avec le pion
72      for c in range(COLUMNS):
73          for r in range(ROWS):
74              if grid[r][c] == YELLOW_PIECE:
75                  pygame.draw.circle(screen, YELLOW , (int(c*SQUARESIZE+SQUARESIZE/2), height-int(r*SQUARESIZE+SQUARESIZE/2)), RADIUS)
76              elif grid[r][c] == RED_PIECE:
77                  pygame.draw.circle(screen, RED, (int(c*SQUARESIZE+SQUARESIZE/2), height-int(r*SQUARESIZE+SQUARESIZE/2)), RADIUS)
78      pygame.display.update()
```

- On lance la boucle pour vérifier la grille. Si on trouve “J”, on dessine le cercle jaune. Si on trouve “R”, on dessine le cercle rouge
- La position de dessin, on commence de bas en haut => la position y du centre du cercle est **height-int(r*SQUARESIZE+SQUARESIZE/2)**

pygame.dispay.update(): mettre à jour l'écran **pygame**

Fonction principale

```
82      grid = create_grid()
83      # print_grid(grid)
84      game_over = False
85      turn = 0
86      turn_count = 0
87      player1=input('Nom du 1er joueur (J pion): ')
88      player2=input('Nom du 2ème joueur (R pion): ')
89      players = [(player1, YELLOW_PIECE), (player2, RED_PIECE)]
90
91      pygame.init()
92
93      width = COLUMNS * SQUARESIZE
94      height = (ROWS+1) * SQUARESIZE
95
96      size = (width, height)
97
98      screen = pygame.display.set_mode(size)
99      draw_grid(grid)
100     pygame.display.update()
101
102     # mettre le style du text et sa taille - 24px
103     myfont = pygame.font.SysFont("monospace", 24)
104
```

- ça commence à partir de la ligne 80
- **player1:** nom du premier joueur
- **player2:** nom du 2ème joueur
- **grid**: grille de 6 lignes et 7 colonnes.
- **players:** liste qui contient deux éléments: nom de joueur, leur pion (soit “j” ou soit “R”)
- **turn**: identifier le tour de joueur
- **turn_count**: compter le nombre du tour. Si **turn_count** est 42, ça veut dire que la grille est toute remplie, on ne trouve pas le gagnant => Match Nul
- **game_over**: marquer si le jeu s'est arrêté ou non. **False**, le jeu n’est pas fini. **True**, il termine
- **width** : largeur d'écran en pixel (px)
- **height** : hauteur d’écran en pixel (px)
- **size**: la taille de l’écran sous forme **(width, height)**
- **py.init()**: initialiser la librairie pygame
- **screen = pygame.display.set_mode(size):** mettre en place de l’écran du jeu selon la variable **size**
- **myfont = pygame.font.SysFont("monospace", 16)**: mettre en place du style de text et sa taille
- **draw_grid(grid)**: afficher l’écran du jeu
- **pygame.display.update()**: Mettre à jour l’écran du jeu

While not in game_over

- La boucle s’arrête dès que **game_over** est **True**

```
107
108     for event in pygame.event.get():
109         if event.type == pygame.QUIT:
110             sys.exit()
111
```

- **for event in pygame.event.get()**: permet de récupérer les interactions, les événements sur l’écran de pygame
- **event.type:** est le type de l’événement
- par exemple: **MOUSEMOTION**: la souris bouge sur l'écran. **MOUSEBUTTONDOWN**: clique de souris

Cas event == pygame.QUIT

- programme s’arrête

Cas event == pygame.MOUSEMOTION

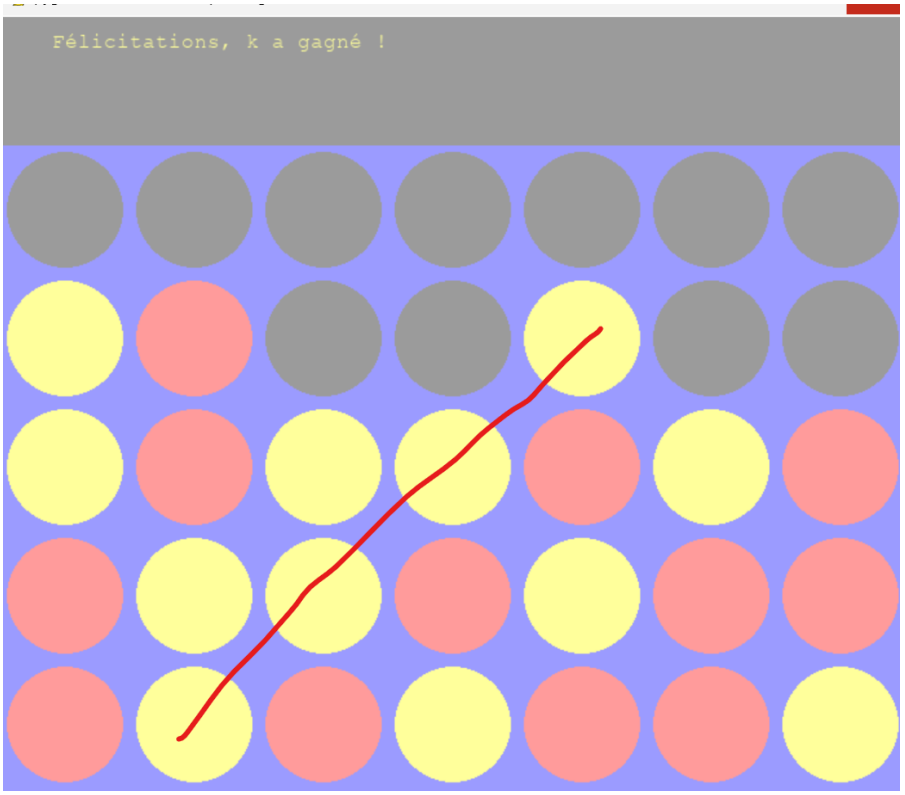
```
107
108     for event in pygame.event.get():
109         if event.type == pygame.QUIT:
110             sys.exit()
111
112         # Dessiner le background noir et la couleur pion en fonction du tour de joueur
113         # turn == 0 => le tour du premier joueur, sinon (turn == 1) => le tour du deuxième joueur
114         if event.type == pygame.MOUSEMOTION:
115             pygame.draw.rect(screen, BLACK, (0,0, width, SQUARESIZE))
116             posx = event.pos[0]
117             if turn == 0:
118                 pygame.draw.circle(screen, YELLOW, (posx, int(SQUARESIZE/2)), RADIUS)
119             else:
120                 pygame.draw.circle(screen, RED, (posx, int(SQUARESIZE/2)), RADIUS)
121         pygame.display.update()
122
```

- Dessiner le background noir du jeu
- **posx = event.pos[0]**: suivre le mouvement de souris dans le sens horizontal
- Afficher un cercle avec la couleur du joueur
- Le pion jaune est le premier joueur. Le pion rouge est le deuxième joueur

Cas event == pygame.MOUSEBUTTONDOWN

```
123         if event.type == pygame.MOUSEBUTTONDOWN:
124             pygame.draw.rect(screen, BLACK, (0,0, width, SQUARESIZE))
125
126             # Demander l'input du premier joueur (pion jaune)
127             if turn == 0:
128                 player, piece = players[turn]
129
130                 posx = event.pos[0]
131                 col = int(math.floor(posx/SQUARESIZE))
132
133                 if is_valid_location(grid, col):
134                     row = get_next_open_row(grid, col)
135                     drop_piece(grid, row, col, piece)
136
137                     if check_victory(grid, piece):
138                         label = myfont.render(f"Félicitations, {player} a gagné !", 1, YELLOW)
139                         screen.blit(label, (40,10))
140                         game_over = True
```

- Recevoir l'événement de clic de la souris
- Dessiner le background noir du jeu
- **posx**: position de la souris dans le sens horizontal.
- **col**: position de la colonne où on clique la souris
- **is_valid_location()**: vérifier si la colonne sélectionnée n'est pas pleine
- **row**: position de la première ligne de la colonne sélectionnée
- **drop_piece**: mettre le pion selon la ligne trouvée et la colonne sélectionnée. Et puis mettre à jour la grille
- **check_victory():** vérifier si on a le gagnant. Si elle retourne **True**, le programme affichera le nom du gagnant. Sinon le programme continue
- **draw_grid(grid):** dessiner l'écran du jeu avec la grille mise à jour
- Par exemple: le jaune a gagné



Match nul!

```
161
162     if turn_count == 42:
163         label = myfont.render("Match nul!", 1, BLUE)
164         screen.blit(label, (40,10))
165         game_over = True
166
```

- Le cas **turn == 42.** La grille est tout remplie, on ne trouve pas le gagnant => **Match Nul**
- L'écran s'affiche "Match nul! ". (40,10) = position (x,y) du text sur l'écran
- On quitte la boucle en mettant **game_over = True** car la fonction a commencé par **while not game_over**

La répartition des tâches entre chaque membre de l’équipe.

J’ai fait toute seule, mon binôme est absent.

Conventions de nommages de vos variables et fonctions

Variables Globales

- Ils sont nommé en **SNAKE_CASE**
- Exemple: YELLOW_PIECE = “J”, COLUMNS = 6

Variables

- Pour les variables dans les fonctions (pas globales), elles sont nommées en **snake_case.**
- Par exemple: **game_over**

Fonctions

- La fonction est nommée en **snake_case** avec le premier mot commence par un verbe.
- Exemple: **create_grid**()