Sujet - Projet Jeu du Puissance 4

May 23, 2024

1 Jeu du Puissance 4

Dans ce projet **guidé** vous allez créer un programme permettant de jouer au jeu du Puissance 4.

Remarques préliminaires : 1) Dans du code Python, les triples guillemets `"""` (ou triple apostrophe `'''`) permettent de délimiter une chaîne de caractères située **sur plusieurs lignes** (chaîne de caractères multi-lignes). Ceci est très pratique pour écrire les spécifications au début des fonctions et également pour mettre en commentaire toute une partie du code que l'on ne veut pas encore tester mais que l'on veut conserver quand même. **2)** Lorsque vous exécutez un programme Python et **qu'il reste bloqué**, vous pouvez l'arrêter en appuyant sur *CTRL+C ou CTRL+D*.

3) Vos bloc de code de TEST **seront notés** dans ce projet. Suivez donc bien attentivement les instructions.

1.1 1) Travail à faire:

- Sur votre bureau d'ordinateur par exemple, créer un dossier **"Projet Jeu du Puissance 4"**.
- Dans ce dossier, et à l'aide de **Spyder** (ou d'un autre Editeur Python comme Thonny), créer un fichier **NOM-PRENOM-Puissance4.py** qui contient le code suivant (à copier coller):

```
[ ]: # ********************************
     # Projet NSI : JEU DU PUISSANCE 4
     # Date: ..........
     # Nom: ............
     # Prénom: ............
     # ********************************
```

1.2 Présentation du Puissance 4 attendu.

La grille du jeu sera affichée comme ci dessous:

Le jeu se joue à deux joueurs, avec des jetons qui seront représentés par `X` pour le Joueur 1 et par `O` pour le Joueur 2.

Lisez **attentivement** et **COMPLETEMENT** les deux séquences de jeu fournies avec le sujet du projet: * `Exemple_sequence_de_jeu_n1.pdf` * `Exemple_sequence_de_jeu_n2.pdf`

1.3 2) Partie INITIALISATION

Il faut décomposer notre programme en plusieurs morceaux. Il y aura toujours les 3 parties classiques:
* INITIALISATION * FONCTIONS ET PROCEDURES * PROGRAMME PRINCIPAL .

Ajouter le code suivant au début de votre programme:

RAPPEL SUR NOTION DE VARIABLE GLOBALE:

Dans ce projet, nous utiliserons des **variables globales**, c'est-à-dire, des variables qui sont accessibles à la fois dans le programme principal mais également à l'intérieur de toutes les fonctions et procédures que nous écrirons. Ce sera le cas de la variable `grille` qui contient la grille du jeu et que nous avons initialisée juste ci-dessus.

A l'intérieur d'une fonction, si l'on veut remplir ou utiliser une variable déjà déclarée dans le programme principal ou bien au tout début du programme dans la partie initialisation, il faudra la déclarer en début de la fonction à l'aide du mot clef `global`.

C'est pour cette raison que vous verrez la ligne de code `global grille` dans la fonction donnée ci-dessous:

1.4 Partie FONCTIONS ET PROCEDURES

1.5 3) Procédure `creation_grille_vierge`

- Ajouter le code suivant dans votre programme et **compléter la ligne 27**:

[11]:
```
# -----------------------------------------------------------
# -------- FONCTIONS ET PROCEDURES --------------------------
# -----------------------------------------------------------

def creation_grille_vierge():
    ''' Cette fonction va remplir la grille du jeu par des "."

        IN: Rien
        OUT: Rien car on va remplir la variable globale grille, déjà déclarée␣
 ↪au tout début du programme

        On fabrique une liste de liste de string (de 1 caractère) modélisant␣
 ↪notre grille de jeu de 6 lignes par 7 colonnes
            [ [".", ".", ".", "." , "." , "." , "."],
              [".", ".", ".", "." , "." , "." , "."],
              ...
              [".", ".", ".", "." , "." , "." , "."] ]
    '''
    # On déclare que l'on va utiliser la variable globale grille (déjà␣
 ↪initialisée)
    global grille

    # Pour chaque ligne
```

```
    for num_ligne in range(6):
        ligne = [] # on crée un nouveau tableau vide en mémoire
        for j in range(7): # On ajoute les sept éléments ".", séparémment.
            ligne.append(".")

        # On ajoute la nouvelle ligne à la grille
        grille.append(.......)

    # Le travail est terminé, la variable globale grille est remplie, on quitte␣
↪la procédure
    return

# ------ TEST TEMPORAIRE --------
#creation_grille_vierge()
#print(grille)
```

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

2 Rappel important sur les tableaux de tableaux:

Un tableau bi-dimensionnel se représente par une variable de type tableau de tableaux ou encore une liste de liste (structure identique aux matrices en mathématiques)

`grille[i][j]` donnera accès à la case située à la **`ligne d'indice i`** et la **`colonne d'indice j`**

Par cohérence avec vos apprentissage futurs (maths. physique, sciences ...), **employez toujours cette convention** pour vos variables:

- **`i`** indice de la **ligne**
- **`j`** indice de la **colonne**

Retenir: `tableau[ligne][colonne]` Par exemple si l'on considère cette grille de jeu:

`grille[0][0]` contient `"."` , tout en haut à gauche.

`grille[2][3]` contient `"X"` (à la ligne d'indice **i=2** et la colonne d'indice **j=3**)

`grille[5][6]` contient `"O"` , tout en bas à droite (à la ligne d'indice **i=5** et la colonne d'indice **j=6**).

2.1 4) Fonction `affiche_grille`

Il nous faut maintenant une fonction qui affiche cette grille dans la console Python, avec une présentation identique à celle représentée plus haut. Copier-Coller la fonction suivante en l'ajoutant à la suite de votre code et compléter là:

[12]:
```
def affiche_grille() -> None :
    ''' Cette fonction affiche la grille de jeu telle que ci-dessous
    IN: Rien
```

```
    OUT: Rien
    Affichage souhaitée :
       0 1 2 3 4 5 6

    0 |.|.|.|.|.|.|.|
    1 |.|.|.|.|.|.|.|
    2 |.|.|.|.|.|.|.|
    3 |.|.|.|.|.|.|.|
    4 |.|.|.|X|.|.|.|
    5 |O|X|.|X|O|X|O|
      ---------------
    '''
    global grille

    # Affichage des indices du haut de la grille (0 à 6)
    print("   ", end="")
    for i in range(...):
        print(str(i)+" ",end="")
    print("\n  ") # '\n' est un saut de ligne  (passage à une nouvelle ligne)␣
 ↪suivi de deux espaces

    # Affichage des lignes
    for i in range(...): # Pour chaque ligne i
        print(str(i)+" ",end="")
        for j in range(...): # Pour chaque colonne j
            print("|"+ grille[i][j],end ="")
        print("|")

    # Affichage du trait en bas de la grille
    print(.........)


# -------- TEST TEMPORAIRE ------------
....
....
```

TEST TEMPORAIRE:

Dans la zone TEST ci-dessus, ajoutez quelques pions avec des lignes de codes : * `grille[2][3] = "O"` * `grille[5][0] = "X"` * `grille[5][1] = "X"` etc.

et testez cette fonction afin de vérifier que votre affichage est tel que demandé.

Une fois que tout fonctionne bien, **laissez vos instructions de test apparentes**, elles seront **notés**. Mais remettez en commentaire celles-ci pour pouvoir continuer le projet à l'aide des triples quote pour rappel, comme ci-dessous :

```
# -------- TEST TEMPORAIRE ------------

'''
```

```
Bloc de code de test
'''
```

2.2 5) Fonction `colonne_pleine`

- Nous allons avoir besoin d'une fonction qui détecte si une colonne est déjà pleine. En effet, on ne peut plus jouer dans une colonne qui est déjà pleine. Ajoutez la fonction suivante et complétez là:

[ ]:
```
def colonne_pleine(indice_colonne: int) -> bool:
    '''
        IN: indice de la colonne à analyser (0 à 6)
        OUT: un booleen (True si la colonne est déjà pleine, False sinon)
    '''
    global grille

    .........
    .........
    .........

# --------- TEST TEMPORAIRE --------------
.......
.......
```

TEST TEMPORAIRE:

Ajoutez des pions dans votre grille pour remplir une colonne (on pourra utiliser un boucle for) et testez votre fonction pour s'assurer de son bon fonctionnement.

Comme précédemment, **laissez vos instructions de test apparentes**, en zone de commentaire, elles feront partie de la **note** du projet.

2.3 6) Procédure `joue_jeton`

Nous avons besoin d'une procédure `joue_jeton(num_joueur, indice_colonne)` qui joue un nouveau jeton du joueur spécifié dans la colonne indiquée.

Evidemment le jeton doit se placer **le plus bas possible** dans la colonne indiquée puisque la grille de jeu est verticale.

Il faut donc rechercher la première case libre en partant du bas et y placer le jeton.

Vous pourrez utiliser pour cela une boucle `for` avec un indice de ligne `i` qui décroit (revoir `for i in range(m,n,p)` avec `p = -1`)

Souvenez vous également que : `jetons_joueur[0]` contient `'X'` et que `jetons_joueur[1]` contient `'O'`.

Ajoutez et compléter la procédure suivante:

[ ]:
```python
def joue_jeton(num_joueur: int, indice_colonne: int) -> None:
    ''' Place un jeton du joueur numéros num_joueur, dans la colonne␣
 ↪indice_colonne.

        IN: num_joueur (int qui vaut 1 ou 2)
        OUT: Rien puisque cette fonction va modifier directement la variable␣
 ↪global grille.
    '''
    # On utilise les deux variables globales suivantes
    global grille
    global jetons_joueur

    # Dans la colonne indice_colonne, en partant, du bas, on cherche la␣
 ↪première case vide.
    ............
    ............
    ............

# ------------ TEST TEMPORAIRE -----------------
'''
joue_jeton(1, 3)  # Joueur 1 joue
affiche_grille()

joue_jeton(2, 0)  # Joueur 2 joue
affiche_grille()

joue_jeton(1, 3)  # Joueur 1 joue
affiche_grille()

etc.
'''
```

TEST TEMPORAIRE: Testez cette fonction et vérifiez que les jetons ont été convenablement positionné dans la grille. **Laissez vos tests en commentaire.**

2.4 7) Fonction `demander_ou_jouer`

On va créer une fonction `demander_ou_jouer()` qui demandera à l'utilisateur dans quel indice de colonne il souhaite jouer, qui fera les vérifications nécessaires et qui nous retournera un indice de colonne jouable (colonne non pleine).

[19]:
```python
def demander_ou_jouer() -> int:
    ''' Doit demander au joueur dans quel indice de colonne il souhaite jouer.
        Si l'indice n'est pas valable (non compris entre 0 et 6), ou bien s'il␣
 ↪correspond à une colonne pleine, on lui indique
        que sa saisie est incorrecte et on lui renouvelle la question.
        Si l'utilisateur saisie 'Q' (pour "Quitter"), la partie doit se␣
 ↪terminer.
```

```
        IN: rien
        OUT: Renvoie un indice de colonne (int) valable (colonne non pleine) où␣
↪l'on peut jouer.
    '''

    while True:
        saisie = input(.....................)

        if len(saisie)==... and (saisie in "..........."):
            #La saisie est correct (1 seul caractère et il est autorisé)
            if saisie==... :
                exit()

            # On vérifie que la colonne n'est pas pleine
            j = int(.......)

            if colonne_pleine(......):
                print("ATTENTION, cette colonne est déjà pleine !")
            else: #Sinon, il y a encore de la place
                # On renvoie l'indice de la colonne choisie
                return ......
        else:
            print("SAISIE INCORRECTE")

# ------------ TEST TEMPORAIRE ----------------
'''
colonne = demander_ou_jouer()
joue_jeton(1, colonne)  # Joueur 1 joue
affiche_grille()

etc.
'''
```

TEST TEMPORAIRE:

Testez cette fonction en simulant deux ou trois coups en alternant joueur 1 et 2. **Laissez vos tests en commentaire.**

2.5 8) Fonction `Quatre_jetons_en_ligne`

Nous arrivons dans la partie de détection d'une victoire. Elle peut avoir lieu avec : - 4 jetons alignés en ligne; - 4 jetons alignés en colonne; - 4 jetons alignés en diagonale.

Pour faire cette détection, une méthode possible consiste à utiliser les chaines de caractères:

En effet avec une chaîne de caractère qui représente le contenu d'une ligne, par exemple `"O.XXXXO"`, il est facile de détecter la présence de `"XXXX"` grâce à l'opérateur `in` (rappel: `texte in chaine` renvoie un booleen) :

On peut faire pareil avec les colonnes (en recopiant le contenue d'une colonne dans une chaine de caractère). Et l'on peut aussi faire pareil avec les diagonales (en recopiant le contenue d'une diagonale une chaine de caractère).

Commençez par ajouter la fonction `Quatre_jetons_en_ligne()` ci-dessous et compléter là:

[ ]:
```
def Quatre_jetons_en_ligne(num_joueur: int) -> bool:
    '''
        IN: Numéros du joueur à détecter 1 ou 2
        OUT: booleen (True si 4 jetons alignés trouvés en ligne, False sinon)
    '''
    # On déclare les variables globales qui nous seront utiles
    global jetons_joueur
    global grille

    # définition du jeton à trouver
    jeton = jetons_joueur[.............]

    chaine_a_trouver = jeton * 4

    ........................
    ........................
    ........................
    ........................
    ........................
    ........................
    ........................

    # Si on arrive ici, c'est qu'aucun alignement de 4 jetons n'a été trouvé en␣
 ↪ligne
    return False


# ------------ TEST TEMPORAIRE ----------------
'''


'''
```

TEST TEMPORAIRE:

Testez cette fonction en construisant au préalable une grille gagnante en ligne et une autre non gagnante. **Laissez vos tests en commentaire.**

2.6 9) Fonction `Quatre_jetons_en_colonne`

Ajouter la fonction `Quatre_jetons_en_colonne()` sur le même principe que la fonction précédente.

Elle devra **renvoyer un booléen** (True si 4 jetons alignés trouvés en colonneligne et False sinon).

TEST TEMPORAIRE:

Testez cette fonction en construisant au préalable une grille gagnante en colonne et une autre non gagnante. **Laissez vos tests en commentaire.**

2.7 10) Fonction `Quatre_jetons_diagonal`

Nous arrivons ici à une fonction un peu plus délicate à écrire: Celle qui va détecter un alignement de 4 jetons en diagonale.

Elle devra renvoyer un booleen.

Il y a plusieurs façons de procéder. En voici une: On peut découper le problème en deux parties: * **PARTIE 1** : Détection alignement 4 jetons en **diagonale descendante vers la droite**; * **PARTIE 2** : Détection alignement 4 jetons en **diagonale descendante vers la gauche**;

2.7.1 Pour la partie 1:

On peut déjà réfléchir à trouver les coordonnées des cases de départs possibles pour **un alignement de 4 jetons**. Si on met le caractère `D` dans ces cases, voilà ce que l'on trouve:

```
    0   1   2   3   4   5   6
0   D   D   D   D   .   .   .
1   D   D   D   D   .   .   .
2   D   D   D   D   .   .   .
3   .   .   .   .   .   .   .
4   .   .   .   .   .   .   .
5   .   .   .   .   .   .   .
   ---------------------------
```

Ainsi, pour chacune de ces cases là, il faut tester si on a un alignement de 4 jetons identiques en descendant vers la droite. Il ne peut pas y en avoir ailleurs. Pour une case `grille[i][j]`, sa voisine en bas à droite est donc `grille[i+1][j+1]`, puis `grille[i+2][j+2]` etc.

2.7.2 Pour la partie 2:

On fait la même chose mais en descendant vers la gauche cette fois-ci. Les cases "point de départ" possibles sont:

```
    0   1   2   3   4   5   6
0   .   .   .   D   D   D   D
1   .   .   .   D   D   D   D
2   .   .   .   D   D   D   D
3   .   .   .   .   .   .   .
4   .   .   .   .   .   .   .
5   .   .   .   .   .   .   .
   ---------------------------
```

Maintenant que nous avons décortiqué le problème, ajouter et compléter la fonction suivante:

[ ]:
```python
def Quatre_jetons_diagonal(num_joueur: int) -> bool:
    '''
```

```
        IN: Numéros du joueur à détecter 1 ou 2.
        OUT: booleen (True si 4 jetons alignés trouvé en diagonale, False sinon)
    '''
    global grille
    global jetons_joueur

    jeton = jetons_joueur[..................]

    chaine_a_trouver = jeton *4

    # -----------------------------------------------------
    # PARTIE 1 : Recherche sur les diagonales descendantes vers la droite:
    '''
    # On définit la liste des coordonnées des points de départ possible pour les diagonales descendantes vers la droite.
        0   1   2   3   4   5   6
    0   X   X   X   X
    1   X   X   X   X
    2   X   X   X   X
    3
    4
    5  ----------------------------

    '''

    ........................
    ........................
    ........................
    ........................
    ........................
    ........................
    ........................


    # Si un alignement en diagonale a été trouvé
    if ............:
        # Une diagonale complète trouvée
        print("VICTOIRE EN DIAGONALE DE " + jeton)
        return True

    #----------------- FIN PARTIE 1 ----------------------


    # -----------------------------------------------------
    # PARTIE 2 : Recherche sur les diagonales descendantes vers la gauche:
    '''
```

```
    # On définit la liste des coordonnées des points de départ possible pour␣
 ↪les diagonales descendantes vers la gauche.
        0   1   2   3   4   5   6
    0               X   X   X   X
    1               X   X   X   X
    2               X   X   X   X
    3
    4
    5  ---------------------------

    '''


    ........................
    ........................
    ........................
    ........................
    ........................
    ........................
    ........................

    # Si un alignement en diagonale a été trouvé
    if ......................:
        # Une diagonale complète trouvée
        print("VICTOIRE EN DIAGONALE DE " + jeton)
        return True

    #----------------- FIN PARTIE 2 -----------------


    # Si on arrive ici, aucune diagonale n'a été trouvée, on renvoie False
    return False

# ----------- TEST TEMPORAIRE -------------
'''

'''
```

TEST TEMPORAIRE:

Testez cette fonction en construisant au préalable une grille gagnante en diagonale. **Laissez vos tests en commentaire.**

2.8 11) Fonction `Recherche_si_victoire`

Il nous faut maintenant une fonction de recherche de victoire d'un joueur donné. Elle va bien sur utiliser les 3 fonctions précédentes.

Ajouter et compléter la fonction suivante:

[ ]:
```
def Recherche_si_victoire(num_joueur) -> bool:
    '''
        IN: num_joueur (1 ou 2)
        OUT: un booléen (True si le joueur indiqué a gagné, False sinon)
    '''
    ........................
    ........................
    ........................
    ........................

    return ...

# ----------- TEST TEMPORAIRE -------------
'''

'''
```

2.9 12) Fonction `grille_pleine`

Il est possible que la grille devienne pleine sans aucune victoire. Dans ce cas, il faudra que la partie s'arrête. Il nous faut donc une fonction pour le détecter.

Elle devra **renvoyer un booleen** : `True` si la grille est pleine et `False` sinon.

Est-il vraiment nécessaire de tester si toutes les lignes sont pleines ?

Ecrire la fonction `grille_pleine()` **qui fasse le moins de test possible** dans votre programme.

2.10 13) Programme principal

Il faut maintenant assembler tout ce que l'on a fait précédemment pour créer le jeu.

N'oubliez pas de mettre des **commentaires** pour expliquer ce que fait votre code. Ils seront pris en compte dans **la note du projet**.

Ajouter puis compléter le code suivant:

[ ]:
```
# ------------------------------------
# -------- PROGRAMME PRINCIPAL ---------
# ------------------------------------

creation_grille_vierge()
affiche_grille()

# Tant que la partie n'est pas terminée, un joueur joue.
while partie_terminee==False:

    ........................
    ........................
```

```
    ........................
    ........................
    ........................
    ........................
    ........................
    ........................


    # On change le numéros du joueur courant
    if num_joueur_courant==1:
        num_joueur_courant=2
    else:
        num_joueur_courant=1

# FIN DU WHILE

# On est sorti de la boucle donc:
print("FIN DE PARTIE")
```

2.11 14) Jouer contre l'ordinateur

Avez Spyder ou Thonny, **créer un nouveau programme** nommé **NOM-PRENOM-Puissance4-VS-COMPUTER.py**

Copier-coller y tout votre code précédent.

Vous allez **modifier/ajouter** tout ce qu'il faut afin que **le joueur 2 soit l'ordinateur**.

Nous allons simplement nous contenter de faire jouer l'ordinateur **dans une colonne au hasard**. Comme l'ordinateur est très rapide, il faudra ajouter un peu de code pour pouvoir "attendre" une demi-seconde par exemple. Voici le code pour faire cela:.

```
import time # A mettre en début de programme

time.sleep(0.5) # attends 1/2 second
```

2.11.1 Générer un nombre entier au hasard:

```
import random # A mettre en début de programme

random.seed() # A mettre dans la partie INITIALISATION
```

Cette fonction `seed()` sert à ré-initialiser le générateur aléatoire sur l'heure de la machine. De cette façon, les nombres tirés au hasard ne sont pas toujours les même.

```
print(random.randint(a,b)) # qui renvoie un entier compris entre a et b.
```

2.12 Pour aller plus loin (facultatif):

Pour ceux qui se sente de faire mieux, c'est-à-dire, de jouer le "meilleur coup", vous pouvez me faire une version supplémentaire mais je vous demande dans tous les cas de **me rendre la version "l'ordi joue au hasard".**

2.12.1 On touche du doigt le sujet de l' IA (Intelligence Articielle):

Domaine très intéressant, mais complexe et un peu ambitieux à ce stade de l'année.

Il pourra cependant être mieux traité **en classe de Terminale** grâce à la **notion d'arbre** et d' **algorithme Min-Max**.

L'idée est de simuler les coups possibles en créant en mémoire toutes les grilles correspondantes (en plus de la grille actuelle du jeu).

Pour chacune d'elle, il faut pouvoir lui donner une note (sur 100 par exemple).

On aurait donc besoin d'une fonction `Eval_grille(une_grille) -> int` qui note la grille donnée en paramètre. Ensuite, on pourra choisir le coup qui donne la meilleure note. La note de 100 pourra par exemple être attribuée à une grille gagnante. La note de 50 pour une grille qui donne 3 jetons alignés, et 25 pour une grille qui donne 2 jetons alignés. Mais on peut encore compliqué cela en recherchant le coup qui empêche l'adversaire de gagner au prochain coup… Cela sera possible avec la notion d'arbre qui sera vue en Terminale.

J.B. Mouzet (mailto:jbmouzet@gmail.com), Lycée Camille Sée - 75015 Paris