

connect four gAME



|  |
| --- |
|  |
|  | import random: **Random values** |
|  | import pygame: **To create the board and graphics** |
|  | import sys: **To be able to control system like exiting it** |
|  | import math**: To use math values like infinity**  import numpy as np: **library to make so things more easier** |
|  |  |
|  | BLUE = (0,0,255) **//A very blue RGB value (Red, Green, Blue)** |
|  | BLACK = (0,0,0) |
|  | RED = (255,0,0) |
|  | YELLOW = (255,255,0) |
|  |  |
|  | ROW\_COUNT = 6 |
|  | COLUMN\_COUNT = 7 |
|  |  |
|  | PLAYER = 0 |
|  | AI = 1 |
|  |  |
|  | EMPTY = 0 |
|  | PLAYER\_PIECE = 1 |
|  | AI\_PIECE = 2 |
|  |  |
|  | WINDOW\_LENGTH = 4 |
|  |  |
|  | def create\_board(): |
|  | board = np.zeros((ROW\_COUNT,COLUMN\_COUNT)) |
|  | return boar**d**  **//first code we create board with library numpy** |
|  |  |
|  | def drop\_piece(board, row, col, piece): |
|  | board[row][col] = piece |
|  | **//create function of drop piece** |
|  | def is\_valid\_location(board, col): |
|  | return board[ROW\_COUNT-1][col] == 0 |
|  | **//create function search for valid location by search in the row 5 if it is empty or no to let the piece go in the col** |
|  | def get\_next\_open\_row(board, col): |
|  | for r in range(ROW\_COUNT): |
|  | if board[r][col] == 0: |
|  | return r |
|  | **// check for the next available row** |
|  | def print\_board(board): |
|  | print(np.flip(board, 0)) |
|  | **// make the piece in the row 0 by flip and zero to swap the last row and first row** |
|  | def winning\_move(board, piece): |
|  | **//Check horizontal locations for win by see if there are four in a horizontal**  **And we check that or range to 4 columns from zero to 4 because we cannot make two (four pieces in one row )**  **We increase the Colum one by one to make it in horizontal** |
|  | for c in range(COLUMN\_COUNT-3): |
|  | for r in range(ROW\_COUNT): |
|  | if board[r][c] == piece and board[r][c+1] == piece and board[r][c+2] == piece and board[r][c+3] == piece: |
|  | return True |
|  |  |
|  | **//Check vertical locations for win see if there are four in a vertical**  **And we check that or range to 3 rows from zero to 3 because we cannot make two (four pieces in one columns )**  **We increase the row one by one to make it in vertical** |
|  | for c in range(COLUMN\_COUNT): |
|  | for r in range(ROW\_COUNT-3): |
|  | if board[r][c] == piece and board[r+1][c] == piece and board[r+2][c] == piece and board[r+3][c] == piece: |
|  | return True |
|  |  |
|  | **// Check positively sloped diagonals see if there are four in a diagonal forward**  **And we check that or range to 3 rows from zero to 3 and 4 columns.**  **We increase the row and columns one by one.** |
|  | for c in range(COLUMN\_COUNT-3): |
|  | for r in range(ROW\_COUNT-3): |
|  | if board[r][c] == piece and board[r+1][c+1] == piece and board[r+2][c+2] == piece and board[r+3][c+3] == piece: |
|  | return True |
|  |  |
|  | **//Check negatively sloped diagonals see if there are four in a diagonal forward**  **And we check that or range 4 columns from 0 to 3 we start the rows from row 3 to be backward.**  **We decrease the row one by one and increase the columns one by one** |
|  | for c in range(COLUMN\_COUNT-3): |
|  | for r in range(3, ROW\_COUNT): |
|  | if board[r][c] == piece and board[r-1][c+1] == piece and board[r-2][c+2] == piece and board[r-3][c+3] == piece: |
|  | return True |
|  |  |
|  | def evaluate\_window(window, piece):  **//we make these function to delete the repetition of the score in each move (vertical , horizontal , diagonal)** |
|  | score = 0 |
|  | opp\_piece = PLAYER\_PIECE |
|  | if piece == PLAYER\_PIECE: |
|  | opp\_piece = AI\_PIECE |
|  |  |
|  | if window.count(piece) == 4: |
|  | score += 100 |
|  | elif window.count(piece) == 3 and window.count(EMPTY) == 1: |
|  | score += 5 |
|  | elif window.count(piece) == 2 and window.count(EMPTY) == 2: |
|  | score += 2 |
|  |  |
|  | if window.count(opp\_piece) == 3 and window.count(EMPTY) == 1: |
|  | score -= 4 |
|  | **// these negative because the against player will win** |
|  | return score |
|  |  |
|  | def score\_position(board, piece): |
|  | score = 0 |
|  |  |
|  | ## Score center column |
|  | center\_array = [int(i) for i in list(board[:, COLUMN\_COUNT//2])] |
|  | center\_count = center\_array.count(piece) |
|  | score += center\_count \* 3 |
|  |  |
|  | ## Score Horizontal |
|  | for r in range(ROW\_COUNT): |
|  | row\_array = [int(i) for i in list(board[r,:])]  **//we make loop to do to all the rows and gave I to go one by one** |
|  | for c in range(COLUMN\_COUNT-3):  **//we make it -3 because in the row we can make 4 pieces so we make column equal column +4 (window length) that mention to four pieces** |
|  | window = row\_array[c:c+WINDOW\_LENGTH] |
|  | score += evaluate\_window(window, piece) |
|  |  |
|  | ## Score Vertical |
|  | for c in range(COLUMN\_COUNT): |
|  | col\_array = [int(i) for i in list(board[:,c])] |
|  | for r in range(ROW\_COUNT-3):  **//we make it -3 because in the col we can make 4 pieces so we make row equal row +4 (window length) that mention to four pieces** |
|  | window = col\_array[r:r+WINDOW\_LENGTH] |
|  | score += evaluate\_window(window, piece) |
|  |  |
|  | ## Score posiive sloped diagonal |
|  | for r in range(ROW\_COUNT-3): |
|  | for c in range(COLUMN\_COUNT-3): |
|  | window = [board[r+i][c+i] for i in range(WINDOW\_LENGTH)]  **//we put I in loop to increase in range zero to 4 as we said for mention to 4 pieces** |
|  | score += evaluate\_window(window, piece) |
|  | ## Score negative sloped diagonal |
|  | for r in range(ROW\_COUNT-3): |
|  | for c in range(COLUMN\_COUNT-3): |
|  | window = [board[r+3-i][c+i] for i in range(WINDOW\_LENGTH)]  **//we make the row +3 because the diagonal is backward so we can start from the beginning we start from the middle and -1 because we go down** |
|  | score += evaluate\_window(window, piece) |
|  |  |
|  | return score |
|  |  |
|  | def is\_terminal\_node(board): **// This function checks if the node is terminal which means that one of them has won or there are no more available spaces and returns the boolean value for this.** |
|  | return winning\_move(board, PLAYER\_PIECE) or winning\_move(board, AI\_PIECE) or len(get\_valid\_locations(board)) == 0 |
|  |  |
|  | def minimax(board, depth, alpha, beta, maximizingPlayer): **//Minimax algorithm function the takes 5 parameters (node, depth, alpha, beta, maximizing player) , in our case the node is our board, the depth is how deep we are in the tree, we initialize alpha with negative infinity and beta with infinity and the Boolean value of maximizingPlayer is the one that allow us to switch turns between the computer and the player** |
|  | valid\_locations = get\_valid\_locations(board) |
|  | is\_terminal = is\_terminal\_node(board) |
|  | if depth == 0 or is\_terminal: |
|  | if is\_terminal**://If this condition is true then we have 3 cases** |
|  | if winning\_move(board, AI\_PIECE): |
|  | return (None, 100000000000000) **//Return a high score if the computer wins** |
|  | elif winning\_move(board, PLAYER\_PIECE): |
|  | return (None, -10000000000000) **//Return a low score if the player wins** |
|  | else: **# Game is over, no more valid moves** |
|  | return (None, 0) |
|  | else: **# If Depth is zero then return the heuristic value of the board** |
|  | return (None, score\_position(board, AI\_PIECE)) |
|  | if maximizingPlayer: |
|  | value = -math.inf **//You set the lowest value because you want to move in the positive direction as a maximizing player** |
|  | column = random.choice(valid\_locations) |
|  | for col in valid\_locations: **//Iterate through all the possible choices** |
|  | row = get\_next\_open\_row(board, col) |
|  | b\_copy = board.copy() |
|  | drop\_piece(b\_copy, row, col, AI\_PIECE) |
|  | new\_score = minimax(b\_copy, depth-1, alpha, beta, False)[1] |
|  | if new\_score > value: |
|  | value = new\_score **//Finding the maximum value among the negative infinity value and the recursive minimax function of the new value and if the new value is closer to our goal we store this value** |
|  | column = col **//The best column we can get** |
|  | alpha = max(alpha, value) |
|  | if alpha >= beta: |
|  | Break |
|  | return column, value |
|  |  |
|  | else: # Minimizing player |
|  | value = math.inf  **//You set the highest value because you want to move in the negative direction as a Minimizing** **player** |
|  | column = random.choice(valid\_locations) |
|  | for col in valid\_locations: |
|  | row = get\_next\_open\_row(board, col) |
|  | b\_copy = board.copy() |
|  | drop\_piece(b\_copy, row, col, PLAYER\_PIECE) |
|  | new\_score = minimax(b\_copy, depth-1, alpha, beta, True)[1] |
|  | if new\_score < value: |
|  | value = new\_score |
|  | column = col |
|  | beta = min(beta, value) |
|  | if alpha >= beta: |
|  | break |
|  | return column, value |
|  | **//Here we search for valid location to drop the piece in it** |
|  | def get\_valid\_locations(board): |
|  | valid\_locations = [] |
|  | for col in range(COLUMN\_COUNT): |
|  | if is\_valid\_location(board, col): |
|  | valid\_locations.append(col) |
|  | return valid\_locations |
|  |  |
|  | def pick\_best\_move(board, piece): |
|  | **//we choose the best move deepen on the score and the valid location** |
|  | valid\_locations = get\_valid\_locations(board) |
|  | best\_score = -10000 |
|  | best\_col = random.choice(valid\_locations)  **//we make the pc choose random location to but in it** |
|  | for col in valid\_locations: |
|  | row = get\_next\_open\_row(board, col)  **//see if the row is open row we can put in it or not** |
|  | temp\_board = board.copy()  **//make a copy of the board**  **to check somethings** |
|  | drop\_piece(temp\_board, row, col, piece) |
|  | score = score\_position(temp\_board, piece)  **//make the score equal score position after we said that score equal zero we give the score another value depend on the number of pieces** |
|  | if score > best\_score:  **//check if the score bigger that the best score to change the score to best score to increase the score and it is help the pc to choose the best move to increase the score** |
|  | best\_score = score |
|  | best\_col = col |
|  |  |
|  | return best\_col |
|  |  |
|  | def draw\_board(board): **// A function that prints the board with the graphics** |
|  | for c in range(COLUMN\_COUNT): **//Iterate through every spot in the board** |
|  | for r in range(ROW\_COUNT): |
|  | pygame.draw.rect(screen, BLUE, (c\*SQUARESIZE, r\*SQUARESIZE+SQUARESIZE, SQUARESIZE, SQUARESIZE)) **// We need to pass 4 values to draw a rectangle (surface, color, rectangle, width) and to define the 3rd value (rectangle) we need the size of width and height and also the position so the position on the x-axis is going to be c\*SQAURESIZE while on the Y-axis it’s r\*SQUARESIZE + SQUARESIZE (so we can leave an empty slot at the top instead of starting at (0,0)), for the width and height they are going to be the same it’s a square** |
|  | pygame.draw.circle(screen, BLACK, (int(c\*SQUARESIZE+SQUARESIZE/2), int(r\*SQUARESIZE+SQUARESIZE+SQUARESIZE/2)), RADIUS) **// Drawing a circle using pygame libraries (scree, color, position, radius)** |
|  |  |
|  | for c in range(COLUMN\_COUNT): |
|  | for r in range(ROW\_COUNT): |
|  | if board[r][c] == PLAYER\_PIECE: |
|  | pygame.draw.circle(screen, RED, (int(c\*SQUARESIZE+SQUARESIZE/2), height-int(r\*SQUARESIZE+SQUARESIZE/2)), RADIUS) |
|  | elif board[r][c] == AI\_PIECE: |
|  | pygame.draw.circle(screen, YELLOW, (int(c\*SQUARESIZE+SQUARESIZE/2), height-int(r\*SQUARESIZE+SQUARESIZE/2)), RADIUS) |
|  | pygame.display.update() |
|  |  |
|  | board = create\_board() |
|  | print\_board(board) |
|  | game\_over = False |
|  | **//Initialize pygame** |
|  | pygame.init() |
|  | **//Define the screen size in pixels** |
|  | SQUARESIZE = 100 |
|  |  |
|  | width = COLUMN\_COUNT \* SQUARESIZE **// The width equals the number of columns multiplied buy the size of the square** |
|  | height = (ROW\_COUNT+1) \* SQUARESIZE **//the height equals the number of rows plus 1 multiplied by the size of the square because the additional row will be for moving the pieces on the top of the screen before dropping them** |
|  |  |
|  | size = (width, height) **//The size is packaged together in a variable called size** |
|  |  |
|  | RADIUS = int(SQUARESIZE/2 - 5) **// The circle should be smaller than the square size so the circles won’t be touching that’s why we subtracted 5 from the radius** |
|  |  |
|  | screen = pygame.display.set\_mode(size) **// To get pygame to read the size** |
|  | draw\_board(board) **// Draw the board** |
|  | pygame.display.update() **// Update the display screen** |
|  |  |
|  | myfont = pygame.font.SysFont("monospace", 75) |
|  |  |
|  | turn = random.randint(PLAYER, AI) |
|  |  |
|  | while not game\_over: |
|  |  |
|  | for event in pygame.event.get(): **//pygame is an event based game library that read all your movements as individual events** |
|  | if event.type == pygame.QUIT: **// allowing the game to exist out by clicking the button exit(X) on the top of the screen** |
|  | sys.exit() **//system exit happens** |
|  |  |
|  | if event.type == pygame.MOUSEMOTION: **//To see the motion of the mouse in the empty slot at the top when we move the piece to drop them in a column** |
|  | pygame.draw.rect(screen, BLACK, (0,0, width, SQUARESIZE)) **//Draw a black rectangle on the top empty slot to blackout any previous circles that had been drown while moving the mouse with piece to drop** |
|  | posx = event.pos[0] |
|  | if turn == PLAYER: |
|  | pygame.draw.circle(screen, RED, (posx, int(SQUARESIZE/2)), RADIUS) |
|  |  |
|  | pygame.display.update() |
|  |  |
|  | if event.type == pygame.MOUSEBUTTONDOWN: **//event that drops a piece by clicking on the mouse on a specific spot** |
|  | pygame.draw.rect(screen, BLACK, (0,0, width, SQUARESIZE)) |
|  | #print(event.pos) |
|  | # Ask for Player 1 Input |
|  | if turn == PLAYER: |
|  | posx = event.pos[0] **//Define the x position of the click which could be a number between 0 and 700 pixels** |
|  | col = int(math.floor(posx/SQUARESIZE)) **//This division will give us a number between 0 and 7 and assign the value to the specified column index.** |
|  |  |
|  | if is\_valid\_location(board, col): |
|  | row = get\_next\_open\_row(board, col) |
|  | drop\_piece(board, row, col, PLAYER\_PIECE) |
|  |  |
|  | if winning\_move(board, PLAYER\_PIECE): |
|  | label = myfont.render("Player 1 wins!!", 1, RED) **// Print the winning statement on the board** |
|  | screen.blit(label, (40,10)) **//Updates this specific part of the screen** |
|  | game\_over = True |
|  |  |
|  | turn += 1 |
|  | turn = turn % 2 |
|  |  |
|  | print\_board(board) |
|  | draw\_board(board) |
|  |  |
|  |  |
|  | # # Ask for Player 2 Input |
|  | if turn == AI and not game\_over: |
|  |  |
|  | #col = random.randint(0, COLUMN\_COUNT-1) |
|  | #col = pick\_best\_move(board, AI\_PIECE) |
|  | col, minimax\_score = minimax(board, 5, -math.inf, math.inf, True) **//We initialize the depth with 5, the alpha with negative infinity, the beta with infinity and the maximizing player with true.** |
|  |  |
|  | if is\_valid\_location(board, col): |
|  | #pygame.time.wait(500) |
|  | row = get\_next\_open\_row(board, col) |
|  | drop\_piece(board, row, col, AI\_PIECE) |
|  |  |
|  | if winning\_move(board, AI\_PIECE): |
|  | label = myfont.render("Player 2 wins!!", 1, YELLOW) |
|  | screen.blit(label, (40,10)) |
|  | game\_over = True |
|  |  |
|  | print\_board(board) |
|  | draw\_board(board) |
|  |  |
|  | turn += 1 |
|  | turn = turn % 2 |
|  |  |
|  | if game\_over: |
|  | pygame.time.wait(3000) **//Wait 3 seconds after the game is over before closing the window** |

**Implementation صورة تحتوي على نص

تم إنشاء الوصف تلقائياً**

