Connect4 – Report

Contents

[Technical Specifications 2](#_Toc107849686)

[Program Goals 2](#_Toc107849687)

[Program license used 2](#_Toc107849688)

[Program Limitations (out of scope) 2](#_Toc107849689)

[Program Features 2](#_Toc107849690)

[Classes 2](#_Toc107849691)

[Functions 2](#_Toc107849692)

[Verifying program behaviour 3](#_Toc107849693)

[Terminal Execution 3](#_Toc107849694)

[Architecture Model 3](#_Toc107849695)

[Code on GitHub 4](#_Toc107849696)

[Terminal Screenshot 4](#_Toc107849697)

[Code 5](#_Toc107849698)

# Technical Specifications

## Program Goals

The goal of this program is to create a digital Connect4 game that can either be played by two players or by one player against a bot. The player can choose from multiple bots with various skill levels to play against. The game is combined with a GUI to enhance the visual game experience.

## Program license used

For this program, we decided to use the MIT License because we want our game code to be available freely so that users can enjoy playing it easily without big licensing hurdles. Additionally, we have fellow developers in mind as well. Choosing this license, we want to maximize code availability so that other coders (for companies) can base further feature modifications/ expansions on it without a source code sharing requirement.

## Program Limitations (out of scope)

This program covers the basic Connect4 rules as well as the visual board experience. Furthermore, it expands the game’s original capabilities from two players required to one player required. Nonetheless, possible functionality expansion options are:

* 3D Connect4 board to make the game more challenging
* Customizable timer to limit the time for a player to make a move
* Sophisticated bot via reinforcement learning
* Selecting the game mode single-player vs double-player via UI instead of terminal
* 2 players playing via 2 different devices

## Program Features

Connect4 is a 2-player game that consists of a 6x7 matrix, and the goal is to drop coins in turns into the matrix until 4 coins of the own colour are in one line. The player who achieves this first has won the game.

To accomplish this game dynamic in player vs player and player vs bot games we used the following classes and functions:

### Classes

Main class = “Connect4” provides all the functionalities to play the game. At instantiation it comes with the following attributes: num\_players (= 1 or 2), bot\_difficulty (= none), pygame (= initiate pygame with the title “Connect4”)

Inheritance class = “Bot(Connect4)” provides all functionalities needed to play against the computer in single-player mode. Inheritance from the class “Connect4” is needed to use the main methods. At instantiation it comes with the following updated attributes: bot\_difficulty (= in single-player mode bot difficulty level is either easy, medium, or hard depending on the user input)

### Functions

All functionalities for a double-player game are inside the class “Connect4” to simplify the user interactions:

* draw\_board()
  + visualizes empty board using rectangles and circles
* check\_move\_allowed()
  + checks if the desired move is allowed/ possible, i.e. the column is not full
* place\_coin()
  + places the coin in the desired column, after the desired move is checked regarding feasibility
* check\_win()
  + checks the winning conditions and draw condition. Is this the case, this ends the game.
* event\_handler()
  + This function handles the UI by iterating through every event in pygame window. It inputs users interactions (mouse motion & mouse click) into the place\_coin() function for each player and checks for winning conditions via the check\_win() function
* draw\_coin()
  + creates a new coin at the beginning of each player's turn. Player 1 receives a red coin. Player 2 receives a yellow coin.
* gameloop()
  + loops through event\_handler() until one of the winning conditions is met and it updates the board until the game is over.

All functionalities for a single-player game are inside the class “Bot(Connect4)” to simplify the user interactions:

* same functions as above plus following functions for bot interactions
* set\_current\_board():
  + creates copy of the board state to be evaluated by the bot for its moves
* random\_move():
  + picks a random column and places a coin
* place\_4th\_coin():
  + checks if there is a winning move for either player and places the coin in that position (priority on own winning move)
* check\_mistake():
  + chucks if a move would create a new winning opportunity for the opponent
* bot\_move():
  + uses the current state of the board and the difficulty level selected to return a column which represents the move of the bot player

## Verifying program behaviour

Because we are using a GUI, we have purposely limited the user’s interactions to mouse clicks on the game board. Consequently, only if the user clicks in the intended areas (above the columns to place the coin in the respective column) a reaction will be triggered. In all other cases, the program waits for the correct user input.

## Terminal Execution

1. Upon file execution, the user will need to pass the argument “-Number of Players” and “-bot level” (only required for single player) indicating the number of players that want to play and the difficulty level of the bot in case 1 player is playing. This is done via argparse.
2. The GUI with an empty board starts automatically and requests player 1 to place a coin.
3. The game gets executed on the GUI.
4. Upon game completion the GUI can be closed.

# Architecture Model

It is optional – should we do it?

# Code on GitHub

Link to GitHub repository: <https://github.com/Urbinator/AdvPy_Connect4>

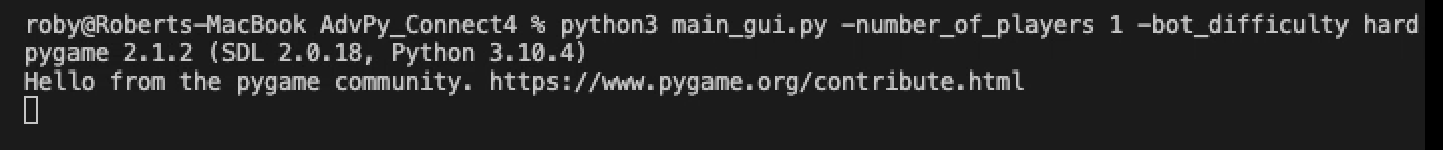
Last valid Commit ID: 3fb970a27a79ec43b6e355d729e6fd8db1684082

Author: Urbinator

Date: 03.07.2022

Comment: Update requirements

# Terminal Screenshot



Background pattern

Description automatically generated with medium confidence

# Code

|  |
| --- |
| ## main\_gui.py |
| # main script to play with GUI |
|  |
| # MIT License |
| # |
| # Copyright (c) 2022 Simon Urban, Robert Bernasconi, Casper Kirch, Jose Marques, Akshit Gupta |
| # |
| # Permission is hereby granted, free of charge, to any person obtaining a copy |
| # of this software and associated documentation files (the "Software"), to deal |
| # in the Software without restriction, including without limitation the rights |
| # to use, copy, modify, merge, publish, distribute, sublicense, and/or sell |
| # copies of the Software, and to permit persons to whom the Software is |
| # furnished to do so, subject to the following conditions: |
|  |
| # The above copyright notice and this permission notice shall be included in all |
| # copies or substantial portions of the Software. |
|  |
| # THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR |
| # IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, |
| # FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE |
| # AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER |
| # LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, |
| # OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE |
| # SOFTWARE. |
|  |
| import argparse |
| import numpy as np |
| import pygame |
| from time import sleep |
| import math |
|  |
| #GLOBAL VARIABLES |
| BLUE = (0, 0, 255) |
| RED = (255, 0, 0) |
| YELLOW = (255, 255, 0) |
| BLACK = (0, 0, 0) |
|  |
| num\_columns = 7 |
| num\_rows = 6 |
| SQUARESIZE = 100 |
| RADIUS = int(SQUARESIZE / 2 - 5) |
|  |
| width = num\_columns \* SQUARESIZE |
| height = (num\_rows + 1) \* SQUARESIZE |
| size = (width, height) |
| #GLOBAL VARIABLES END |
|  |
| class Connect4: |
| '''Main Connect 4 class with all functionalities needed |
| Attributes |
| ----------- |
| screen: A UI screen for the users to interact upon |
| \_\_container: a hidden parameter that contains the up to date values of the game. |
| turn: it's an int (0) |
| myfont: defines UI font and font size |
| game\_over: initialy set to False |
| num\_players: number of players (1 or 2) |
| bot\_difficulty: if in single player mode, it is the both difficulty level (easy,medium,hard) |
|  |
| Methods |
| ---------- |
| draw\_board() |
| visualizes empty board |
| check\_move\_allowed() |
| checks if the desired move is allowed/ possible |
| place\_coin() |
| places the coin in the desired column, after the desired move is checked regarding feasibility |
| check\_win() |
| checks the winning conditions |
| event\_handler() |
| inputs users interactions into the place\_coin() function for each player and checks for winning condition via the check\_win() function |
| draw\_coin() |
| creates a new coin at the beginning of each player's turn |
| gameloop() |
| loops through event\_handler() until winning condition is met |
|  |
| ''' |
| def \_\_init\_\_(self,num\_players: int,bot\_difficulty: str = None) -> None: |
| ''' |
| Initialization function |
| num\_players: number of players (1 or 2) |
| bot\_difficulty: if in single player mode, it is the both difficulty level (easy,medium,hard) |
| pygame: initiate game with a title "CONNECT 4" |
| return: None |
| ''' |
| pygame.init() |
| pygame.display.set\_caption('CONNECT 4') |
| self.screen = pygame.display.set\_mode(size) |
| self.\_\_container = np.zeros((6, 7)) |
| self.turn = 0 |
| self.myfont = pygame.font.SysFont("comicsansms", 75) |
| self.game\_over = False |
| self.num\_players = num\_players |
| self.bot\_difficulty = bot\_difficulty |
|  |
| def draw\_board(self): |
| ''' |
| Draws empty board using rectangles and circles |
| ''' |
| for c in range(num\_columns): |
| for r in range(num\_rows): |
| pygame.draw.rect(self.screen, BLUE, |
| (c \* SQUARESIZE, r \* SQUARESIZE + SQUARESIZE, SQUARESIZE, SQUARESIZE)) |
| pygame.draw.circle(self.screen, BLACK, ( |
| int(c \* SQUARESIZE + SQUARESIZE / 2), int(r \* SQUARESIZE + SQUARESIZE + SQUARESIZE / 2)), RADIUS) |
|  |
| for c in range(num\_columns): |
| for r in range(num\_rows): |
| if self.\_\_container[r][c] == 1: |
| pygame.draw.circle(self.screen, RED, ( |
| int(c \* SQUARESIZE + SQUARESIZE / 2), int(r \* SQUARESIZE + SQUARESIZE + SQUARESIZE / 2)), |
| RADIUS) |
| elif self.\_\_container[r][c] == 2: |
| pygame.draw.circle(self.screen, YELLOW, ( |
| int(c \* SQUARESIZE + SQUARESIZE / 2), int(r \* SQUARESIZE + SQUARESIZE + SQUARESIZE / 2)), |
| RADIUS) |
| pygame.display.update() |
|  |
| def check\_move\_allowed(self, column: int) -> bool: |
| ''' |
| This hidden function checks if the move is allowed, i.e. the column is not full |
| column: an int from 0 to 5 representing the column where to check if is not full |
| returns: True if the move is allowed |
| ''' |
| return self.\_\_container[0, column] == 0 |
|  |
| def place\_coin(self, column: int, player: int) -> bool: |
| ''' |
| This function checks if a move is allowed and place the coin in the selected column. |
| In the selected column it checks from the bottom which is the first empty place to place it. |
| column: an int from 0 to 5 representing the column where to place the coin |
| player: an int (0 or 1) representing the current player making the move |
| returns: True if the move was allowed and the board is updated |
| ''' |
| res = self.check\_move\_allowed(column) |
| if res: |
| for row in reversed(range(len(self.\_\_container[:, column]))): |
| if self.\_\_container[row, column] == 0: |
| break |
| self.\_\_container[row, column] = player # Here the coin is inserted |
| return res |
|  |
| def check\_win(self, player: int) -> int: |
| ''' |
| This function evaluates whether a move ends the game (either winning/draw) |
| ''' |
| # Check horizontal locations for win |
| for c in range(7 - 3): |
| for r in range(6): |
| if self.\_\_container[r][c] == self.\_\_container[r][c + 1] == self.\_\_container[r][c + 2] == \ |
| self.\_\_container[r][c + 3] == player: |
| return player |
| # Check for vertical win |
| for c in range(7): |
| for r in range(6 - 3): |
| if self.\_\_container[r][c] == self.\_\_container[r + 1][c] == self.\_\_container[r + 2][c] == \ |
| self.\_\_container[r + 3][c] == player: |
| return player |
| # Check for decreasing diagonal win (slope <0) |
| for c in range(7 - 3): |
| for r in range(6 - 3): |
| if self.\_\_container[r][c] == self.\_\_container[r + 1][c + 1] == self.\_\_container[r + 2][c + 2] == \ |
| self.\_\_container[r + 3][c + 3] == player: |
| return player |
| # Check for increasing diagonal win (slope >0) |
| for c in range(7 - 3): |
| for r in range(3, 6): |
| if self.\_\_container[r][c] == self.\_\_container[r - 1][c + 1] == self.\_\_container[r - 2][c + 2] == \ |
| self.\_\_container[r - 3][c + 3] == player: |
| return player |
| # Check if draw |
| if np.all(self.\_\_container): |
| return 3 |
|  |
| def event\_handler(self): |
| ''' |
| This function handles the UI by iterating through every event in pygame window |
| MOUSEMOTION - handles the coin hovering above the board before being played |
| MOUSEBUTTONDOWN - handles playing the coin on the board |
| Checks for winning move using check\_win |
| ''' |
| for event in pygame.event.get(): |
| if event.type == pygame.QUIT: |
| pygame.quit() |
| exit() |
| if event.type == pygame.MOUSEMOTION: |
| self.draw\_coin(event) |
| if event.type == pygame.MOUSEBUTTONDOWN: |
| pygame.draw.rect(self.screen, BLACK, (0, 0, width, SQUARESIZE)) |
| if self.turn == 0: |
| # Ask for Player 1 Input |
| posx = event.pos[0] |
| col = int(math.floor(posx / SQUARESIZE)) |
| if self.place\_coin(col, 1): |
| if self.check\_win(1) == 1: |
| label = self.myfont.render("Player 1 wins!!", 1, RED) |
| self.screen.blit(label, (110,0)) |
| self.game\_over = True |
| if self.check\_win(1) == 3: |
| label = self.myfont.render("It's a Draw!!", 1, BLUE) |
| self.screen.blit(label, (110,0)) |
| self.game\_over = True |
| else: |
| break |
| else: |
| # Ask for Player 2 Input |
| posx = event.pos[0] |
| col = int(math.floor(posx / SQUARESIZE)) |
| if self.place\_coin(col, 2): |
| if self.check\_win(2) == 2: |
| label = self.myfont.render("Player 2 wins!!", 1, YELLOW) |
| self.screen.blit(label, (110,0)) |
| self.game\_over = True |
| if self.check\_win(1) == 3: |
| label = self.myfont.render("It's a Draw!!", 1, RED) |
| self.screen.blit(label, (110,0)) |
| self.game\_over = True |
| else: |
| break |
|  |
| # if 2 players change turn, if 1 make the bot move |
| if self.num\_players==2: # Change turn if 2 players |
| self.turn += 1 |
| self.turn = self.turn % 2 |
| else: |
| if not self.game\_over: |
| # Bot input |
| bot = Bot(self.bot\_difficulty) #instantiate bot object |
| col = bot.bot\_move(self.\_\_container) |
| if self.place\_coin(col,2): |
| if self.check\_win(2) == 2: |
| label = self.myfont.render("Player 2 wins!!", 1, YELLOW) |
| self.screen.blit(label, (110,0)) |
| self.game\_over = True |
| if self.check\_win(1) == 3: |
| label = self.myfont.render("It's a Draw!!", 1, RED) |
| self.screen.blit(label, (110,0)) |
| self.game\_over = True |
| self.draw\_board() |
| if self.game\_over: |
| pygame.time.wait(3000) |
|  |
| def draw\_coin(self, event): |
| ''' |
| Creates a new coin for the next player to use. |
| Player 1 receives a red coin. Player 2 receives a yellow coin. |
| ''' |
| pygame.draw.rect(self.screen, BLACK, (0, 0, width, SQUARESIZE)) |
| posx = event.pos[0] |
| if self.turn == 0: |
| pygame.draw.circle(self.screen, RED, (posx, int(SQUARESIZE / 2)), RADIUS) |
| else: |
| pygame.draw.circle(self.screen, YELLOW, (posx, int(SQUARESIZE / 2)), RADIUS) |
| pygame.display.update() |
|  |
| def gameloop(self): |
| ''' |
| A loop that lets the players continue to make their next move and updates the board until the game is over. |
| ''' |
| while not self.game\_over: |
| self.event\_handler() |
| pygame.display.update() |
|  |
| class Bot(Connect4): |
| ''' |
| Bot class with all functionalities needed to play against the computer in single player mode. |
| Inheritance from the class Connect4 is needed to use the main methods |
| Attributes |
| ----------- |
| All the ones of inherithed from Connect4 class (only \_Connect4\_\_container and bot\_difficulty are used) |
|  |
| Methods |
| ---------- |
| All the ones of inherithed from Connect4 class (only place\_coin,check\_win and check\_move\_allowed are used) |
| set\_current\_board: creates copy of the board state to be evaluated by the bot for its moves |
| random\_move: picks a random column and place a coin |
| place\_4th\_coin: check if there is a winning move for either player and place the coin in that position (priority to own winning move) |
| check\_mistake: check if a move would create a new winning opportunity for the opponent |
| bot\_move: uses the current state of the board and the difficulty level selected to return a column which represent the move for the Bot player |
| ''' |
|  |
| def \_\_init\_\_(self,difficulty: str) -> None: |
| ''' |
| Initialization function |
| Inherits attributes and methods from Connect4 class |
| Also allow to set the bot\_difficulty (which is none by default in the parent class) |
| ''' |
| super().\_\_init\_\_(1,difficulty) |
|  |
| def set\_current\_board(self, curr\_board: object): |
| ''' |
| This function set the copy of the board state to be evaluated by the bot for its moves |
| board: the matrix containing the current state of the board |
| ''' |
| self.\_Connect4\_\_container = curr\_board.copy() |
|  |
| def random\_move(self,curr\_board: object) -> int: |
| ''' |
| This function picks a random column and place a coin. |
| board: the matrix containing the current state of the board |
| Return: an int representing the column |
| ''' |
| self.set\_current\_board(curr\_board) |
| while True: |
| col = np.random.randint(0,7) |
| if self.place\_coin(col,2): |
| return col |
|  |
| def place\_4th\_coin(self,curr\_board: object) -> int: |
| ''' |
| This function check if there is a winning move for either player and place the coin in that position. |
| It prioritize a winning move over stopping the win of the other player. |
| board: the matrix containing the current state of the board |
| Return: an int representing the column |
| ''' |
| self.set\_current\_board(curr\_board) |
| for player in reversed(range(1,3)): |
| for col in range(7): |
| if self.place\_coin(col,player): |
| if self.check\_win(player) == player: |
| return col |
| self.set\_current\_board(curr\_board) |
|  |
| def check\_mistake(self,column: int,curr\_board: object) -> bool: |
| '''This function check if a move would create a new winning opportunity for the opponent |
| column: int representing the column of the move to check |
| board: the matrix containing the current state of the board |
| Return: an bool indicating if the move is a mistake |
| ''' |
| self.set\_current\_board(curr\_board) |
| if self.place\_coin(column,2): |
| updated\_board = self.\_Connect4\_\_container #current\_board updated with the potential move/mistake |
| for col in range(7): |
| if self.place\_coin(col,1): |
| if self.check\_win(1) == 1: |
| return True |
| self.set\_current\_board(updated\_board) |
|  |
| def bot\_move(self,curr\_board: object): |
| ''' |
| This function uses the current state of the board and the difficulty level selected to make a move for the Bot player |
| board: the matrix containing the current state of the board |
| ''' |
| if self.bot\_difficulty == 'easy': #random move |
| return self.random\_move(curr\_board) |
| elif self.bot\_difficulty == 'moderate': #identify 3 in a row and place coin in the forth space (priority to winning move, then to stop the other's win), otherwise random move |
| res = self.place\_4th\_coin(curr\_board) |
| if res != None: |
| return res |
| else: |
| return self.random\_move(curr\_board) |
| elif self.bot\_difficulty == 'hard': #same as 'moderate' but it also check if a move is a mistake (mistake: the move creat a new winning opportunity for the other player) |
| res = self.place\_4th\_coin(curr\_board) |
| if res != None: |
| return res |
| else: |
| # Check if there are moves that are not mistakes |
| available\_moves = 0 |
| for col in range(7): |
| if self.check\_move\_allowed(col): |
| if not self.check\_mistake(col,curr\_board): |
| available\_moves += 1 |
| # If there are: select randomly |
| if available\_moves > 0: |
| while True: |
| rand\_move = self.random\_move(curr\_board) |
| if not self.check\_mistake(rand\_move,curr\_board): |
| return rand\_move |
| # If there aren't moves that are not mistakes: place the mistake anyway (to avoid entering an infinite loop and halt the game) |
| else: |
| return self.random\_move(curr\_board) |
|  |
|  |
| if \_\_name\_\_ == "\_\_main\_\_": |
| #Arguments to pass |
| parser = argparse.ArgumentParser( |
| description="This program takes 2 arguments:\n1)Number of Players\n2)The bot level (only required for single player)") |
| parser.add\_argument('-number\_of\_players','-num', type=int, required=True, help='Select number of Players: 1 or 2') |
| parser.add\_argument('-bot\_difficulty','-diff', choices=['easy', 'moderate', 'hard'], required=False, help='Select bot difficulty in single player') |
| argument\_list = parser.parse\_args() |
| # Make the -bot\_difficulty argument necessary only if 2 players are selected |
| if argument\_list.number\_of\_players == 1 and argument\_list.bot\_difficulty is None: |
| parser.error("-number\_of\_players = 1 requires -bot\_difficulty/-diff") |
|  |
| # Start Game |
| game = Connect4(argument\_list.number\_of\_players,argument\_list.bot\_difficulty) |
| game.gameloop() |