

**POCKET**

**CHESS ARENA**

(Computer Science Project 2022)

By:

Gurumurthy V (9)

Sriram Srinivasan (24)

Subham Patra (25)

**CONTENTS**

| Serial no. | Title | Page no. |
| --- | --- | --- |
| 1 | Introduction | 3 |
| 2 | System requirements | 5 |
| 3 | Basic Algorithm | 6 |
| 4 | Modules and Library functions | 8 |
| 5 | User Defined Functions | 9 |
| 6 | Variables | 13 |
| 7 | Source Code | 14 |
| 8 | Screenshots | 160 |
| 9 | Bibliography | 166 |

**INTRODUCTION**

Pocket Chess Arena, a cross-platform desktop application, enables you to play casual unrated games with your friends offline or online and also revisit past games. The objective was to develop a pocket-sized application to help users quickly strike up a game with a friend in the midst of using another application.

The application was built using tkinter, turtle and MySQL.

**SYSTEM REQUIREMENTS**

* Software Requirements
  + At least 16-bit 100 MB RAM required
  + Python 3.6+
* Hardware Requirements
  + Monitor with at least 40 Hz refresh rate
  + 1 MB disk space

**BASIC ALGORITHM**

* GUI
  + Process mouse clicks and convert them to possible chess moves
  + Validate moves and move the piece from the starting square to the ending square
  + Check for check, checkmate and stalemate and display them in the GUI
* Logic

| **2D List Index** | | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **2D List Index** | **Chess**  **Naming** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** |  |
| **0** | **8** | **blrook1** | **blhor1** | **blbish1** | **blqueen** | **blking** | **blbish2** | **blhor2** | **blrook2** | **8** |
| **1** | **7** | **p1** | **p2** | **p3** | **p4** | **p5** | **p6** | **p7** | **p8** | **7** |
| **2** | **6** |  |  |  |  |  |  |  |  | **6** |
| **3** | **5** |  |  |  |  |  |  |  |  | **5** |
| **4** | **4** |  |  |  |  |  |  |  |  | **4** |
| **5** | **3** |  |  |  |  |  |  |  |  | **3** |
| **6** | **2** | **P1** | **P2** | **P3** | **P4** | **P5** | **P6** | **P7** | **P8** | **2** |
| **7** | **1** | **whrook1** | **whhor1** | **whbish1** | **whqueen** | **whking** | **whbish2** | **whhor2** | **whrook2** | **1** |
|  |  | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** |  |

* + piecevariable= [identity, symbol, rownumber, columnnumber]
    - \* in the symbol indicates white and its absence indicates black.
    - Eg:
      * whrook1 = [“rook”, “\*][“, 7, 0]
      * whqueen=["queen", "\*Q", 7, 3]
      * blhor1 =["horse", "/>", 0, 1]
  + list2d is a highly dynamic 2 dimensional list containing all the piecevariables. Every time a move takes place, the following changes are made:
    - list2d is modified
    - rownumber and the columnnumber of the moving piece is modified.
* Networking
  + Star topology
  + Rooms (that have a unique room code) are created for the players(black and white players) to play with each other.
  + Rooms are stored in the server as hashmap (dictionary) i.e. {<room code> : <room info>}
  + The server is basically a website publicly hosted but not accessible via a browser, where the exchange of data is done by making get and post requests from the client program.
* Threading
  + Multithreading is used for online mode to make requests to the server.
  + Updation of the timers on the screen, by comparing it to the change in time.time() which returns the value of time in seconds since 0 hours, 0 minutes, 0 seconds, January 1, 1970.

**MODULES AND LIBRARY FUNCTIONS**

* Graphics libraries
  + tkinter=8.6
  + turtle=0.3.2
  + ctypes=1.0.2 (for Windows)
* Networking libraries
  + flask=2.0.1
  + requests=2.26.0
* Database libraries
  + MySQL
* Automated Testing modules
  + pytest
* Utility modules
  + time
  + os
  + random
  + functools
  + clipboard=0.0.4 (for windows)
  + xclip (bash command for unix based operating systems)
  + datetime=4.4
  + playsound=1.2.2
  + threading
  + config parser
  + colorsys

**USER-DEFINED FUNCTIONS**

*home.py*

| Function Name | Task |
| --- | --- |
| main\_menu, settings, setup\_game, about | GUI Navigation |
| send\_challenge\_request, accept\_challenge\_request | Exchange of initialisation data with server (online) |
| enable\_game\_controls, disable\_game\_controls | Enables/Disables all game controls: board colour, volume, flip board, legal moves |
| on\_scale\_board\_release, on\_pin\_toggle, on\_color\_chooser, on\_flip, on\_volume\_toggle, on\_legal\_moves | Handling configuration/game controls |
| on\_white\_resign, on\_black\_resign, on \_draw | Handles white resign, black resign and draw agreement (offline) or draw offer (online) |
| on\_extend\_left, on\_extend\_right | Expands and Collapses the side wings of the game |
| on\_code\_copy | Copy room code to clipboard (online) |
| show\_labels | Displays the chess coordinates on the board |

*game.py*

| Function Name | Task |
| --- | --- |
| configure\_game | Displays the board and the pieces |
| change\_square\_clr | Changes the colour of a specific board square |
| show\_legal\_squares | Highlights all possible legal squares a piece can go to |
| display\_check\_and\_checkmate, display\_stalemate | Updates the GUI in times of a check, checkmate and stalemate respectively |
| add\_to\_db | Updates game data to database based on user’s choice |
| on\_white\_flag, on\_black\_flag | Handles the situation when the white or black player run out of time respectively (online) |
| main\_updater\_offline, main\_updater\_online | The main game loop (while loop) |
| process\_response | This parses the get request made to the server |
| connection | Constantly makes requests to inform the server about the status of the client internet |
| generate\_pgn | Generates a portable game notation for analysis with engines online(websites) |
| get\_move, process\_move, make\_move, reset\_move | Catches mouse clicks, processes the move made, displays the move in the GUI and resets certain logical/GUI elements once the move is made |
| promote\_pawn | Handles pawn promotion |
| hide\_dead\_pieces | Hides all attacked pieces |

*replay.py*

| save\_configuration, show\_saved\_configurations | Save a board configuration and add notes, show all the saved configurations with the added notes |
| --- | --- |
| list\_of\_games\_played\_page, setup\_replay | GUI Navigation for list of games played page and setting up the replay |
| update\_timers | Updates the timers displayed |
| previous\_move, next\_move | Makes GUI changes to go to the next move and back to the previous move |
| configure\_game | Displays the board and the pieces |

*logic.py*

| rookmove, hormove, bishmove, queenmove, kingmove, pawnmove | Validating moves of different pieces |
| --- | --- |
| move | Changes list2d after a move is validated |
| sameteam | Checks if two pieces belong to the same team |
| piecesbetween | Returns a list of all the pieces between two squares on the board including the piece on the final square |
| castle | Validates castling |
| legal | Returns all the legal squares a piece can go to |
| check, checkmate, stalemate | Detects check, checkmate and stalemate respectively |
| gameprocessing | Central processor in logic.py of move details obtained from game.py |
| pawnpromotion | Makes changes in list2d when a pawn promotes |
| winner\_on\_flag | Calculates game result when a player gets flagged |

*database\_functions.py*

| update\_game\_details | Updates all the game details of a single game |
| --- | --- |
| receive\_game\_details | Received the game details of a particular game |
| receive\_all\_game\_details | Receiving all game details |
| update\_configuration\_saved | Updates the note of a specific configuration in a game |
| delete\_configuration | Deletes the note for a specific configuration |
| receive\_configurations\_saved | Receive all configurations saved in a particular game |

*utils.py*

| tfor | Formats the seconds into a human-readable string, eg. 62 sec -> 00:01:02 |
| --- | --- |
| get\_chess\_notation | Returns the chess notation of a move |
| sounds | Playing the sound based on the the move type |
| adjust\_color\_lightness, lighten\_color, darken\_color | Processes a colour and lightens or darkens it respectively |
| hex\_to\_rgb, rgb\_to\_hex | Interconverts hex code and RGB for colours |

*server.py* (synchronous functions)

| clean | Deletes the room after the game is over |
| --- | --- |
| generate\_code | Generates a unique room code as a combination of letters and numbers |
| handle\_room | A while loop that handles the room. |

*server.py* (asynchronous functions)

| on\_connect | Sends a unique room code to the client |
| --- | --- |
| host | Hosts the website and exchanges the moves |
| on\_connection\_recv | Processes the constant connection made by the client |
| debug\_ | To debug |

**VARIABLES**

*home.py:*

root\_turtle, canvas\_turtle, wn, mode\_of\_play, canvas\_main\_menu, url, player\_side, white\_name, black\_name, touch\_move, challenger, code, program\_running, theme\_bg

*game.py*

size, white, black, legalmoves, touch\_move, turn, move\_stage, wh\_list\_piece\_numbers, bl\_list\_piece\_numbers, wh\_piece\_count, bl\_piece\_count, legaladd, active\_piece, move\_duration, game\_result, pgn, move\_number

*replay.py*

size, drift*, verticaldrift*, boardview, replay\_list2d, list\_piece\_numbers, piece\_count, move\_count, replay\_match\_number, moves, times, min\_time, increment, times\_white, times\_black

*database\_functions.py*

chessdb, mycur

*spear.py*

server, rooms, timers, inncheck, move\_durations, offers, connection, debug

*constant.py*

COORD, pieces, LIGHTSQUARECLR, DARKSQUARECLR, CHECKSQUARECLR, SELECTEDLIGHTSQUARECLR, SELECTEDDARKSQUARECLR, LEGALLIGHTSQUARECLR, LEGALDARKSQUARECLR, BLACKPIECECLR, WHITEPIECECLR, PAWNPROMOTIONWINDOWCLR, DARKBGCLR, LIGHTBGCLR, DARKBGTEXTCLR, LIGHTBGTEXTCLR, ACTIVEBGCLR, ACTIVEFGCLR, PRIMARYCLR, SECONDARYCLR, TERTIARYCLR, DRIFT\_WV\_BOARD, DRIFT\_BOARD\_END\_OF\_GAME, ABOUT\_INFO

**SOURCE CODE**

home.py

#Import of Modules

import turtle

import time

from tkinter import \*

from tkinter import ttk

from tkinter import font

from tkinter import colorchooser

from tkinter import messagebox

import random

import requests

import configparser

from threading import Thread

import clipboard

import ctypes

import os

#Import of created files

import game

import logic

import replay

import constant

import database\_functions

from utils import Tooltip, CustomButton, rgb\_to\_hex, ScrolledFrame, tfor

#General Purpose Function: Used by both game.py and replay.py

def show\_labels(x,y,division, divisionstart, reverse, delete=False): #origin coordinates and space in between each character

    wn.tracer(0)

    if reverse==False:

        letters = [chr(\_) for \_ in range(65,65+8)]

        numbers = [\_ for \_ in range(1,9)]

    elif reverse==True:

        letters = [chr(\_) for \_ in range(65+7,64,-1)]

        numbers = [\_ for \_ in range(8,0,-1)]

    if reverse==False:

        lt = "w"

    else:

        lt = "b"

    if delete == True:

        for letter in letters:

            globals()[f'{lt}{letter}'].clear()

            del globals()[f'{lt}{letter}']

        for number in numbers:

            globals()[f'{lt}l{number}'].clear()

            del globals()[f'{lt}l{number}']

        return None

    letter\_coor=[None]

    number\_coor=[None]

    letter\_coor = [((x + divisionstart) + a\*division, y) for a in range(0,8)]

    number\_coor = [(x, (y + divisionstart) + a\*division) for a in range(0,8)]

    for letter,coor in zip(letters,letter\_coor):

        globals()[f'{lt}{letter}'] = turtle.Turtle()

        globals()[f'{lt}{letter}'].ht()

        globals()[f'{lt}{letter}'].up()

        globals()[f'{lt}{letter}'].pencolor(constant.DARKBGTEXTCLR)

        globals()[f'{lt}{letter}'].goto(coor[0]+5,coor[1])

        globals()[f'{lt}{letter}'].write(letter,font=("Comic Sans", 10, "bold"))

    for number,coor in zip(numbers,number\_coor):

        globals()[f'{lt}l{number}'] = turtle.Turtle()

        globals()[f'{lt}l{number}'].ht()

        globals()[f'{lt}l{number}'].up()

        globals()[f'{lt}l{number}'].pencolor(constant.DARKBGTEXTCLR)

        globals()[f'{lt}l{number}'].goto(coor[0],coor[1])

        globals()[f'{lt}l{number}'].write(str(number),font=("Comic Sans", 10, "bold"))

    wn.update()

def main\_menu():

    global canvas\_turtle, img\_logo, img\_history, img\_play\_offline, img\_play\_online, img\_toggle\_on, img\_toggle\_off, canvas\_main\_menu, mode\_of\_play

    def on\_play\_over\_board():

        global mode\_of\_play

        mode\_of\_play = "offline"

        settings()

    def on\_play\_with\_friends():

        global mode\_of\_play

        mode\_of\_play = "online"

        settings()

    def on\_visit\_past\_games():

        global mode\_of\_play, canvas\_main\_menu

        mode\_of\_play = None

        destroy\_all\_widgets\_in\_canvas\_main\_menu()

        canvas\_main\_menu.destroy()

        if database\_functions.check\_connection()==False:

            #Opening Database function

            database\_functions.open\_connection()

        if database\_functions.check\_connection()==True:

            root\_turtle.grid\_columnconfigure((0,1,2,3), weight = 0)

            root\_turtle.grid\_rowconfigure((0,1,2,3), weight = 0)

            replay.list\_of\_games\_played\_page()

        else:

            messagebox.showerror("Error", "Check your internet connection and try again.")

    def on\_about():

        about()

    def on\_turtle\_close():

        global program\_running

        program\_running = False

        root\_turtle.destroy()

        database\_functions.close\_connection()

    root\_turtle.protocol("WM\_DELETE\_WINDOW", on\_turtle\_close)

    #canvas\_turtle has been placed at the location below temporarily. It will get replaced by canvas\_left. If canvas\_turtle is NOT placed anywhere on the screen, turtle will throw an error. So, we are actually fooling turtle.

    canvas\_turtle.grid(row = 0, column = 0)

    #Gridding root\_turtle

    root\_turtle.grid\_columnconfigure((0,1,2,3), weight = 0)

    root\_turtle.grid\_rowconfigure((0,1,2,3), weight = 0)

    root\_turtle.grid\_rowconfigure(0, weight = 1)

    root\_turtle.grid\_columnconfigure(0, weight = 1)

    #Creating the main menu window

    canvas\_main\_menu = Canvas(root\_turtle, bg = constant.DARKBGCLR, bd = 0, highlightthickness = 0)

    canvas\_main\_menu.grid(row = 0, column = 0, sticky = NSEW, ipadx = 5, ipady = 5)

    #Gridding canvas\_main\_menu

    canvas\_main\_menu.grid\_columnconfigure((0,1), weight = 1)

    canvas\_main\_menu.grid\_rowconfigure((0,1,2), weight = 1)

    #Adding the pocket chess arena logo

    img\_logo = PhotoImage(file = "./Icons/pocket\_chess\_arena\_logo.png").subsample(3, 3)

    label\_logo = Label(canvas\_main\_menu, image = img\_logo, text = "POCKET\nCHESS  ARENA", compound = TOP, font = ('Comic Sans', 28, 'bold'), fg = constant.LIGHTBGTEXTCLR, bg = constant.LIGHTBGCLR, bd = 2, highlightthickness = 2, justify = CENTER)

    label\_logo.grid(row = 0, column = 0, rowspan = 4, sticky = NSEW, ipadx = 10, ipady = 10)

    #Creating the icons for the buttons/toggles

    img\_history = PhotoImage(file = "./Icons/history.png").subsample(6,6)

    img\_play\_offline = PhotoImage(file = "./Icons/play\_offline.png").subsample(6,6)

    img\_play\_online = PhotoImage(file = "./Icons/play\_online.png").subsample(6,6)

    img\_toggle\_on = PhotoImage(file = "./Icons/on.png")

    img\_toggle\_off = PhotoImage(file = "./Icons/off.png")

    #Creating all the buttons inside

    button\_play\_offline = CustomButton(canvas\_main\_menu, text = "    PLAY OVER THE BOARD", image = img\_play\_offline, compound = TOP, font = ('Comic Sans', 18, 'bold'), bg = constant.PRIMARYCLR, fg = constant.LIGHTBGTEXTCLR, relief = "ridge", justify = "center", command = on\_play\_over\_board)

    button\_play\_online = CustomButton(canvas\_main\_menu, text = "    PLAY A FRIEND ONLINE", image = img\_play\_online, compound = TOP,font = ('Comic Sans', 18, 'bold'), bg = constant.PRIMARYCLR, fg = constant.LIGHTBGTEXTCLR, relief = "ridge", justify = "center", command = on\_play\_with\_friends)

    button\_visit\_past\_games = CustomButton(canvas\_main\_menu, text = "    VISIT PAST GAMES    ", image = img\_history, compound = TOP, font = ('Comic Sans', 18, 'bold'), bg = constant.PRIMARYCLR, fg = constant.LIGHTBGTEXTCLR, relief = "ridge", justify = "center", command = on\_visit\_past\_games)

    button\_about = CustomButton(canvas\_main\_menu, text = "ABOUT", font = ('Comic Sans', 12, 'bold'), bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, bd = 0, highlightthickness = 0, relief = "ridge", command = on\_about)

    #Gridding all the buttons

    button\_play\_offline.grid(row = 0, column = 1, sticky = NSEW, ipadx = 10, ipady = 10, padx = 40, pady = (30, 20))

    button\_play\_online.grid(row = 1, column = 1, sticky = NSEW, ipadx = 10, ipady = 10, padx = 40, pady = 20)

    button\_visit\_past\_games.grid(row = 2, column = 1, sticky = NSEW, ipadx = 10, ipady = 10, padx = 40, pady = 20)

    button\_about.grid(row = 3, column = 1, sticky = EW, ipadx = 5, ipady = 5, padx = 40, pady = 5)

def destroy\_all\_widgets\_in\_canvas\_main\_menu():

    global canvas\_main\_menu

    for widget in canvas\_main\_menu.winfo\_children():

        try:

            for child\_widget in widget.winfo\_children():

                child\_widget.destroy()

        except:

            pass

        widget.destroy()

def send\_challenge\_request(code): #Challenger

    global url, oppname, received , oppside, black\_name , white\_name

    info = {

        'code': code,

        'name': player\_name,

        'side': player\_side,

        'mintime': min\_time,

        'inc': increment,

        'tm' : touch\_move,

        'status':'waiting'

    }

    r = requests.post(geturl('rooms'), data = info)

    while True:

        try:

            r = requests.get(geturl('rooms') , params={'code':code})

            r = r.json()

            if r['status'] == 'running':

                received = True

                break

        except Exception as e:

            print(type(e) , e)

            pass

        time.sleep(0.2)

    oppname = r['name']

    oppside = 'white' if player\_side == 'black' else 'black'

    if player\_side == 'white':

        white\_name = player\_name

        black\_name = oppname

        root\_turtle.title(f'{code}: {player\_name} vs {oppname}')

    else:

        white\_name = oppname

        black\_name = player\_name

        root\_turtle.title(f'{code}: {oppname} vs {player\_name}')

def accept\_challenge\_request(code): #Joinee

    global accepted, oppname, oppside, touch\_move, min\_time, increment, player\_side, black\_name, white\_name

    info = {

        'code': code,

        'name': player\_name,

        'status': 'running'

    }

    while 1:

        try:

            r = requests.get(geturl('rooms') , params={'code' : code})

            r = r.json()

            if r['status'] == 'waiting':

                accepted = True

                requests.post(geturl('rooms'), data=info)

                break

        except KeyError:

            messagebox.showerror("Error" , f"Invalid room code {code}")

            print(f'Invalid room code')

            return False

        except Exception  as e:

            pass

            #print(type(e) , e)

        time.sleep(0.2)

    oppinfo = r

    oppname = oppinfo['name']

    oppside = oppinfo['side']

    player\_side = 'white' if oppside == 'black' else 'black'

    touch\_move = (oppinfo['tm'].lower() == 'true')

    min\_time = oppinfo['mintime']

    increment = int(oppinfo['inc']) if oppinfo['inc'] else 0

    if player\_side == 'white':

        white\_name = player\_name

        black\_name = oppname

        root\_turtle.title(f'{code}: {player\_name} vs {oppname}')

    else:

        white\_name = oppname

        black\_name = player\_name

        root\_turtle.title(f'{code}: {oppname} vs {player\_name}')

    return True

#For the timers and stuff (pre game page)

def settings():

    global canvas\_turtle, var\_min\_time, var\_increment, mode\_of\_play, canvas\_main\_menu, img\_play\_offline, img\_play\_online, img\_history, img\_back, img\_join\_game, img\_create\_game, button\_touch\_move, touch\_move, challenger, joinee\_code, var\_white\_name, var\_black\_name

    possible\_min\_times=["Unlimited", '1', '2', '3', '5', '10', '15', '20', '30', '45', '60', '90']

    possible\_increments=['0','3','5','10','30','60']

    def select\_min\_time(e):

        global combo\_increment

        min\_time\_ = var\_min\_time.get()

        if min\_time\_ == "Unlimited":

            var\_increment.set('')

            combo\_increment['state'] = DISABLED

        elif min\_time\_ != "Unlimited" and str(combo\_increment['state']) == str(DISABLED):

            combo\_increment['state'] = 'readonly'

    def on\_start\_game():

        global canvas\_turtle, canvas\_main\_menu, var\_white\_name, var\_black\_name, var\_min\_time, var\_increment, white\_name, black\_name, min\_time, increment

        white\_name = var\_white\_name.get().strip().strip('\*')

        black\_name = var\_black\_name.get().strip().strip('\*')

        min\_time = var\_min\_time.get()

        increment = var\_increment.get()

        if not(white\_name and black\_name and ((min\_time and increment) or min\_time.lower() == "unlimited")):

            return

        #print(white\_name, black\_name, min\_time, increment)

        destroy\_all\_widgets\_in\_canvas\_main\_menu()

        canvas\_main\_menu.destroy()

        setup\_game()

    def on\_touch\_move\_toggle():

        global touch\_move, button\_touch\_move

        if touch\_move == True:

            touch\_move = False

            button\_touch\_move.configure(image = img\_toggle\_off)

        elif touch\_move == False:

            touch\_move = True

            button\_touch\_move.configure(image = img\_toggle\_on)

        pass

    def on\_back():

        global canvas\_main\_menu

        destroy\_all\_widgets\_in\_canvas\_main\_menu()

        canvas\_main\_menu.destroy()

        main\_menu()

    def on\_choice\_create\_game():

        global canvas\_main\_menu, frame\_self\_details, label\_player\_side, label\_player\_name, challenger, var\_player\_side, var\_player\_name

        def on\_create\_game(): #Online

            global received, canvas\_turtle, canvas\_main\_menu, var\_player\_side, var\_player\_name, player\_side, player\_name, min\_time, increment, code, started

            player\_side = var\_player\_side.get().lower().strip('\*')

            player\_name = var\_player\_name.get().strip().strip('\*')

            min\_time = var\_min\_time.get()

            increment = var\_increment.get()

            if not(player\_side and player\_name and ((min\_time and increment) or min\_time.lower() == "unlimited")):

                return

            if player\_side == "random":

                player\_side = random.choice(("white", "black"))

            received = False

            code = requests.get(geturl('connect')).text

            Thread(target=send\_challenge\_request,args=[code]).start()

            destroy\_all\_widgets\_in\_canvas\_main\_menu()

            canvas\_main\_menu.destroy()

            setup\_game()

        #The player is a challenger

        challenger = True

        #Creating a frame and adding entry boxes to input name and side chosen by the player

        frame\_self\_details = Frame(canvas\_main\_menu, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

        frame\_self\_details.grid(row = 1, column = 1, sticky = EW, padx = 30, pady = (0,10), ipadx = 10, ipady = 10)

        #Gridding frame\_names

        frame\_self\_details.grid\_columnconfigure((0,1), weight = 1)

        frame\_self\_details.grid\_rowconfigure((0,1), weight = 1)

        label\_player\_side = Label(frame\_self\_details, text = "Your side", bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('Consolas', 20, 'bold'), bd = 0, highlightthickness=0)

        label\_player\_name = Label(frame\_self\_details, text = "Your name", bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('Consolas', 20, 'bold'), bd = 0, highlightthickness=0)

        var\_player\_side = StringVar()

        combo\_player\_side = ttk.Combobox(frame\_self\_details, textvariable = var\_player\_side, justify = CENTER, font = ("Consolas", 20), state = 'readonly')

        combo\_player\_side['values'] = ('White', 'Black', 'Random')

        combo\_player\_side.current()

        var\_player\_name = StringVar()

        entry\_player\_name = Entry(frame\_self\_details, textvariable = var\_player\_name, font = ('Consolas', 20), justify = CENTER)

        #Gridding all the contents of frame\_self\_details

        label\_player\_side.grid(row = 0, column = 0, sticky = NSEW, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        combo\_player\_side.grid(row = 0, column = 1, sticky = EW, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        label\_player\_name.grid(row = 1, column = 0, sticky = NSEW, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        entry\_player\_name.grid(row = 1, column = 1, sticky = EW, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        #Creating the button to create game

        button\_create\_game = CustomButton(canvas\_main\_menu, text = "CREATE GAME", font = ('Comic Sans', 22, 'bold'), bg = constant.SECONDARYCLR, fg = constant.DARKBGTEXTCLR, relief = "ridge", justify = "center", command = on\_create\_game)

        button\_create\_game.grid(row = 4, column = 1, sticky = EW, padx = 30, pady = (5, 20), ipadx = 10, ipady = 10)

        #Taking out focus

        label\_player\_side.bind("<Button-1>", lambda e: label\_player\_side.focus\_set())

        label\_player\_name.bind("<Button-1>", lambda e: label\_player\_name.focus\_set())

        offline\_or\_challenger()

    def on\_choice\_join\_game():

        global challenger, canvas\_main\_menu, frame\_joinee\_code, label\_player\_name, label\_joinee\_code, var\_joinee\_code, var\_player\_name

        def on\_join\_game(): #Online

            global canvas\_turtle, canvas\_main\_menu, var\_player\_name, player\_name, var\_joinee\_code, joinee\_code, player\_side, min\_time, increment, touch\_move, code, accepted

            player\_name = var\_player\_name.get().strip().strip('\*')

            joinee\_code = var\_joinee\_code.get().strip().strip('\*')

            code = joinee\_code

            if not (player\_name and joinee\_code):

                return

            #Check if valid game code and get player\_side, min\_time, increment and touch\_move

            accepted = False

            success = accept\_challenge\_request(joinee\_code)

            if success:

                    destroy\_all\_widgets\_in\_canvas\_main\_menu()

                    canvas\_main\_menu.destroy()

                    setup\_game()

        #The player is a joinee

        challenger = False

        #Creating a frame and adding entry boxes to input name and side chosen by the player

        frame\_names = Frame(canvas\_main\_menu, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

        frame\_names.grid(row = 1, column = 1, sticky = EW, padx = 30, pady = 10, ipadx = 5, ipady = 5)

        #Gridding frame\_names

        frame\_names.grid\_rowconfigure(0, weight = 1)

        frame\_names.grid\_columnconfigure((0,1), weight = 1)

        label\_player\_name = Label(frame\_names, text = "Your name", bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('Consolas', 20, 'bold'), bd = 0, highlightthickness=0)

        label\_player\_name.grid(row = 0, column = 0, sticky = EW, padx = 5, pady = 5, ipadx = 10, ipady = 2)

        var\_player\_name = StringVar()

        entry\_player\_name = Entry(frame\_names, textvariable = var\_player\_name, font = ('Consolas', 20), justify = CENTER)

        entry\_player\_name.grid(row = 0, column = 1, sticky = EW, padx = 5, pady = 5, ipadx = 10, ipady = 2)

        frame\_joinee\_code = Frame(canvas\_main\_menu, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

        frame\_joinee\_code.grid(row = 2, column = 1, sticky = EW, padx = 30, pady = 10, ipadx = 5, ipady = 5)

        #Gridding frame\_joinee\_code

        frame\_joinee\_code.grid\_rowconfigure(0, weight = 1)

        frame\_joinee\_code.grid\_columnconfigure((0,1), weight = 1)

        label\_joinee\_code = Label(frame\_joinee\_code, text = "Game code", bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('Consolas', 20, 'bold'), bd = 0, highlightthickness=0)

        label\_joinee\_code.grid(row = 0, column = 0, sticky = EW, padx = 5, pady = 5, ipadx = 10, ipady = 2)

        var\_joinee\_code = StringVar()

        entry\_joinee\_code = Entry(frame\_joinee\_code, textvariable = var\_joinee\_code, font = ('Consolas', 20), justify = CENTER)

        entry\_joinee\_code.grid(row = 0, column = 1, sticky = EW, padx = 5, pady = 5, ipadx = 10, ipady = 2)

        #Creating the button to join game

        button\_join\_game = CustomButton(canvas\_main\_menu, text = "JOIN GAME", font = ('Comic Sans', 22, 'bold'), bg = constant.SECONDARYCLR, fg = constant.DARKBGTEXTCLR, relief = "ridge", justify = "center", command = on\_join\_game)

        button\_join\_game.grid(row = 3, column = 1, sticky = EW, padx = 30, pady = (5, 20), ipadx = 10, ipady = 10)

        #Taking out focus

        canvas\_main\_menu.bind("<Button-1>", lambda e: canvas\_main\_menu.focus\_set())

        frame\_names.bind("<Button-1>", lambda e: frame\_names.focus\_set())

        frame\_joinee\_code.bind("<Button-1>", lambda e: frame\_joinee\_code.focus\_set())

        label\_joinee\_code.bind("<Button-1>", lambda e: label\_joinee\_code.focus\_set())

        label\_player\_name.bind("<Button-1>", lambda e: label\_player\_name.focus\_set())

    def offline\_or\_challenger():

        global canvas\_main\_menu, touch\_move, var\_min\_time, var\_increment, button\_touch\_move, combo\_increment

        #Creating a frame for the timer initialisation and then creating the timers

        frame\_timers = Frame(canvas\_main\_menu, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

        frame\_timers.grid(row = 2, column = 1, sticky = EW, padx = 30, pady = 10, ipadx = 10, ipady = 10)

        #Gridding frame\_timers

        frame\_timers.grid\_columnconfigure((0,1), weight = 1)

        frame\_timers.grid\_rowconfigure((0,1), weight = 1)

        #Creating the title labels for the content in frame\_timers

        label\_min\_time = Label(frame\_timers, text = "Min time (min)", font = ("Consolas", 20, 'bold'), bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, bd = 0, highlightthickness=0)

        label\_increment = Label(frame\_timers, text = "Increment (sec)", font = ("Consolas", 20, 'bold'), bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, bd = 0, highlightthickness=0)

        #Creating the comboboxes for receiving their choices for the min time and increment

        var\_min\_time = StringVar()

        combo\_min\_time= ttk.Combobox(frame\_timers, textvariable = var\_min\_time, justify = CENTER, font = ("Consolas", 20), state = 'readonly')

        var\_increment = StringVar()

        combo\_increment = ttk.Combobox(frame\_timers, textvariable = var\_increment, justify = CENTER, font = ("Consolas", 20), state = 'readonly')

        combo\_min\_time['values'] = tuple(possible\_min\_times)

        combo\_increment['values'] = tuple(possible\_increments)

        combo\_min\_time.current()

        combo\_increment.current()

        combo\_min\_time.bind('<<ComboboxSelected>>', select\_min\_time)

        #Packing all the contents of frame\_timers

        label\_min\_time.grid(row = 0, column = 0, sticky = NSEW, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        combo\_min\_time.grid(row = 0, column = 1, sticky = EW, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        label\_increment.grid(row = 1, column = 0, sticky = NSEW, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        combo\_increment.grid(row = 1, column = 1, sticky = EW, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        #Creating a frame and then adding touchpiece toggle

        touch\_move = False

        frame\_touch\_move = Frame(canvas\_main\_menu, bg = constant.DARKBGCLR, bd = 0, highlightthickness = 0)

        frame\_touch\_move.grid(row = 3, column = 1, sticky = EW, padx = 30, pady = 10, ipadx = 10, ipady = 10)

        #Gridding frame\_touch\_move

        frame\_touch\_move.grid\_columnconfigure((0,1), weight = 1)

        frame\_touch\_move.grid\_rowconfigure(0, weight = 1)

        label\_touch\_move = Label(frame\_touch\_move, text = "Enable touch move", font = ("Consolas", 20, 'bold'), bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, bd = 0, highlightthickness=0)

        button\_touch\_move = Button(frame\_touch\_move, image = img\_toggle\_off, bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, activebackground = constant.DARKBGCLR, activeforeground = constant.DARKBGTEXTCLR, font = ('Consolas', 20, 'bold'), command = on\_touch\_move\_toggle, bd = 0, highlightthickness = 0)

        label\_touch\_move.grid(row = 0, column = 0, sticky = E, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        button\_touch\_move.grid(row = 0, column = 1, sticky = W, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        #Taking out focus

        canvas\_main\_menu.bind("<Button-1>", lambda e: canvas\_main\_menu.focus\_set())

        frame\_timers.bind("<Button-1>", lambda e: frame\_timers.focus\_set())

        label\_min\_time.bind("<Button-1>", lambda e: label\_min\_time.focus\_set())

        label\_increment.bind("<Button-1>", lambda e: label\_increment.focus\_set())

        label\_touch\_move.bind("<Button-1>", lambda e: label\_touch\_move.focus\_set())

    challenger = None #Undecided

    destroy\_all\_widgets\_in\_canvas\_main\_menu()

    #Regridding canvas\_main\_menu

    canvas\_main\_menu.grid\_columnconfigure((0,1), weight = 1)

    canvas\_main\_menu.grid\_rowconfigure(0, weight = 0)

    canvas\_main\_menu.grid\_rowconfigure((1,2,3,4), weight = 1)

    #Adding a back button

    img\_back = PhotoImage(file = "./Icons/back.png").subsample(2,2)

    button\_back = Button(canvas\_main\_menu, image = img\_back, bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, activebackground = constant.DARKBGCLR, activeforeground = constant.DARKBGTEXTCLR, font = ('Consolas', 20, 'bold'), command = on\_back, bd = 0, highlightthickness = 0)

    button\_back.grid(row = 0, column = 1, sticky = W, padx = 10)

    if mode\_of\_play == "offline":

        #Displaying the choice (online/offline) on the left half of the screen

        img\_play\_offline = PhotoImage(file = "./Icons/play\_offline.png").subsample(2,2)

        label\_play\_offline = Label(canvas\_main\_menu, text = "PLAY OVER THE\nBOARD", image = img\_play\_offline, compound = TOP, font = ('Comic Sans', 25, 'bold'), bg = constant.PRIMARYCLR, fg = constant.LIGHTBGTEXTCLR, justify = 'center')

        label\_play\_offline.grid(row = 0, column = 0, rowspan = 5, sticky = NSEW, ipadx = 30, ipady = 20)

        #Creating a frame and adding the entry boxes in it to input the name of the players

        frame\_names = Frame(canvas\_main\_menu, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

        frame\_names.grid(row = 1, column = 1, sticky = EW, padx = 30, pady = (0,10), ipadx = 10, ipady = 10)

        #Gridding frame\_names

        frame\_names.grid\_columnconfigure((0,1), weight = 1)

        frame\_names.grid\_rowconfigure((0,1), weight = 1)

        label\_white\_name = Label(frame\_names, text = "White name", bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('Consolas', 20, 'bold'), bd = 0, highlightthickness=0)

        var\_white\_name = StringVar()

        entry\_white\_name = Entry(frame\_names, textvariable = var\_white\_name, font = ('Consolas', 20), justify = CENTER)

        label\_black\_name = Label(frame\_names, text = "Black name", bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('Consolas', 20, 'bold'), bd = 0, highlightthickness=0)

        var\_black\_name = StringVar()

        entry\_black\_name = Entry(frame\_names, textvariable = var\_black\_name, font = ('Consolas', 20), justify = CENTER)

        label\_white\_name.grid(row = 0, column = 0, sticky = NSEW, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        entry\_white\_name.grid(row = 0, column = 1, sticky = EW, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        label\_black\_name.grid(row = 1, column = 0, sticky = NSEW, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        entry\_black\_name.grid(row = 1, column = 1, sticky = EW, padx = 5, pady = 10, ipadx = 10, ipady = 2)

        #Creating the button to start game

        button\_start\_game = CustomButton(canvas\_main\_menu, text = "START GAME", font = ('Comic Sans', 22, 'bold'), bg = constant.SECONDARYCLR, fg = constant.DARKBGTEXTCLR, relief = "ridge", justify = "center", command = on\_start\_game)

        button\_start\_game.grid(row = 4, column = 1, sticky = EW, padx = 30, pady =  (5, 20), ipadx = 10, ipady = 10)

        #Taking out focus

        label\_play\_offline.bind("<Button-1>", lambda e: label\_play\_offline.focus\_set())

        frame\_names.bind("<Button-1>", lambda e: frame\_names.focus\_set())

        label\_white\_name.bind("<Button-1>", lambda e: label\_white\_name.focus\_set())

        label\_black\_name.bind("<Button-1>", lambda e: label\_black\_name.focus\_set())

        offline\_or\_challenger()

    elif mode\_of\_play == "online":

        #Displaying the choice (online/offline) on the left half of the screen

        img\_play\_online = PhotoImage(file = "./Icons/play\_online.png").subsample(3,3)

        label\_play\_online = Label(canvas\_main\_menu, text = "PLAY A FRIEND\nONLINE", image = img\_play\_online, compound = TOP, font = ('Comic Sans', 25, 'bold'), bg = constant.PRIMARYCLR, fg = constant.LIGHTBGTEXTCLR, justify = 'center')

        label\_play\_online.grid(row = 0, column = 0, rowspan = 5, sticky = NSEW, ipadx = 30, ipady = 20)

        #Creating the images for create game and join game

        img\_create\_game = PhotoImage(file = "./Icons/plus.png")

        img\_join\_game = PhotoImage(file = "./Icons/enter.png")

        #Adding the buttons to know whether the user wants to join a game or create a game

        button\_choice\_create\_game = CustomButton(canvas\_main\_menu, text = "CREATE GAME", image = img\_create\_game, compound = TOP,  font = ('Comic Sans', 18, 'bold'), bg = constant.PRIMARYCLR, fg = constant.LIGHTBGTEXTCLR, relief = "ridge", justify = "center", command = lambda : (button\_choice\_create\_game.destroy(), button\_choice\_join\_game.destroy(), on\_choice\_create\_game()))

        button\_choice\_create\_game.grid(row = 1, column = 1, sticky = EW, ipadx = 50, ipady = 30, padx = 40, pady = (0,30))

        button\_choice\_join\_game = CustomButton(canvas\_main\_menu, text = "JOIN GAME", image = img\_join\_game, compound = TOP,  font = ('Comic Sans', 18, 'bold'), bg = constant.PRIMARYCLR, fg = constant.LIGHTBGTEXTCLR, relief = "ridge", justify = "center", command = lambda : (button\_choice\_join\_game.destroy(), button\_choice\_create\_game.destroy(), on\_choice\_join\_game()))

        button\_choice\_join\_game.grid(row = 2, column = 1, sticky = EW, ipadx = 50, ipady = 30, padx = 40, pady = (30, 60))

        #Taking out focus

        label\_play\_online.bind("<Button-1>", lambda e: label\_play\_online.focus\_set())

def disable\_game\_controls():

    global scale\_board, button\_color\_chooser, button\_volume\_toggle, button\_legal\_moves, button\_flip

    scale\_board.configure(state = DISABLED)

    button\_color\_chooser.configure(state = DISABLED)

    button\_volume\_toggle.configure(state = DISABLED)

    button\_legal\_moves.configure(state = DISABLED)

    button\_flip.configure(state = DISABLED)

    scale\_board.unbind("<ButtonRelease-1>")

def enable\_game\_controls():

    global scale\_board, button\_color\_chooser, button\_volume\_toggle, button\_legal\_moves, button\_flip

    scale\_board.configure(state = NORMAL)

    button\_color\_chooser.configure(state = NORMAL)

    button\_volume\_toggle.configure(state = NORMAL)

    button\_legal\_moves.configure(state = NORMAL)

    button\_flip.configure(state = NORMAL)

    scale\_board.bind("<ButtonRelease-1>", on\_scale\_board\_release)

def on\_scale\_board\_release(e):

    global scale\_board, var\_board\_scaler

    game.size = 600 \* var\_board\_scaler.get() / 50

    game.sqsize = game.size / 8

    game.stretch\_square=(game.size/8)/20

    if e is not None:

        game.configure\_game()

    size\_canvas\_turtle = int(game.size) + 15

    size\_root\_turtle = size\_canvas\_turtle + 35

    canvas\_turtle.configure(width = size\_canvas\_turtle, height = size\_canvas\_turtle)

    canvas\_turtle.update()

    width\_root\_turtle = size\_root\_turtle

    if button\_extend\_left['text'] == ">":

        frame\_left.update\_idletasks()

        width\_root\_turtle += frame\_left.winfo\_width()

    if button\_extend\_right['text'] == "<":

        frame\_right.update\_idletasks()

        width\_root\_turtle += frame\_right.winfo\_width()

    root\_turtle.geometry(f"{width\_root\_turtle}x{size\_root\_turtle+95}")

def on\_pin\_toggle():

    global pinned, button\_pin

    if pinned == True:

        pinned = False

        root\_turtle.attributes('-topmost',False)

        button\_pin.configure(image = img\_pin\_outline)

    elif pinned == False:

        pinned = True

        root\_turtle.attributes('-topmost',True)

        button\_pin.configure(image = img\_pin\_filled)

def on\_extend\_left():

    global button\_extend\_left, frame\_left

    w = root\_turtle.winfo\_width()

    h = root\_turtle.winfo\_height()

    frame\_left.update\_idletasks()

    if button\_extend\_left['text'] == "<":

        frame\_left.grid(row = 0, column = 0, rowspan = 3, sticky = NSEW)

        new\_w = w + frame\_left.winfo\_width()

        button\_extend\_left.configure(text = ">")

    elif button\_extend\_left['text'] == ">":

        new\_w = w - frame\_left.winfo\_width()

        frame\_left.grid\_forget()

        button\_extend\_left.configure(text = "<")

    root\_turtle.geometry(f"{new\_w}x{h}")

def on\_extend\_right():

    global button\_extend\_right, frame\_right

    w = root\_turtle.winfo\_width()

    h = root\_turtle.winfo\_height()

    frame\_right.update\_idletasks()

    if button\_extend\_right['text'] == ">":

        frame\_right.grid(row = 0, column = 2, rowspan = 3, sticky = NSEW)

        new\_w = w + frame\_right.winfo\_width()

        button\_extend\_right.configure(text = "<")

    elif button\_extend\_right['text'] == "<":

        new\_w = w - frame\_right.winfo\_width()

        frame\_right.grid\_forget()

        button\_extend\_right.configure(text = ">")

    root\_turtle.geometry(f"{new\_w}x{h}")

def on\_color\_chooser():

    try:

        clr\_dark = colorchooser.askcolor(title ="Choose DARK square color", color = game.dark\_square\_clr)[0]

        if clr\_dark is None:

            return

    except:

        return

    game.dark\_square\_clr = rgb\_to\_hex(clr\_dark)

    game.configure\_game()

def on\_volume\_toggle():

    global volume\_toggle, button\_volume\_toggle

    if volume\_toggle == True:

        volume\_toggle = False

        button\_volume\_toggle.configure(image = img\_volume\_off)

    elif volume\_toggle == False:

        volume\_toggle = True

        button\_volume\_toggle.configure(image = img\_volume\_on)

def on\_flip():

    global label\_white\_timer\_top, label\_white\_timer\_bottom, label\_black\_timer\_top, label\_black\_timer\_bottom

    if mode\_of\_play == "offline":

        global button\_white\_resign, button\_black\_resign

    if game.white == True:

        game.white = False

        game.black = True

        label\_white\_timer\_bottom.grid\_forget()

        label\_white\_timer\_top.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 2)

        label\_black\_timer\_top.grid\_forget()

        label\_black\_timer\_bottom.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 2)

        if mode\_of\_play == "offline":

            button\_white\_resign.grid(row = 0, column = 0, sticky = NSEW)

            button\_black\_resign.grid(row = 2, column = 0, sticky = NSEW)

    elif game.black == True:

        game.black = False

        game.white = True

        label\_white\_timer\_top.grid\_forget()

        label\_white\_timer\_bottom.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 2)

        label\_black\_timer\_bottom.grid\_forget()

        label\_black\_timer\_top.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 2)

        if mode\_of\_play == "offline":

            button\_white\_resign.grid(row = 2, column = 0, sticky = NSEW)

            button\_black\_resign.grid(row = 0, column = 0, sticky = NSEW)

    game.configure\_game()

def on\_game\_over():

    #Generating pgn

    game.generate\_pgn()

    #Permanently removing extend left

    frame\_left.grid\_forget()

    button\_extend\_left.configure(text = "<")

    button\_extend\_left.grid\_forget()

    on\_scale\_board\_release(None)

    #Disabling mouse activity on the turtle canvas

    game.mouse\_vacant = False

    wn.onclick(lambda x,y: None)

    canvas\_turtle.unbind("<ButtonPress>")

    canvas\_turtle.unbind("<B1-Motion>")

    canvas\_turtle.unbind("<ButtonRelease-1>")

    #Displaying the result in the status bar

    if game.game\_result == "W":

        root\_turtle.title(f'\*{white\_name} vs {black\_name}')

    elif game.game\_result == "B":

        root\_turtle.title(f'{white\_name} vs \*{black\_name}')

    elif game.game\_result == "D":

        root\_turtle.title(f'\*{white\_name} vs \*{black\_name}')

        on\_draw(False)

    elif game.game\_result == "S":

        root\_turtle.title(f'#{white\_name} vs #{black\_name}')

    #Pause the Timers

    #Adding game to database

    game.add\_to\_db(game.game\_date, game.game\_start\_time, white\_name, black\_name, game.game\_result, logic.List\_of\_Moves, game.move\_duration , min\_time,  increment)

def on\_draw(confirm = True):

    global draw, draw\_boundary, button\_extend\_left, frame\_left, draw\_offer

    if confirm:

        if mode\_of\_play == "offline":

            choice = messagebox.askyesno('Draw agreement', 'Do you both agree?')

            if not choice:

                return

        elif mode\_of\_play == "online":

            requests.post(geturl('connection') , data= {'code':code, 'side':player\_side, 'draw':'idk'})

            while 1:

                try :

                    r = requests.get(geturl('connection') , params= {'side' : player\_side , 'code' : code , 'connectop':False})

                    time.sleep(0.2)

                    info = r.json()['offers']['draw offers']

                    if info in ('accepted' , 'rejected'):

                        break

                except:

                    pass

            requests.post(geturl('connection') , params = {'code' : code , 'reset':None})

            if info == 'rejected':

                return

        game.game\_result = "D"

        on\_game\_over()

    #Display draw with a green border around the board

    try:

        draw\_boundary.clear()

        wn.update()

    except:

        pass

    wn.tracer(0)

    draw\_boundary=turtle.Turtle()

    draw\_boundary.ht()

    draw\_boundary.width(7)

    draw\_boundary.color("green")

    draw\_boundary.pu()

    draw\_boundary.goto(-(game.size/2)+game.drift,(game.size/2))

    draw\_boundary.pd()

    for \_ in range(4):

        draw\_boundary.fd(game.size)

        draw\_boundary.rt(90)

    wn.tracer(1)

    def on\_turtle\_close():

        global program\_running

        program\_running = False

        root\_turtle.after(500, lambda: root\_turtle.destroy())

        database\_functions.close\_connection()

    root\_turtle.protocol("WM\_DELETE\_WINDOW", on\_turtle\_close)

def on\_white\_resign():

    global button\_extend\_left, frame\_left, move\_duration

    if mode\_of\_play == "offline":

        choice = messagebox.askyesno('White resign', f'Are you ({white\_name}) sure?')

        if not choice:

            return

    elif mode\_of\_play == "online":

        if messagebox.askyesno('White resign' , f'Are you sure?'):

            requests.post(geturl('connection') , data = {'resign' : 'idk' ,'side' : 'white' ,'code' : code})

        else:

            return

    game.game\_result = "B"

    on\_game\_over()

def on\_black\_resign():

    global button\_extend\_left, frame\_left, move\_duration, player\_side, challenger

    if mode\_of\_play == "offline":

        choice = messagebox.askyesno('Black resign', f'Are you ({black\_name}) sure?')

        if not choice:

            return

    elif mode\_of\_play == "online":

        print('resign')

        if messagebox.askyesno('Black resign', f'Are you sure?'):

            requests.post(geturl('connection') , data = {'resign' : 'idk' ,'side' : 'black' ,'code' : code})

        else:

            return

    game.game\_result = "W"

    on\_game\_over()

def on\_legal\_moves():

    global button\_legal\_moves, img\_legal\_moves\_off, img\_legal\_moves\_on

    if game.legalmoves == True:

        game.legalmoves = False

        button\_legal\_moves.configure(image = img\_legal\_moves\_off)

    elif game.legalmoves == False:

        game.legalmoves = True

        button\_legal\_moves.configure(image = img\_legal\_moves\_on)

    if game.legalmoves==True and game.move\_stage==1:

        for m,n in game.legaladd:

            blm, bln=7-m, 7-n

            game.change\_square\_clr(m, n, blm, bln, constant.LEGALLIGHTSQUARECLR, constant.LEGALDARKSQUARECLR, update=False)

        wn.tracer(1)

    elif game.legalmoves==False and game.move\_stage==1:

        for m,n in game.legaladd:

            blm, bln=7-m, 7-n

            game.change\_square\_clr(m, n, blm, bln, game.light\_square\_clr, game.dark\_square\_clr, update=False)

        wn.tracer(1)

def on\_code\_copy():

    global code

    if os.sys.platform.lower() == 'windows':

        clipboard.copy(code)

    else:

        os.system(f'echo -n "{code}" | xclip -selection clipboard')

def setup\_game():

    global canvas\_turtle, touch\_move, var\_board\_scaler, scale\_board, mode\_of\_play, pinned, img\_pin\_outline, img\_pin\_filled, button\_pin, img\_color\_chooser, img\_flip, img\_draw, img\_resign, volume\_toggle, button\_volume\_toggle, img\_volume\_on, img\_volume\_off, button\_extend\_left, button\_extend\_right, frame\_left, frame\_right, label\_white\_timer\_top, label\_white\_timer\_bottom, label\_black\_timer\_top, label\_black\_timer\_bottom, img\_legal\_moves\_on, img\_legal\_moves\_off, button\_legal\_moves, img\_touch\_move\_on, img\_touch\_move\_off, button\_color\_chooser, button\_flip, frame\_moves, chess\_not\_row, button\_white\_resign, button\_black\_resign, button\_resign, img\_copy, frame\_pgn

    #Configuring root\_turtle

    root\_turtle.configure(bg = constant.LIGHTBGCLR)

    #Reconfiguring minsize of the window

    root\_turtle.minsize(0,0)

    root\_turtle.resizable(width = False, height = False)

    root\_turtle.overrideredirect(False)

    root\_turtle.withdraw()

    root\_turtle.deiconify()

    #Gridding root\_turtle

    root\_turtle.grid\_rowconfigure(0, weight = 0)

    root\_turtle.grid\_columnconfigure(0, weight = 0)

    root\_turtle.grid\_rowconfigure(1, weight = 1)

    root\_turtle.grid\_columnconfigure(1, weight = 1)

    #Placing canvas\_turtle inside root\_turtle

    canvas\_turtle.grid(row = 1, column = 1, sticky = NSEW)

    #Giving a background colour to wn

    wn.bgcolor("#000000")

    #Adding all the other elements present during the game (except timers)

    #Creating all the frames for the 4 sides

    frame\_top = Frame(root\_turtle, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    frame\_bottom = Frame(root\_turtle, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    frame\_left = Frame(root\_turtle, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    frame\_right = Frame(root\_turtle, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    #Adding all the frames to the screenready\_to\_play = True

    frame\_top.grid(row = 0, column = 1, sticky = NSEW)

    frame\_bottom.grid(row = 2, column = 1, sticky = NSEW)

    #Gridding all the frames

    frame\_top.grid\_columnconfigure((2,3,4), weight = 1)

    frame\_bottom.grid\_columnconfigure((0,1,2), weight = 1)

    frame\_left.grid\_rowconfigure((0,1,2), weight = 1)

    frame\_right.grid\_rowconfigure(0, weight = 1)

    frame\_right.grid\_columnconfigure(0, weight = 1)

    button\_extend\_left = Button(frame\_top, text = "<", font = ('consolas', 20, 'bold'), bg = constant.LIGHTBGCLR, fg = constant.LIGHTBGTEXTCLR, activebackground = constant.LIGHTBGCLR, activeforeground = constant.LIGHTBGTEXTCLR, command = on\_extend\_left, bd = 0, highlightthickness = 0)

    button\_extend\_left.grid(row = 0, column = 0, sticky = NSEW, padx = (0,1))

    button\_extend\_right = Button(frame\_top, text = ">", font = ('consolas', 20, 'bold'), bg = constant.LIGHTBGCLR, fg = constant.LIGHTBGTEXTCLR, activebackground = constant.LIGHTBGCLR, activeforeground = constant.LIGHTBGTEXTCLR, command = on\_extend\_right, bd = 0, highlightthickness = 0)

    button\_extend\_right.grid(row = 0, column = 5, sticky = NSEW, padx = (1,0))

    #Adding frame\_moves to frame\_right

    frame\_moves = ScrolledFrame(frame\_right, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0, max\_height = 1000)

    frame\_moves.grid(row = 0, column = 0, sticky = NSEW, padx = 2, pady = 2, ipadx = 2, ipady = 2)

    #Gridding frame\_moves.viewPort

    frame\_moves.viewPort.grid\_columnconfigure((1,2), weight = 1)

    #Adding frame\_pgn for copying and downloading pgn of the game after it gets over

    frame\_pgn = Frame(frame\_right, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    frame\_pgn.grid\_columnconfigure(0, weight = 1)

    frame\_pgn.grid(row = 1, column = 0, sticky = NSEW)

    #Initialising chess\_not\_row

    chess\_not\_row = 0

    #Adding the board scaler to frame\_top

    var\_board\_scaler = IntVar()

    scale\_board = ttk.Scale(frame\_top, variable = var\_board\_scaler, from\_ = 30, to = 70, orient = HORIZONTAL)

    var\_board\_scaler.set(50)

    scale\_board.bind("<ButtonRelease-1>", on\_scale\_board\_release)

    scale\_board.grid(row = 0, column = 4, sticky = EW, padx = 5)

    #Adding the pin to frame\_top

    img\_pin\_outline = PhotoImage(file = "./Icons/pin\_outline.png").subsample(15,15)

    img\_pin\_filled = PhotoImage(file = "./Icons/pin\_filled.png").subsample(15,15)

    button\_pin = Button(frame\_top, image = img\_pin\_outline, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = on\_pin\_toggle, bd = 0, highlightthickness = 0)

    pinned = False

    button\_pin.grid(row = 0, column = 1, sticky = NSEW)

    #Creating frame\_icons to contain all the icons

    frame\_icons = Frame(frame\_bottom, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    #Adding frame\_icons to the screen

    frame\_icons.grid(row = 0, column = 0, sticky = NSEW, padx = 5)

    #Gridding frame\_icons

    frame\_icons.grid\_columnconfigure((0,1,2,3,4), weight = 1)

    #Adding the color chooser icon to frame\_bottom

    img\_color\_chooser = PhotoImage(file = "./Icons/color\_chooser.png").subsample(3,3)

    button\_color\_chooser = Button(frame\_icons, image = img\_color\_chooser, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = on\_color\_chooser, bd = 0, highlightthickness = 0)

    #Adding volume toggle icon to frame\_bottom

    img\_volume\_on = PhotoImage(file = "./Icons/volume\_on.png").subsample(4,4)

    img\_volume\_off = PhotoImage(file = "./Icons/volume\_off.png").subsample(4,4)

    volume\_toggle = True

    button\_volume\_toggle = Button(frame\_icons, image = img\_volume\_on, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = on\_volume\_toggle, bd = 0, highlightthickness = 0)

    #Adding flip icon to frame\_bottom

    img\_flip = PhotoImage(file = "./Icons/flip.png").subsample(3,3)

    button\_flip = Button(frame\_icons, image = img\_flip, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = on\_flip, bd = 0, highlightthickness = 0)

    #Adding the legal moves icon to frame\_bottom

    img\_legal\_moves\_on = PhotoImage(file = "./Icons/arrows\_on.png")

    img\_legal\_moves\_off = PhotoImage(file = "./Icons/arrows\_off.png")

    button\_legal\_moves = Button(frame\_icons, image = img\_legal\_moves\_on, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = on\_legal\_moves, bd = 0, highlightthickness = 0)

    #Adding draw icon to frame\_left

    img\_draw = PhotoImage(file = "./Icons/draw.png").subsample(16, 16)

    button\_draw = Button(frame\_left, image = img\_draw, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = on\_draw, bd = 0, highlightthickness = 0)

    #Adding resign icon to frame\_left

    img\_resign = PhotoImage(file = "./Icons/flag.png").subsample(3,3)

    #Adding the info about if touch move is enabled or disabled

    img\_touch\_move\_on = PhotoImage(file = "./Icons/touch\_move\_on.png")

    img\_touch\_move\_off = PhotoImage(file = "./Icons/touch\_move\_off.png")

    label\_touch\_move = Label(frame\_icons, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    #Adding the room code copy button only for challenger if online

    img\_copy = PhotoImage(file = "./Icons/clipboard.png").subsample(3,3)

    if challenger:

        button\_code\_copy = Button(frame\_icons, image = img\_copy, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = on\_code\_copy, bd = 0, highlightthickness = 0)

        button\_code\_copy.grid(row = 0, column = 5, sticky = NSEW)

        Tooltip(button\_code\_copy, text = f"Copy room code: {code}")

    if touch\_move:

        label\_touch\_move.configure(image = img\_touch\_move\_on)

        Tooltip(label\_touch\_move, text = "Touch move enabled")

    elif not touch\_move:

        label\_touch\_move.configure(image = img\_touch\_move\_off)

        Tooltip(label\_touch\_move, text = "Touch move disabled")

    if mode\_of\_play == "online":

        button\_resign = Button(frame\_left, image = img\_resign, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

        button\_resign.grid(row = 0, column = 0, sticky = NSEW)

        if player\_side == "white":

            button\_resign.configure(command = on\_white\_resign)

        elif player\_side == "black":

            button\_resign.configure(command = on\_black\_resign)

        Tooltip(button\_resign, text = "Resign")

    elif mode\_of\_play == "offline":

        button\_white\_resign = Button(frame\_left, image = img\_resign, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = on\_white\_resign, bd = 0, highlightthickness = 0)

        button\_black\_resign = Button(frame\_left, image = img\_resign, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = on\_black\_resign, bd = 0, highlightthickness = 0)

        button\_white\_resign.grid(row = 2, column = 0, sticky = NSEW)

        button\_black\_resign.grid(row = 0, column = 0, sticky = NSEW)

        Tooltip(button\_white\_resign, text = "White resign")

        Tooltip(button\_black\_resign, text = "Black resign")

    #Adding all the icons to frame\_icons

    button\_color\_chooser.grid(row = 0, column = 0, sticky = NSEW)

    button\_volume\_toggle.grid(row = 0, column = 1, sticky = NSEW)

    button\_flip.grid(row = 0, column = 2, sticky = NSEW)

    button\_legal\_moves.grid(row = 0, column = 3, sticky = NSEW)

    label\_touch\_move.grid(row = 0, column = 4, sticky = NSEW)

    #Adding button\_draw to frame\_left

    button\_draw.grid(row = 1, column = 0, sticky = NSEW)

    #Adding the timer labels to frame\_top and frame\_bottom

    start\_time = tfor(int(min\_time)\*60 if min\_time.isdigit() else 0)[:-2]

    label\_white\_timer\_top = Label(frame\_top, text = start\_time, bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('consolas', 15, 'bold'), bd = 2)

    label\_white\_timer\_bottom = Label(frame\_bottom, text = start\_time, bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('consolas', 15, 'bold'), bd = 2)

    label\_black\_timer\_top = Label(frame\_top, text =  start\_time, bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('consolas', 15, 'bold'), bd = 2)

    label\_black\_timer\_bottom = Label(frame\_bottom, text = start\_time, bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('consolas', 15, 'bold'), bd = 2)

    if mode\_of\_play == "offline":

        label\_white\_timer\_bottom.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 2)

        label\_black\_timer\_top.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 2)

    elif mode\_of\_play == "online":

        if player\_side == "white":

            label\_white\_timer\_bottom.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 2)

            label\_black\_timer\_top.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 2)

        elif player\_side == "black":

            label\_white\_timer\_top.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 2)

            label\_black\_timer\_bottom.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 2)

    #Adding tooltips for the required widgets

    Tooltip(button\_flip, text = "Switch view")

    Tooltip(button\_volume\_toggle, text = "Toggle sound")

    Tooltip(button\_color\_chooser, text = "Change square colour")

    Tooltip(button\_pin, text = "Toggle pin window")

    Tooltip(button\_legal\_moves, text = "Toggle legal moves")

    Tooltip(scale\_board, text = "Scale board size")

    if mode\_of\_play == "offline":

        Tooltip(button\_draw, text = "Draw")

    elif mode\_of\_play == "online":

        Tooltip(button\_draw, text = "Offer draw")

    for \_ in range(2):

        on\_extend\_left()

        on\_extend\_right()

    if challenger:

        if not received:

            if player\_side == "white":

                root\_turtle.title(f'{code}: {player\_name} vs (waiting...) ')

            elif player\_side == "black":

                root\_turtle.title(f'{code}: (waiting...) vs {player\_name}')

    elif challenger is None: #Offline

        root\_turtle.title(f'{white\_name} vs {black\_name}')

    game.game\_main(touch\_move)

def about():

    global img\_back, frame\_about, canvas\_main\_menu, label\_about\_info, bind\_configure

    destroy\_all\_widgets\_in\_canvas\_main\_menu()

    def on\_back():

        global canvas\_main\_menu, bind\_configure, frame\_about

        frame\_about.unbind("<Configure>", bind\_configure)

        frame\_about.destroy()

        destroy\_all\_widgets\_in\_canvas\_main\_menu()

        canvas\_main\_menu.destroy()

        main\_menu()

    def on\_frame\_about\_configure(e):

        global label\_about\_info, frame\_about

        frame\_about.viewPort.update\_idletasks()

        label\_about\_info.configure(wraplength = frame\_about.viewPort.winfo\_width() - 20)

    #Gridding canvas\_main\_menu

    canvas\_main\_menu.grid\_rowconfigure((0,1,2), weight = 0)

    canvas\_main\_menu.grid\_columnconfigure((0,1,2), weight = 0)

    canvas\_main\_menu.grid\_rowconfigure(0, weight = 1)

    canvas\_main\_menu.grid\_columnconfigure(0, weight = 1)

    #Creating, gridding and adding frame\_about to canvas\_main\_menu

    frame\_about = ScrolledFrame(canvas\_main\_menu, bg = constant.DARKBGCLR, bd = 0, highlightthickness = 0, max\_height = 1000)

    frame\_about.grid(row = 0, column = 0, sticky = NSEW, padx = 5, pady = 5)

    frame\_about.viewPort.grid\_columnconfigure(1, weight = 1)

    frame\_about.viewPort.update\_idletasks()

    #Adding the title to the about page

    Label(frame\_about.viewPort, text = "ABOUT   ", font = ('Comic Sans', 30, 'bold'), bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, bd = 0, highlightthickness = 0).grid(row = 0, column = 1, sticky = NSEW, padx = 5, pady = 5)

    label\_about\_info = Label(frame\_about.viewPort, text = constant.ABOUT\_INFO, font = ('Comic Sans', 18), bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, bd = 0, highlightthickness = 0, wraplength = frame\_about.viewPort.winfo\_width() - 20, justify = LEFT)

    label\_about\_info.grid(row = 1, column = 0, columnspan = 2, sticky = NSEW, padx = 5, pady = 5)

    bind\_configure = frame\_about.bind("<Configure>", on\_frame\_about\_configure, add = "+")

    #Adding a back button

    img\_back = PhotoImage(file = "./Icons/back.png").subsample(2,2)

    button\_back = Button(frame\_about.viewPort, image = img\_back, bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, activebackground = constant.DARKBGCLR, activeforeground = constant.DARKBGTEXTCLR, font = ('Consolas', 20, 'bold'), command = on\_back, bd = 0, highlightthickness = 0)

    button\_back.grid(row = 0, column = 0, sticky = W, padx = 5, pady = 5)

#Called by main.py

def main():

    #Setting up the screen

    global wn, root\_turtle, canvas\_turtle, program\_running, theme\_bg, url, game\_result, img\_pocket\_chess\_arena\_icon, draw\_offer

    wn=turtle.Screen()

    wn.colormode(255)

    wn.bgcolor('black')

    wn.title("Pocket Chess Arena")

    wn.tracer(1)

    program\_running = True

    config = configparser.ConfigParser()

    config.read('config.ini')

    ip = config['SERVER DETAILS']['IP']

    port = config['SERVER DETAILS']['PORT']

    url = f'http://{ip}:{port}/'

    global geturl

    geturl = lambda x : url + x

    draw\_offer = None

    #Getting the canvas and top level window from wn

    canvas\_turtle = wn.getcanvas()

    canvas\_turtle.configure(bd = 0, highlightthickness = 0)

    root\_turtle = canvas\_turtle.winfo\_toplevel()

    #Setting the minsize of root\_turtle

    root\_turtle.minsize(1050, 550)

    #Setting the title bar icon

    img\_pocket\_chess\_arena\_icon = PhotoImage(file = "./Icons/pocket\_chess\_arena\_logo.png")

    root\_turtle.iconphoto(False, img\_pocket\_chess\_arena\_icon)

    #Adjusting the resolution

    if os.sys.platform.lower() == 'windows':

        ctypes.windll.shcore.SetProcessDpiAwareness(True)

    #Setting up scollbars for the turtle window

    #wn.screensize(root\_turtle.winfo\_width(), root\_turtle.winfo\_height())

    #Storing the shapes of all the pieces

    wn.register\_shape("Pawn", constant.COORD["pawn"])

    wn.register\_shape("Horse", constant.COORD["horse"])

    wn.register\_shape("Bishop", constant.COORD["bishop"])

    wn.register\_shape("Queen", constant.COORD["queen"])

    wn.register\_shape("King", constant.COORD["king"])

    wn.register\_shape("Rook", constant.COORD["rook"])

    #Changing the default font of combobox drop downs. This font size looks good only on the home page. It will be changed later for the other pages.

    font\_combo\_listbox = font.Font(family="Comic Sans", size = 17, weight = "normal")

    root\_turtle.option\_add("\*TCombobox\*Listbox\*Font", font\_combo\_listbox)

    #Configuration Variables

    global touchpieceval, vocalval, whiteview, blackview, doubleview

    touchpieceval, vocalval, whiteview, blackview, doubleview=None,None,None,None,None

    #Timer Variables

    global timer\_name, timer\_cords , neutral\_bound , active\_bound , timer\_bound ,active\_bound\_width , neutral\_bound\_width

    timer\_name = []

    neutral\_bound = '#ffe599'

    active\_bound = '#2315D4'

    active\_bound\_width = 8

    neutral\_bound\_width = 4

    timer\_cords = {

                    'doubleview':[(-365,-330),(220,-330)],

                    'whiteview':[(353,-142),(353,142)],

                    'blackview':[(353,142),(353,-142)]

            }

    timer\_bound = {

                    'doubleview':{

                                    'black':[(207,-298),(360,-328)],

                                    'white':[(-376,-296),(-227,-327)]

                            },

                    'blackview':{

                                    'white':[(343,177),(482,142)],

                                    'black':[(343,-107),(482,-142)]

                            },

                    'whiteview':{

                                    'black':[(343,177),(482,142)],

                                    'white':[(343,-107),(482,-142)]

                            }

    }

    main\_menu()

    turtle.mainloop()

game.py

#Import of Modules

import turtle

import time

import datetime

from tkinter import \*

from tkinter import messagebox

import requests

from playsound import playsound

from threading import Thread

import clipboard

import os

#Import of created files

import home

import logic

import database\_functions

import constant

import logmessage #To help in debugging

#Import of supporting game file

from utils import \*

def define\_basic\_global\_variables(touch\_move\_):

    global size, sqsize, stretch\_square, drift, white, black, name, legalmoves, touch\_move, turn, move\_stage, DRIFT, wh\_list\_piece\_numbers, bl\_list\_piece\_numbers, wh\_piece\_count, bl\_piece\_count, legaladd, code, light\_square\_clr, dark\_square\_clr, active\_piece, move\_duration, game\_result, pgn, movenumber, last\_sec

    #Initialising a chess for the chess board and a corresponding size for root\_turtle

    length = 600

    size\_canvas\_turtle = int(length) + 15

    size\_root\_turtle = size\_canvas\_turtle + 35

    home.canvas\_turtle.configure(width = size\_canvas\_turtle, height = size\_canvas\_turtle)

    home.canvas\_turtle.update()

    home.root\_turtle.geometry(f"{size\_root\_turtle}x{size\_root\_turtle+95}")

    #Making the size of the board a global variable which can be accessed everywhere

    size = length

    #Size of 1 UNIT square in the chess board

    sqsize = size/8

    #The value by which each unit square should be stretched.

    stretch\_square = (size/8)/20

    #Initialsing game\_result [None, 'W', 'B', 'D', 'S']

    game\_result = None

    #Initialising the view

    if home.mode\_of\_play == "online":

        if home.player\_side == "white":

            white = True

            black = False

        elif home.player\_side == "black":

            white = False

            black = True

    else:

        white = True

        black = False

    #Initialsing active\_piece as None because no piece has been dragged yet

    active\_piece = None

    #Making the drift of the white board a global variable...which implies that drift of the black view board can as also be accessed as it is negative of the drift of the white view board

    if white == True and black == True:

        drift = constant.DRIFT\_WV\_BOARD

    else:

        drift = 0

    #white view list of piece numbers

    wh\_list\_piece\_numbers = [[0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0]]

    wh\_piece\_count = 0

    #black view list of piece numbers

    bl\_list\_piece\_numbers = [[0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0]]

    bl\_piece\_count=0

    name = {'pawn' : 'pawn',

        'rook' : 'rook',

        'horse' : 'knight',

        'bishop' : 'bishop',

        'queen' : 'queen',

        'king' : 'king'}

    #Drawing a boundary around the 'legal moves' button on the screen

    legalmoves=True

    home.wn.colormode(255)

    #Touchpiece

    touch\_move = touch\_move\_

    #defining the colours of lightsquare colour and dark square colour

    light\_square\_clr = constant.LIGHTSQUARECLR

    dark\_square\_clr = constant.DARKSQUARECLR

    #0 = Waiting for Starting Address, 1 = waiting for Ending Address

    move\_stage = 0

    #Who's turn is it?

    turn="white"

    #List of all the possible legal squares

    legaladd = []

    #Initialising move\_duration for the timers

    move\_duration = []

    #Initialising movenumber

    movenumber = 0

    #Initialising pgn - portable game notation

    pgn = []

    #last seconds

    last\_sec = 20

#Displays A,B,C, ...H and 1,2,3...8 on the side of the board for whiteview and blackview taking into account the value of boardview

def display\_labels():

    #Deletes the OLD labels

    try:

        home.show\_labels(None, None, None, None, False, True)

    except:

        pass

    try:

        home.show\_labels(None, None, None, None, True, True)

    except:

        pass

    #Creates the NEW labels

    distance=17

    startdistance=(sqsize/2)+8

    if white==True:

        home.show\_labels(-(size/2)+drift-distance,(-size/2)-distance, sqsize, startdistance, False)

    if black == True:

        home.show\_labels(-(size/2)-drift-distance,(-size/2)-distance, sqsize,startdistance, True)

#Can be used to create a SINGLE piece at the desired board location (UNLIKE pieces\_setup in game.py which creates ALL the pieces in the BASE configuration)

def create\_chess\_piece(row, col, identity, colour, view):

    global wh\_piece\_count, bl\_piece\_count, wh\_list\_piece\_numbers, bl\_list\_piece\_numbers

    #row, col --> wrt white view / list2d

    if view == "white":

        wh\_piece\_count+=1

        wh\_list\_piece\_numbers[row][col] = wh\_piece\_count

        piece\_count = wh\_piece\_count

    elif view == "black":

        bl\_piece\_count+=1

        blrow = 7- row

        blcol = 7 - col

        bl\_list\_piece\_numbers[blrow][blcol] = bl\_piece\_count

        piece\_count = bl\_piece\_count

    globals()[f'{view[:2]}{piece\_count}']=turtle.Turtle(visible=False)

    globals()[f'{view[:2]}{piece\_count}'].ht()

    globals()[f'{view[:2]}{piece\_count}'].pu()

    globals()[f'{view[:2]}{piece\_count}'].shape(identity.title())

    stretch\_piece = stretch\_square \* 0.2

    border\_width = int(stretch\_piece \* 4)

    globals()[f'{view[:2]}{piece\_count}'].shapesize(stretch\_piece, stretch\_piece, border\_width)

    globals()[f'{view[:2]}{piece\_count}'].speed(8)

    if colour == "white":

        globals()[f'{view[:2]}{piece\_count}'].color(constant.WHITEPIECECLR)

        globals()[f'{view[:2]}{piece\_count}'].pencolor("black")

    elif colour == "black":

        globals()[f'{view[:2]}{piece\_count}'].color(constant.BLACKPIECECLR)

        globals()[f'{view[:2]}{piece\_count}'].pencolor("black")

    globals()[f'{view[:2]}{piece\_count}'].goto(coord\_from\_add(row, col, view))

    globals()[f'{view[:2]}{piece\_count}'].st()

#Return: Central pixel coordinates as a tuple (IMP: Using the global variable boardview, it also acknowledges the changes in BOARD VIEW and returns the coordinates accordingly)

def coord\_from\_add(row, col, view):

    step=size/8

    if view == "white":

        return ((col\*step) - (4\*step) + (step/2) + drift, (4\*step) - (row\*step) - (step/2))

    elif view == "black":

        return ( -1 \* ((col\*step) - (4\*step) + (step/2)) - drift, -1 \* ((4\*step) - (row\*step) - (step/2)))

#Deletes all the existing turtle chess pieces on the board, creates new turtle chess pieces for the current configuration. Updates/Resets wh\_list\_of\_piece\_numbers, bl\_list\_of\_piece\_numbers, wh\_piece\_count, bl\_piece\_count.

def configure\_pieces():

    global wh\_list\_piece\_numbers, bl\_list\_piece\_numbers, wh\_piece\_count, bl\_piece\_count

    #Initialisation of some variables

    try:

        del wh\_list\_piece\_numbers, bl\_list\_piece\_numbers, wh\_piece\_count, bl\_piece\_count

    except:

        pass

    wh\_list\_piece\_numbers = [[0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0]]

    bl\_list\_piece\_numbers = [[0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0]]

    wh\_piece\_count = 0

    bl\_piece\_count = 0

    if white == True:

        for i in range(8):

            for j in range(8):

                if logic.list2d[i][j] != logic.emp:

                    create\_chess\_piece(i, j, logic.list2d[i][j][0], logic.colour(logic.list2d[i][j]), "white")

    if black == True:

        for i in range(8):

            for j in range(8):

                if logic.list2d[i][j] != logic.emp:

                    create\_chess\_piece(i, j, logic.list2d[i][j][0], logic.colour(logic.list2d[i][j]), "black")

def configure\_board():

    #For the individual square turtles, identification is first row number and then column number in the variable name.

    #White's View

    if white==True:

        for x in range(len(wh\_list\_piece\_numbers)):

            for y in range(len(wh\_list\_piece\_numbers[x])):

                globals()[f't{x}{y}'] = turtle.Turtle()

                globals()[f't{x}{y}'].shape('square')

                globals()[f't{x}{y}'].shapesize(stretch\_square,stretch\_square)

                globals()[f't{x}{y}'].up()

                if (x+y)%2==0:

                    globals()[f't{x}{y}'].color(light\_square\_clr)

                elif (x+y)%2==1:

                    globals()[f't{x}{y}'].color(dark\_square\_clr)

                globals()[f't{x}{y}'].goto((sqsize/2) + sqsize\*(y-4)+drift , -(sqsize/2) + sqsize\*(4-x))

    if black==True:

        #Black's View

        for x in range(len(bl\_list\_piece\_numbers)):

            for y in range(len(bl\_list\_piece\_numbers[x])):

                globals()[f'tb{x}{y}'] = turtle.Turtle()

                globals()[f'tb{x}{y}'].shape('square')

                globals()[f'tb{x}{y}'].shapesize(stretch\_square,stretch\_square)

                globals()[f'tb{x}{y}'].up()

                if (x+y)%2==0:

                    globals()[f'tb{x}{y}'].color(light\_square\_clr)

                elif (x+y)%2==1:

                    globals()[f'tb{x}{y}'].color(dark\_square\_clr)

                globals()[f'tb{x}{y}'].goto((sqsize/2) + sqsize\*(y-4)-drift , -(sqsize/2) + sqsize\*(4-x))

    display\_labels()

def configure\_game():

    global whsrow, whscol, blsrow, blscol

    home.wn.tracer(0)

    #Hiding and Deleting all the pieces on the screen

    for j, k in zip((wh\_piece\_count, bl\_piece\_count), ('wh', 'bl')):

        for i in range(1,j+1):

            try:

                globals()[f'{k}{i}'].ht()

                globals()[f'{k}{i}'].clear()

                del globals()[f'{k}{i}']

            except:

                pass

    #Hiding and Deleting all the board turtles on the screen

    for x in range(len(wh\_list\_piece\_numbers)):

        for y in range(len(wh\_list\_piece\_numbers[x])):

            try:

                globals()[f't{x}{y}'].ht()

                globals()[f't{x}{y}'].clear()

                del globals()[f't{x}{y}']

            except:

                pass

    for x in range(len(bl\_list\_piece\_numbers)):

        for y in range(len(bl\_list\_piece\_numbers[x])):

            try:

                globals()[f'tb{x}{y}'].ht()

                globals()[f'tb{x}{y}'].clear()

                del globals()[f'tb{x}{y}']

            except:

                pass

    configure\_board()

    configure\_pieces()

    try:

        if legalmoves == True:

            show\_legal\_squares(True)

        try:

            pwhsrow, pwhscol, pwherow, pwhecol = prev\_move\_add

            for row,col in ((pwhsrow, pwhscol), (pwherow, pwhecol)):

                change\_square\_clr(row, col, 7-row, 7-col, constant.SELECTEDLIGHTSQUARECLR, constant.SELECTEDDARKSQUARECLR)

        except:

            pass

        change\_square\_clr(whsrow, whscol, blsrow, blscol, constant.SELECTEDLIGHTSQUARECLR, constant.SELECTEDDARKSQUARECLR)

        change\_square\_clr(wherow, whecol, blerow, blecol, constant.SELECTEDLIGHTSQUARECLR, constant.SELECTEDDARKSQUARECLR)

    except:

        pass

    display\_check\_and\_checkmate(logic.list2d, constant.CHECKSQUARECLR, True, False)

    display\_stalemate(logic.list2d, False)

    if game\_result == "draw":

        home.on\_draw(False)

    home.wn.tracer(1)

#Parameters: WV-row num, WV-col num, BV-row num, BV-col num, light clr, dark clr, should the change be updated on the screen using update or tracer?

def change\_square\_clr(whsrow, whscol, blsrow, blscol, light, dark, update=True):

    add="filled"

    if wh\_list\_piece\_numbers[whsrow][whscol] == 0:

        add="empty"

    home.wn.tracer(0)

    if (whsrow+whscol)%2==0: #Lighter Squares

        if white==True:

            globals()[f't{whsrow}{whscol}'].color(light)

            if add!="empty":

                globals()[f'wh{wh\_list\_piece\_numbers[whsrow][whscol]}'].st()

        if black==True:

            globals()[f'tb{blsrow}{blscol}'].color(light)

            if add!="empty":

                globals()[f'bl{bl\_list\_piece\_numbers[blsrow][blscol]}'].st()

    elif (whsrow+whscol)%2==1: #Darker Squares

        if white==True:

            globals()[f't{whsrow}{whscol}'].color(dark)

            if add!="empty":

                globals()[f'wh{wh\_list\_piece\_numbers[whsrow][whscol]}'].st()

        if black==True:

            globals()[f'tb{blsrow}{blscol}'].color(dark)

            if add!="empty":

                globals()[f'bl{bl\_list\_piece\_numbers[blsrow][blscol]}'].st()

    if update==True:

        home.wn.tracer(1)

def show\_legal\_squares(show):

    global legaladd

    for m,n in legaladd:

        blm, bln=7-m, 7-n

        if show == True:

            change\_square\_clr(m, n, blm, bln, constant.LEGALLIGHTSQUARECLR, constant.LEGALDARKSQUARECLR, update=False)

        elif show == False:

            change\_square\_clr(m, n, blm, bln, light\_square\_clr, dark\_square\_clr, update=False)

#Parameters: 2 dimensional list, hexadecimal code for the king's square colour when there is a check, if there is a checkmate should it be declared?

def display\_check\_and\_checkmate(f2d, pink, declare, animate = True):

    global mouse\_vacant, drift, game\_result

    toggleDEBUG=False

    if logmessage.DEBUG==True:

        logmessage.DEBUG=False #No debug info to be PRINTED on the screen until the display\_check\_and\_checkmate function gets over

        toggleDEBUG=True #Debug will be changed back to True at the end of this function

    change=False

    roww, colw=logic.whking[2], logic.whking[3]

    rowb, colb=logic.blking[2], logic.blking[3]

    for i, j in ((roww, colw), (rowb, colb)):

        home.wn.tracer(0)

        if logic.check((i,j), f2d)[0]==True:

            if white==True:

                if globals()[f't{i}{j}'].color()!=(pink,pink):

                    globals()[f't{i}{j}'].color(pink)

                    globals()[f'wh{wh\_list\_piece\_numbers[i][j]}'].st()

                    change=True

            if black==True:

                if globals()[f'tb{7-i}{7-j}'].color()!=(pink,pink):

                    globals()[f'tb{7-i}{7-j}'].color(pink)

                    globals()[f'bl{bl\_list\_piece\_numbers[7-i][7-j]}'].st()

                    change=True

        elif logic.check((i,j), f2d)[0]==False:

            if white==True:

                if globals()[f't{i}{j}'].color()==(pink,pink):

                    change=True

                    if (i+j)%2==0:

                        globals()[f't{i}{j}'].color(light\_square\_clr)

                        globals()[f'wh{wh\_list\_piece\_numbers[i][j]}'].st()

                    elif (i+j)%2==1:

                        globals()[f't{i}{j}'].color(dark\_square\_clr)

                        globals()[f'wh{wh\_list\_piece\_numbers[i][j]}'].st()

            if black==True:

                if globals()[f'tb{7-i}{7-j}'].color()==(pink,pink):

                    change=True

                    if (i+j)%2==0:

                        globals()[f'tb{7-i}{7-j}'].color(light\_square\_clr)

                        globals()[f'bl{bl\_list\_piece\_numbers[7-i][7-j]}'].st()

                    elif (i+j)%2==1:

                        globals()[f'tb{7-i}{7-j}'].color(dark\_square\_clr)

                        globals()[f'bl{bl\_list\_piece\_numbers[7-i][7-j]}'].st()

        home.wn.tracer(1)

        if logic.checkmate((i,j), f2d)==True and declare==True and animate == False:

            if white==True:

                globals()[f't{i}{j}'].color("red")

                globals()[f'wh{wh\_list\_piece\_numbers[i][j]}'].st()

            if black==True:

                globals()[f'tb{7-i}{7-j}'].color("red")

                globals()[f'bl{bl\_list\_piece\_numbers[7-i][7-j]}'].st()

            home.wn.update()

        elif logic.checkmate((i,j), f2d)==True and declare==True and animate == True:

            if home.volume\_toggle == True:

                #Sound for checkmate

                playsound("./Sounds/checkmate.mp3", False)

            home.wn.tracer(0)

            for \_ in range(2):

                time.sleep(0.75)

                if (i+j)%2==0:

                    if white==True:

                        globals()[f't{i}{j}'].color(light\_square\_clr)

                        globals()[f'wh{wh\_list\_piece\_numbers[i][j]}'].st()

                    if black==True:

                        globals()[f'tb{7-i}{7-j}'].color(light\_square\_clr)

                        globals()[f'bl{bl\_list\_piece\_numbers[7-i][7-j]}'].st()

                    home.wn.update()

                elif (i+j)%2==1:

                    if white==True:

                        globals()[f't{i}{j}'].color(dark\_square\_clr)

                        globals()[f'wh{wh\_list\_piece\_numbers[i][j]}'].st()

                    if black==True:

                        globals()[f'tb{7-i}{7-j}'].color(dark\_square\_clr)

                        globals()[f'bl{bl\_list\_piece\_numbers[7-i][7-j]}'].st()

                    home.wn.update()

                time.sleep(0.75)

                if \_==1:

                    if white==True:

                        globals()[f't{i}{j}'].color("red")

                        globals()[f'wh{wh\_list\_piece\_numbers[i][j]}'].st()

                    if black==True:

                        globals()[f'tb{7-i}{7-j}'].color("red")

                        globals()[f'bl{bl\_list\_piece\_numbers[7-i][7-j]}'].st()

                    home.wn.update()

                else:

                    if white==True:

                        globals()[f't{i}{j}'].color(pink)

                        globals()[f'wh{wh\_list\_piece\_numbers[i][j]}'].st()

                    if black==True:

                        globals()[f'tb{7-i}{7-j}'].color(pink)

                        globals()[f'bl{bl\_list\_piece\_numbers[7-i][7-j]}'].st()

                    home.wn.update()

            if turn=="white":

                game\_result = "W"

            elif turn=="black":

                game\_result = "B"

            home.on\_game\_over()

            #The required information about database storage has been taken from the user. Now on clicking the close button, the program can directly close without any pass statements

            def on\_turtle\_close():

                home.program\_running = False

                home.root\_turtle.after(500, lambda: home.root\_turtle.destroy())

                database\_functions.close\_connection()

            home.root\_turtle.protocol("WM\_DELETE\_WINDOW", on\_turtle\_close)

    if toggleDEBUG==True:

        logmessage.DEBUG=True

    return change

#Parameters: 2 dimensional list

def display\_stalemate(f2d, animate = True):

    global mouse\_vacant, move\_duration, stalemate\_boundary, game\_result

    #If either of the kings are under check, definitely, it cannot be a stalemate.

    if logic.get\_game\_situation(logic.list2d)[0]:

        return

    if logic.stalemate(logic.list2d, False)==True:

        if home.volume\_toggle == True and animate:

            #Sound of stalemate

            playsound("./Sounds/stalemate.mp3")

            game\_result = "S"

            home.on\_game\_over()

        if turn=="black":

            move\_duration += [1]

            #print(move\_duration)

        elif turn=="white":

            move\_duration += [1]

            #print(move\_duration)

        home.wn.tracer(0)

        try:

            stalemate\_boundary.clear()

        except:

            pass

        home.wn.update()

        stalemate\_boundary=turtle.Turtle()

        stalemate\_boundary.ht()

        stalemate\_boundary.width(7)

        stalemate\_boundary.color("red")

        for sign in (1,-1):

            stalemate\_boundary.pu()

            stalemate\_boundary.goto(-(size/2)+sign\*drift,(size/2))

            stalemate\_boundary.pd()

            for \_ in range(4):

                stalemate\_boundary.fd(size)

                stalemate\_boundary.rt(90)

            if white==True and black==True:

                pass

            else:

                break

        home.wn.tracer(1)

        def on\_turtle\_close():

            home.program\_running = False

            home.root\_turtle.after(500, lambda: home.root\_turtle.destroy())

            database\_functions.close\_connection()

        home.root\_turtle.protocol("WM\_DELETE\_WINDOW", on\_turtle\_close)

#Aim: Displaying the board in a more readible form on the COMMAND line

#Parameters: 2 dimensional list

def display(board):

    return

    for y in range(len(board)):

        for x in range(len(board[y])):

            print(board[y][x] , end = ' ')

        print()

#Aim: Create a tkinter window and ask the users if they want to store the game details to the database. Also take care of network failure exceptions. If users agree, the game details to be uploaded to the database.

def add\_to\_db(date, start\_time, name\_white, name\_black, result, moves:list, times:list, min\_time = 0, increment = 0):

    if home.mode\_of\_play == 'online':

        if not home.challenger:

            return

    min\_time = int(min\_time)\*60 if 1/min\_time else 0

    if min\_time:

        increment = int(increment)

    else:

        increment = 0

    end\_time=datetime.datetime.now().strftime("%H:%M:%S")

    FMT = '%H:%M:%S'

    duration = datetime.datetime.strptime(end\_time, FMT) - datetime.datetime.strptime(start\_time, FMT)

    times = [tfor(times[r]).replace(' ','') for r in range(len(times))]

    while True:

        value = messagebox.askquestion("Store Game", "Save Game to Database?\nRequires Network Connection")

        if value=="yes" and database\_functions.check\_connection()==False:

            database\_functions.open\_connection()

        if value=="yes" and database\_functions.check\_connection()==True:

            database\_functions.update\_game\_details(date, start\_time, end\_time, duration, name\_white, name\_black, result, moves, times, min\_time, increment)

            #After all the details have been uploaded to the database, displaying a message to the user.

            messagebox.showinfo("Status", "Successfully uploaded to database")

            break

        elif value=="no":

            break

        elif value=="yes" and database\_functions.check\_connection()==False:

            messagebox.showinfo("Error", "No Internet Connection")

def get\_turtle\_coord(x,y):

    home.canvas\_turtle.update()

    width = home.canvas\_turtle.winfo\_width()

    height = home.canvas\_turtle.winfo\_height()

    return x - (width/2), (height/2) - y

def get\_tkinter\_coord(x,y):

    home.canvas\_turtle.update()

    width = home.canvas\_turtle.winfo\_width()

    height = home.canvas\_turtle.winfo\_height()

    return x + (width/2), (height/2) - y

def main\_updater\_offline():

    global old, whitetime, blacktime , move\_duration, winner, game\_result

    old = time.time()

    while home.program\_running and (game\_result is None):

        if 1/home.min\_time != 0 and timer\_start:

            if turn == 'white':

                whitetime -= (time.time() - old)

                old = time.time()

                if whitetime < last\_sec:

                    time\_left = tfor(whitetime)

                else:

                    time\_left = tfor(int(whitetime))[:-2]

                home.label\_white\_timer\_bottom.configure(text = time\_left)

                home.label\_white\_timer\_top.configure(text = time\_left)

                if whitetime < 0:

                    game\_result = 'B' if logic.winner\_on\_flag('black') else 'D'

                    home.on\_game\_over()

            elif turn == 'black':

                blacktime -= (time.time() - old)

                old = time.time()

                if blacktime < last\_sec:

                    time\_left = tfor(blacktime)

                else:

                    time\_left = tfor(int(blacktime))[:-2]

                home.label\_black\_timer\_bottom.configure(text = time\_left)

                home.label\_black\_timer\_top.configure(text = time\_left)

                if blacktime < 0:

                    game\_result = 'W' if logic.winner\_on\_flag('white') else 'D'

                    home.on\_game\_over()

        home.wn.tracer(0)

        home.wn.update()

def main\_updater\_online():

    global ready\_to\_play, turn, game\_date , game\_start\_time

    while home.mode\_of\_play == "online" and game\_result is None:

        if (home.challenger and home.received) or ((not home.challenger) and home.accepted):

            ready\_to\_play = True

            while turn != home.player\_side:

                res = requests.get(home.geturl('rooms') , params={'code':home.code})

                out = process\_response(res)

                time.sleep(0.2)

                home.wn.update()

        home.wn.tracer(0)

        home.wn.update()

def process\_response(response):

    global mouse\_vacant, game\_result, turn, o, whsrow, whscol, blsrow, blscol, srow, scol, whecol, wherow, blecol, blerow, division, started, oldtime, timer\_start

    try:

        r = response.json()#[str(home.code)]

        if 'side' in r:

            if r['side'] != home.player\_side:

                sr,sc,er,ec = [int(t) for t in tuple(r['move'])]

                process\_move((sr,sc),(er,ec),r['pppiece'])

                timer\_start = True

            return False

        return r

    except Exception as e:

        return e

def connection():

    global timer\_start, game\_result

    t0 = int(time.time())

    while game\_result is None:

        if 1:

            if ((not home.challenger) and home.accepted) or (home.challenger and home.received):#or ((not home.challenger) and home.accepted):

                connectoption = True if (int(time.time())%30 == 0 and int(time.time()) != t0) else False

                if connectoption:

                    t0 = int(time.time())

                payload = {'code' : home.code , 'side' : home.player\_side , 'connectop' : connectoption}

                res = requests.get(home.geturl('connection') , params=payload)

                data = res.json()

                if data['offers']['status'] == 'gameover':

                    game\_result = data['offers']['game result']

                    if data['offers']['resignations'] == home.oppside:

                        messagebox.showinfo('YOU WON' , 'on resignation')

                        home.on\_game\_over()

                    elif data['offers']['flagged']:

                        if data['offers']['flagged'] == 'white':

                            game\_result = 'W' if logic.winner\_on\_flag('white') else 'D'

                            home.on\_game\_over()

                        else :

                            game\_result = 'B' if logic.winner\_on\_flag('black') else 'D'

                            home.on\_game\_over()

                    elif data['offers']['lost connection'] == home.oppside:

                        messagebox.showinfo('YOU WON' , 'game abandoned')

                        home.on\_game\_over()

                    elif data['offers']['draw offers'] == home.oppside:

                        if messagebox.askyesno('DRAW OFFER' , 'Your opponent has offered a draw, accept?'):

                            requests.post(home.geturl('connection') , data={'code':home.code , 'side':home.player\_side , 'draw':'accepted'})

                            home.on\_game\_over()

                        else:

                            requests.post(home.geturl('connection') , data={'code':home.code , 'side':home.player\_side , 'draw':'rejected'})

                if 1/home.min\_time:

                    wt,bt = data['timers']['white'][0] , data['timers']['black'][0]

                    if bt < last\_sec:

                        home.label\_black\_timer\_bottom.configure(text=tfor(bt))

                        home.label\_black\_timer\_top.configure(text=tfor(bt))

                    else:

                        home.label\_black\_timer\_bottom.configure(text=tfor(bt)[:-2])

                        home.label\_black\_timer\_top.configure(text=tfor(bt)[:-2])

                    if wt < last\_sec:

                        home.label\_white\_timer\_bottom.configure(text=tfor(wt))

                        home.label\_white\_timer\_top.configure(text=tfor(wt))

                    else:

                        home.label\_white\_timer\_bottom.configure(text=tfor(int(wt))[:-2])

                        home.label\_white\_timer\_top.configure(text=tfor(int(wt))[:-2])

                    if bt > last\_sec and wt > last\_sec:

                        time.sleep(0.4)

                else:

                    time.sleep(0.4)

        try:

            pass

        except Exception as e:

            print(e)

            continue

def generate\_pgn():

    global pgn, img\_copy, button\_pgn\_copy

    def on\_pgn\_copy():

        global button\_pgn\_copy

        if os.sys.platform.lower()=='windows':

            clipboard.copy(pgn\_str)

        else:

            os.system(f'echo -n \'{pgn\_str}\'| xclip -selection clipboard')

        button\_pgn\_copy.configure(text = " PGN copied!")

        home.root\_turtle.after(5000, lambda : button\_pgn\_copy.configure(text = " Copy PGN to clipboard"))

    #Adding some additional details to pgn

    pgn\_str = ' '.join(pgn)

    pgn\_result = '\*'

    if game\_result == "W":

        pgn\_result = "1-0"

    elif game\_result == "B":

        pgn\_result = "0-1"

    elif game\_result in ('D', 'S'):

        pgn\_result = "1/2-1/2"

    pgn\_str = f'[Site "Pocket Chess Arena"]\n[White "{home.white\_name}"]\n[Black "{home.black\_name}"]\n[Result "{pgn\_result}"]\n\n' + pgn\_str + " " + pgn\_result

    img\_copy = PhotoImage(file = "./Icons/clipboard.png").subsample(3,3)

    button\_pgn\_copy = Button(home.frame\_pgn, image = img\_copy, text = "Copy PGN to clipboard", compound = LEFT, font = ('Comic Sans', 14, 'bold'), bg = constant.LIGHTBGCLR, fg = constant.LIGHTBGTEXTCLR, activebackground = constant.LIGHTBGCLR, command = on\_pgn\_copy, bd = 0, highlightthickness = 0)

    button\_pgn\_copy.grid(row = 0, column = 0, sticky = NSEW)

#Main function of game.py which is called by home.py to handle the actual game (settings, instructions NOT included)

#Parameters: size-Size of the chess board, drift-Drift of the white's view board, white-white view==>True/False, black-black view==>True/False, Speech=True/False, Touchpiece=True/False

def game\_main(touch\_move\_):

    global process\_move

    #Handling the close of the main turtle window

    def on\_turtle\_close():

        if home.mode\_of\_play == 'offline':

            abort = messagebox.askyesno('Abort Game', 'Are you sure?')

        else:

            abort = messagebox.askyesno('Abandon', 'Do you wish to quit?')

        if abort:

            home.program\_running = False

            home.root\_turtle.after(500, lambda: home.root\_turtle.destroy())

            database\_functions.close\_connection()

    #Move a piece from the Start address to the End address (S to E)

    def make\_move(whsrow, whscol, blsrow, blscol, wherow, whecol, blerow, blecol, unitmovement):

        #print(whsrow, whscol, blsrow, blscol, wherow, whecol, blerow, blecol)

        fraction\_value=1/7

        divisions = 0

        if white==True:

            whcurrx, whcurry=globals()[f'wh{wh\_list\_piece\_numbers[whsrow][whscol]}'].pos()

            whdiffx, whdiffy=sqsize/2 + sqsize\*(whecol-4)+drift-whcurrx, -(sqsize/2) + sqsize\*(4-wherow)-whcurry

            abswhdiffx, abswhdiffy=abs(whdiffx), abs(whdiffy)

            if white==True and black==True: #Double View

                unitmovement=fraction\_value \* max(abswhdiffx, abswhdiffy)

                if unitmovement>=40:

                    unitmovement=45

                else:

                    unitmovement=25

            if int(abswhdiffx)>0:

                divisions=int(abswhdiffx/unitmovement)

            elif int(abswhdiffy)>0:

                divisions=int(abswhdiffy/unitmovement)

        if black==True:

            blcurrx, blcurry=globals()[f'bl{bl\_list\_piece\_numbers[blsrow][blscol]}'].pos()

            bldiffx, bldiffy=sqsize/2 + sqsize\*(blecol-4)-drift-blcurrx, -(sqsize/2) + sqsize\*(4-blerow)- blcurry

            absbldiffx, absbldiffy=abs(bldiffx), abs(bldiffy)

            if white==True and black==True: #Double View

                unitmovement=fraction\_value \* max(absbldiffx, absbldiffy)

                if unitmovement>=40:

                    unitmovement=45

                else:

                    unitmovement=25

            if int(absbldiffx)>0:

                divisions=int(absbldiffx/unitmovement)

            elif int(absbldiffy)>0:

                divisions=int(absbldiffy/unitmovement)

        if divisions == 0:

            divisions = 1

        fraction=1/divisions

        for i in range(1,divisions+1):

            home.wn.tracer(0)

            if white==True:

                #Changes in White's View

                globals()[f'wh{wh\_list\_piece\_numbers[whsrow][whscol]}'].goto( whcurrx + (i\*fraction\*whdiffx), whcurry+ (i\* fraction \* whdiffy)) #Drift is added becuase central x has to be converted to actual x as this line is for DISPLAY.

            if black==True:

                #Changes in Black' View

                globals()[f'bl{bl\_list\_piece\_numbers[blsrow][blscol]}'].goto(blcurrx+ (i\*fraction\*bldiffx) ,blcurry+ (i\*fraction\*bldiffy)) #Drift is added becuase central x has to be converted to actual x as this line is for DISPLAY.

            home.wn.update()

        home.wn.tracer(1)

    #Hiding all the DEAD pieces on the board

    def hide\_dead\_pieces(enpassant, enpassant\_killpawn\_add, wherow, whecol, blerow, blecol):

        if enpassant==True:

            if white==True:

                globals()[f'wh{wh\_list\_piece\_numbers[enpassant\_killpawn\_add[0]][enpassant\_killpawn\_add[1]]}'].ht()

            if black==True:

                globals()[f'bl{bl\_list\_piece\_numbers[7-enpassant\_killpawn\_add[0]][7-enpassant\_killpawn\_add[1]]}'].ht()

        #If the below condition is satisfied, there was a piece in the ending address and a capture took place

        if logic.list2d\_start\_of\_current\_move[wherow][whecol] != logic.emp:

            if white==True:

                globals()[f'wh{wh\_list\_piece\_numbers[wherow][whecol]}'].ht()

            if black==True:

                globals()[f'bl{bl\_list\_piece\_numbers[blerow][blecol]}'].ht()

    def promote\_pawn(wherow, whecol, blerow, blecol):

        global pawnboardturtle, pc, mouse\_vacant, movestr

        turn\_pawnreachedend = turn

        pawnboardturtle=turtle.Turtle(visible=False)

        pawnboardturtle.color("red", constant.PAWNPROMOTIONWINDOWCLR)

        pawnboardturtle.shape("square")

        pawnboardturtle.st()

        scale\_ppboard = int(size \* 13/560)

        for i in range(1,scale\_ppboard):

            pawnboardturtle.shapesize(i,i)

        pccolor={"white": ("black", "white"), "black": ("black", constant.BLACKPIECECLR)}

        pcname=["Horse", "Rook", "Bishop", "Queen"]

        pcsign=((1, 1), (-1, 1), (-1,-1), (1,-1))

        pc={}

        def pcchanger(x,y):

            global move\_duration, wh\_piece\_count, bl\_piece\_count

            inactive\_square\_size = size \* 90/560

            if -inactive\_square\_size<x<inactive\_square\_size and -inactive\_square\_size<y<inactive\_square\_size:

                pass

            else:

                return None

            for i in range(4):

                if x/pcsign[i][0]>30 and y/pcsign[i][1]>25:

                    global pawnboardturtle, pc

                    for k in pc:

                        pc[k].ht()

                    for k in range(scale\_ppboard,0,-1):

                        pawnboardturtle.shapesize(k,k)

                    pawnboardturtle.ht()

                    logic.pawnpromotion((wherow, whecol), pcname[i].lower())

                    home.wn.tracer(0)

                    if white==True:

                        globals()[f'wh{wh\_list\_piece\_numbers[wherow][whecol]}'].shape(pcname[i])

                    if black==True:

                        globals()[f'bl{bl\_list\_piece\_numbers[blerow][blecol]}'].shape(pcname[i])

                    home.wn.tracer(1)

                    if home.mode\_of\_play == 'online':

                        requests.post(

                            home.geturl('rooms'),

                            data = {

                                'code':home.code,

                                'side':home.player\_side,

                                'move':movestr,

                                'pppiece':pcname[i].lower()

                            }

                        )

                    else:

                        move\_duration += [-1]

                    toggle\_turns(False, pcname[i])

                    home.wn.onclick(None)

                    home.canvas\_turtle.bind("<ButtonPress>", lambda e: get\_move(e.x, e.y, "press"))

                    home.enable\_game\_controls()

                    display\_check\_and\_checkmate(logic.list2d, (224, 111, 111), True)

                    display\_stalemate(logic.list2d)

        pawnpcdistance=size \* 60 / 560

        for i in range(4):

            home.wn.tracer(0)

            pc[f'pawnpromotion{pcname[i]}'] =turtle.Turtle(visible = False)

            pc[f'pawnpromotion{pcname[i]}'].pu()

            stretch\_piece = stretch\_square \* 0.2

            border\_width = int(stretch\_piece \* 4)

            pc[f'pawnpromotion{pcname[i]}'].shapesize(stretch\_piece, stretch\_piece, border\_width)

            pc[f'pawnpromotion{pcname[i]}'].color(pccolor[turn\_pawnreachedend][0], pccolor[turn\_pawnreachedend][1])

            pc[f'pawnpromotion{pcname[i]}'].shape(pcname[i])

            pc[f'pawnpromotion{pcname[i]}'].st()

        fractions = 10

        for j in range(fractions):

            home.wn.tracer(0)

            for i in range(4):

                pc[f'pawnpromotion{pcname[i]}'].goto((j/fractions)\*pcsign[i][0]\*pawnpcdistance, (j/fractions)\*pcsign[i][1]\*pawnpcdistance)

            home.wn.tracer(1)

        mouse\_vacant = False

        home.canvas\_turtle.unbind("<ButtonPress>")

        home.wn.onclick(pcchanger)

        home.disable\_game\_controls()

    #Parameters: Pawn promotion?, checkmate?, stalemate?

    def toggle\_turns(pawnv, pppiece = ''):

        global turn, move\_duration, castlev, old, whitetime, blacktime, movenumber

        checkv\_, checkmatev\_, stalematev\_ = logic.get\_game\_situation(logic.list2d)

        chess\_not = get\_chess\_notation(logic.list2d\_start\_of\_current\_move, whsrow, whscol, wherow, whecol, (checkv\_, checkmatev\_), castlev, pppiece)

        label\_chess\_not = Label(home.frame\_moves.viewPort, text = chess\_not, bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('Comic Sans', 15, 'bold'), bd = 2, highlightthickness = 2, highlightbackground = constant.LIGHTBGTEXTCLR)

        if turn=="white":

            movenumber += 1

            Label(home.frame\_moves.viewPort, text = home.chess\_not\_row + 1, bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('Comic Sans', 15, 'bold'), bd = 2, highlightthickness = 2, highlightbackground = constant.LIGHTBGTEXTCLR).grid(row = home.chess\_not\_row, column = 0, sticky = NSEW, padx = 1, pady = 1, ipadx = 5, ipady = 5)

            label\_chess\_not.grid(row = home.chess\_not\_row, column = 1, sticky = NSEW, padx = 1, pady = 1, ipady = 5)

            #Updating pgn

            pgn.append(str(home.chess\_not\_row + 1) + '.')

            pgn.append(chess\_not)

            if checkmatev\_!=True and pawnv!=True and stalematev\_!=True:

                turn="black"

            #Make necessary changes to the timers

            if home.mode\_of\_play == 'offline' and not pawnv:

                if not (checkmatev\_ or stalematev\_):

                    whitetime += home.increment

                move\_duration += [(home.min\_time\*60 - whitetime  + movenumber\*home.increment- sum([x for x in move\_duration[:-1:2] if x <= 0]))]

                if whitetime < last\_sec:

                    home.label\_white\_timer\_top.configure(text = tfor(whitetime))

                    home.label\_white\_timer\_bottom.configure(text = tfor(whitetime))

                else:

                    home.label\_white\_timer\_top.configure(text = tfor(int(whitetime))[:-2])

                    home.label\_white\_timer\_bottom.configure(text = tfor(int(whitetime))[:-2])

        elif turn=="black":

            label\_chess\_not.grid(row = home.chess\_not\_row, column = 2, sticky = NSEW, padx = 1, pady = 1, ipady = 5)

            #Updating pgn

            pgn.append(chess\_not)

            home.chess\_not\_row += 1

            if checkmatev\_!=True and pawnv!=True and stalematev\_!=True:

                turn="white"

            #Make necessary changes to the timers

            if home.mode\_of\_play == 'offline':

                if not(checkmatev\_ or stalematev\_):

                    blacktime += home.increment

                move\_duration += [(home.min\_time\*60 - blacktime  + movenumber\*home.increment- sum([x for x in move\_duration[1:-1:2] if x <= 0]))]

                if blacktime < last\_sec:

                    home.label\_black\_timer\_top.configure(text = tfor(blacktime))

                    home.label\_black\_timer\_bottom.configure(text = tfor(blacktime))

                else:

                    home.label\_black\_timer\_top.configure(text = tfor(int(blacktime))[:-2])

                    home.label\_black\_timer\_bottom.configure(text = tfor(int(blacktime))[:-2])

        home.frame\_moves.canvas.yview\_moveto('1.0')

        home.frame\_moves.update()

    def reset\_move():

        global legaladd

        home.wn.tracer(0)

        #Reverting the colour of the starting square if the user clicks on it again

        if legalmoves==True:

            show\_legal\_squares(False)

            legaladd = []

        if display\_check\_and\_checkmate(logic.list2d, (224, 111, 111), False) == False:

            change\_square\_clr(whsrow, whscol, blsrow, blscol, light\_square\_clr, dark\_square\_clr)

        #Displaying the squares of the previous in case they are changed due to show\_legal\_squares(False)

        try:

            pwhsrow, pwhscol, pwherow, pwhecol = prev\_move\_add

            for row,col in ((pwhsrow, pwhscol), (pwherow, pwhecol)):

                change\_square\_clr(row, col, 7-row, 7-col, constant.SELECTEDLIGHTSQUARECLR, constant.SELECTEDDARKSQUARECLR)

        except:

            pass

        home.wn.tracer(1)

    #type: "press", "motion", "release"

    def get\_move(x, y, mode):

        global move\_stage, blscol, blsrow, whscol, whsrow, turn, legaladd, mouse\_vacant, move\_duration, active\_piece, movestr, timer\_start, old

        # scol-start address col num(RAW data as seen directly from the board on which the user clicks. For the numbering, the boards are considered to be matrices)

        # srow-start address row num('Same as above')

        # whscol-start address col num(white view list of piece numbers)

        # whsrow-start address row num(white PB)

        # blscol-start address col num(black PB)

        # blsrow-start address row num(black PB)

        # whecol-end address col num(white PB)

        # wherow-end address row num(white PB)

        # blecol-end address col num(black PB)

        # blerow-end address row num(black PB)

        #Irrespective of white and black view, srow and scol are the rownumber and the columnnumber considering both the boards to be MATRICES.

        #Irrespective of white and black view, erow and ecol are the rownumber and the columnnumber considering both the boards to be MATRICES

        x,y = get\_turtle\_coord(x,y)

        if ready\_to\_play==False:

            return

        if home.mode\_of\_play == 'online':

            if home.player\_side != turn:

                return

        mouse\_vacant = True

        x = shift\_to\_centre(x)

        if move\_stage==0 and mode == "press":

            #print("New press/piece")

            srow,scol = get(x,y)

            whsrow, whscol, blsrow, blscol = whiteview\_blackview\_equivalents(srow,scol)

            #Checking if it is a valid piece

            if  0<=whsrow<=7 and 0<=whscol<=7 and logic.list2d[whsrow][whscol] != logic.emp :

                if turn=="white" and logic.list2d[whsrow][whscol][1][0] != "\*":

                    return

                elif turn=="black" and logic.list2d[whsrow][whscol][1][0] == "\*":

                    return

            else:

                return

            move\_stage = 1

            #Getting the active piece [currently moving piece]

            if white == True:

                active\_piece = globals()[f'wh{wh\_list\_piece\_numbers[whsrow][whscol]}']

            elif black == True:

                active\_piece = globals()[f'bl{bl\_list\_piece\_numbers[blsrow][blscol]}']

            #Changing the colour of the starting square

            change\_square\_clr(whsrow, whscol, blsrow, blscol, constant.SELECTEDLIGHTSQUARECLR, constant.SELECTEDDARKSQUARECLR)

            #Displaying the legal moves

            legaladd = logic.gameprocessing((whsrow, whscol), (None, None))[8]

            if legalmoves==True:

                show\_legal\_squares(True)

                home.wn.tracer(1)

            #Logging some values

            logmessage.log()

            logmessage.log("Selected Address (ChessNaming) (list2d Numbering): ", info(whsrow, whscol)," ", (whsrow, whscol))

            logmessage.log("Possible Legal Squares: ", legaladd)

            home.canvas\_turtle.unbind("<ButtonPress>")

            home.canvas\_turtle.bind("<B1-Motion>", lambda e: get\_move(e.x, e.y, "motion"))

            home.canvas\_turtle.bind("<ButtonRelease-1>", lambda e: get\_move(e.x, e.y, "release"))

        elif move\_stage == 1 and mode == "motion":

            #print("Motion phase")

            active\_piece.goto(x,y)

        elif move\_stage == 1 and mode == "release":

            erow,ecol=get(x,y)

            wherow, whecol = whiteview\_blackview\_equivalents(erow,ecol)[0:2]

            def return\_piece():

                home.wn.tracer(0)

                if white:

                    sx, sy = antiget(whsrow, whscol)

                elif black:

                    sx, sy = antiget(blsrow, blscol)

                active\_piece.goto(sx, sy)

                home.wn.tracer(1)

            #print("All event bindings removed temporarily")

            home.canvas\_turtle.unbind("<ButtonPress>")

            home.canvas\_turtle.unbind("<B1-Motion>")

            home.canvas\_turtle.unbind("<ButtonRelease-1>")

            success, pawnv = process\_move((whsrow, whscol), (wherow, whecol))

            if success:

                timer\_start = True

                old = time.time()

                movestr = f'{whsrow}{whscol}{wherow}{whecol}'

                if home.mode\_of\_play == 'online' and not pawnv:

                    payload = {

                        'code':home.code,

                        'side':home.player\_side,

                        'move':movestr,

                        'pppiece':''

                    }

                    requests.post(home.geturl('rooms'),data=payload)

                move\_stage = 0

            elif not success:

                home.root\_turtle.after(10, return\_piece)

                #print("Returning the piece to its original position")

            #print("ButtonPress has got back control")

            home.canvas\_turtle.bind("<ButtonPress>", lambda e: get\_move(e.x, e.y, "press"))

        elif move\_stage == 1 and mode == "press":

            #print("Second Press")

            row,col = get(x,y)

            if not(0<=row<=7 and 0<=col<=7):

                return

            press\_x, press\_y = x, y

            whrow, whcol = whiteview\_blackview\_equivalents(row,col)[0:2]

            def on\_point\_release(x,y):

                global move\_stage, legaladd

                x,y = get\_turtle\_coord(x,y)

                if (x,y) == (press\_x, press\_y) and (touch\_move==False or (touch\_move==True and legaladd == [])):

                    reset\_move()

                    move\_stage = 0

                    home.canvas\_turtle.unbind("<B1-Motion>")

                    home.canvas\_turtle.unbind("<ButtonRelease-1>")

                    home.canvas\_turtle.bind("<ButtonPress>", lambda e: get\_move(e.x, e.y, "press"))

            if (whrow, whcol) == (whsrow, whscol):

                home.canvas\_turtle.bind("<B1-Motion>", lambda e: get\_move(e.x, e.y, "motion"))

                home.canvas\_turtle.bind("<ButtonRelease-1>", lambda e: on\_point\_release(e.x, e.y))

            else:

                if logic.sameteam(logic.list2d[whrow][whcol], logic.list2d[whsrow][whscol]) and (touch\_move==False or (touch\_move==True and legaladd == [])):

                    reset\_move()

                    x,y = get\_tkinter\_coord(x,y)

                    move\_stage = 0

                    get\_move(x, y, 'press')

                    return

            home.canvas\_turtle.bind("<ButtonRelease-1>", lambda e: get\_move(e.x, e.y, "release"), '+')

    #Returns: success, pawnv

    def process\_move(add1, add2, pppiece=''):

        global legaladd, move\_stage, prev\_move\_add, wherow, whecol, blerow, blecol, movestr, checkv, castlev, movestr, whsrow, whscol, blsrow, blscol

        #scord, ecord - always white view

        sr,sc = add1

        er,ec = add2

        whsrow,whscol,blsrow,blscol = sr,sc,7-sr,7-sc

        wherow,whecol,blerow,blecol = er,ec,7-er,7-ec

        success = False

        if 0<=wherow<=7 and 0<=whecol<=7:

            success, fadd, ladd, checkv, checkmatev, pawnv, castlev, legalv, legaltuple, enpassant, enpassant\_killpawn\_add, stalematev=logic.gameprocessing((whsrow, whscol),(wherow, whecol))

        if not success:

            return False, False

        movestr = f'{whsrow}{whscol}{wherow}{whecol}'

        #Hiding the legal moves after the user clicks on the ending square

        if  0<=wherow<=7 and 0<=whecol<=7 and (whsrow, whscol) != (wherow, whecol) and success==True:

            if legalmoves==True:

                show\_legal\_squares(False)

                legaladd = []

                home.wn.tracer(1)

        #Reverting the colour of the squares of the previous move

        try:

            pwhsrow, pwhscol, pwherow, pwhecol = prev\_move\_add

            for row,col in ((pwhsrow, pwhscol), (pwherow, pwhecol)):

                change\_square\_clr(row, col, 7-row, 7-col, light\_square\_clr, dark\_square\_clr)

        except:

            pass

        #Changing the colour of the square clicked(Only ending square)

        if  0<=wherow<=7 and 0<=whecol<=7 and logic.list2d[wherow][whecol] != logic.emp :

            change\_square\_clr(wherow, whecol, blerow, blecol, constant.SELECTEDLIGHTSQUARECLR, constant.SELECTEDDARKSQUARECLR)

        prev\_move\_add = (whsrow, whscol, wherow, whecol)

        if not castlev:

            if home.volume\_toggle == True:

                sounds(wherow, whecol, enpassant, False, checkv, checkmatev, stalematev)

            make\_move(whsrow, whscol, blsrow, blscol, wherow, whecol, blerow, blecol, 20)

            hide\_dead\_pieces(enpassant, enpassant\_killpawn\_add, wherow, whecol, blerow, blecol)

            display\_check\_and\_checkmate(logic.list2d, (224, 111, 111), False)

            #Changes in white view list of piece numbers

            wh\_list\_piece\_numbers[wherow][whecol] , wh\_list\_piece\_numbers[whsrow][whscol] = wh\_list\_piece\_numbers[whsrow][whscol] , 0

            #Changes in black view list of piece numbers

            bl\_list\_piece\_numbers[blerow][blecol] , bl\_list\_piece\_numbers[blsrow][blscol] = bl\_list\_piece\_numbers[blsrow][blscol] , 0

            if enpassant==True:

                #Changes in white view list of piece numbers

                wh\_list\_piece\_numbers[enpassant\_killpawn\_add[0]][enpassant\_killpawn\_add[1]] = 0

                #Changes in black view list of piece numbers

                bl\_list\_piece\_numbers[7-enpassant\_killpawn\_add[0]][7-enpassant\_killpawn\_add[1]] = 0

            #if home.volume\_toggle == True:

            #    sounds(None, None, False, False, checkv, checkmatev, stalematev)

            if pppiece and pawnv: #Using the choice of promoted piece from the online opponent

                logic.pawnpromotion((wherow, whecol), pppiece)

                home.wn.tracer(0)

                if white==True:

                    globals()[f'wh{wh\_list\_piece\_numbers[wherow][whecol]}'].shape(pppiece.title())

                if black==True:

                    globals()[f'bl{bl\_list\_piece\_numbers[blerow][blecol]}'].shape(pppiece.title())

                display\_check\_and\_checkmate(logic.list2d, (224, 111, 111), False)

                display\_stalemate(logic.list2d)

                toggle\_turns(False, pppiece)

                #print('toggled turns')

                home.wn.tracer(1)

            elif pawnv and not pppiece:

                promote\_pawn(wherow, whecol, blerow, blecol)

                return success , True

            toggle\_turns(pawnv)

            #print('toggled turns')

        elif castlev:

            if home.volume\_toggle == True:

                sounds(None, None, False, True, checkv, checkmatev, stalematev)

            #King Movement

            whksrow, whkscol=fadd[0][0], fadd[0][1]

            whkerow, whkecol=fadd[1][0], fadd[1][1]

            blksrow, blkscol=(7-fadd[0][0]), (7-fadd[0][1])

            blkerow, blkecol=(7-fadd[1][0]), (7-fadd[1][1])

            #Rook Movement

            whrsrow, whrscol=ladd[0][0], ladd[0][1]

            whrerow, whrecol=ladd[1][0], ladd[1][1]

            blrsrow, blrscol=(7-ladd[0][0]), (7-ladd[0][1])

            blrerow, blrecol=(7-ladd[1][0]), (7-ladd[1][1])

            #Moving the king

            make\_move(whksrow, whkscol, blksrow, blkscol, whkerow, whkecol, blkerow, blkecol, 50)

            #Moving the rook

            make\_move(whrsrow, whrscol, blrsrow, blrscol, whrerow, whrecol, blrerow, blrecol, 50)

            display\_check\_and\_checkmate(logic.list2d, (224, 111, 111), False)

            #Changes in white view list of piece numbers

            wh\_list\_piece\_numbers[whkerow][whkecol] , wh\_list\_piece\_numbers[whksrow][whkscol] = wh\_list\_piece\_numbers[whksrow][whkscol] , 0

            wh\_list\_piece\_numbers[whrerow][whrecol] , wh\_list\_piece\_numbers[whrsrow][whrscol] = wh\_list\_piece\_numbers[whrsrow][whrscol] , 0

            #Changes in black view list of piece numbers

            bl\_list\_piece\_numbers[blkerow][blkecol] , bl\_list\_piece\_numbers[blksrow][blkscol] = bl\_list\_piece\_numbers[blksrow][blkscol] , 0

            bl\_list\_piece\_numbers[blrerow][blrecol] , bl\_list\_piece\_numbers[blrsrow][blrscol] = bl\_list\_piece\_numbers[blrsrow][blrscol] , 0

            toggle\_turns(False)

            #print('toggled turns')

        try:

            pwhsrow, pwhscol, pwherow, pwhecol = prev\_move\_add

            for row,col in ((pwhsrow, pwhscol), (pwherow, pwhecol)):

                change\_square\_clr(row, col, 7-row, 7-col, constant.SELECTEDLIGHTSQUARECLR, constant.SELECTEDDARKSQUARECLR)

        except:

            pass

        display\_check\_and\_checkmate(logic.list2d, (224, 111, 111), True)

        display\_stalemate(logic.list2d)

        return success, False

    global timer\_start, whitetime , blacktime, firstmove, oldtime, ready\_to\_play, game\_date, game\_start\_time

    home.root\_turtle.protocol("WM\_DELETE\_WINDOW", on\_turtle\_close)

    logic.define\_basic\_global\_variables()

    define\_basic\_global\_variables(touch\_move\_)

    configure\_game()

    home.wn.tracer(1)

    home.wn.listen()

    if home.mode\_of\_play == "offline":

        ready\_to\_play = True

        game\_date = datetime.date.today()

        game\_start\_time = datetime.datetime.now().strftime("%H:%M:%S")

    else:

        ready\_to\_play = False

    #Add game\_date line for online after getting from Subham

    home.canvas\_turtle.bind("<ButtonPress>", lambda e: get\_move(e.x, e.y, "press"))

    #Timer Variables

    home.min\_time = int(home.min\_time) if home.min\_time.isdigit() else float('inf')

    home.increment = int(home.increment) if home.increment else 0

    timer\_start = False

    whitetime = home.min\_time\*60

    blacktime = home.min\_time\*60

    game\_start\_time = datetime.datetime.now().strftime("%H:%M:%S")

    game\_date = datetime.date.today()

    home.canvas\_turtle.bind("<ButtonPress>", lambda e: get\_move(e.x, e.y, "press"))

    #Thread(target=gettime).start()

    if home.mode\_of\_play == "offline":

        main\_updater\_offline()

    elif home.mode\_of\_play == 'online':

        Thread(target=connection).start()

        main\_updater\_online()

    home.wn.tracer(1)

replay.py

#Import of Modules

import datetime

import turtle

from tkinter import \*

from tkinter import ttk

from tkinter import scrolledtext

from tkinter import messagebox

from functools import partial

from playsound import playsound

from tkinter import colorchooser

from utils import Tooltip, lighten\_color, rgb\_to\_hex, ScrolledFrame, get\_chess\_notation, tfor

#Import of created files

import home

import constant

import database\_functions

import logic

def save\_configuration(replay\_match\_number, config\_no):

    global active\_save\_configuration\_number, root\_saved\_configurations, root\_save\_configuration

    try:

        if config\_no == active\_save\_configuration\_number:

            return None

        else:

            root\_save\_configuration.destroy()

    except:

        pass

    active\_save\_configuration\_number = config\_no

    root\_save\_configuration=Tk()

    root\_save\_configuration.title("Save Current Configuration")

    #root\_save\_configuration.iconphoto(False, home.img\_pocket\_chess\_arena\_icon)

    size=(450, 500)

    xoffset = int(3/5.6 \* home.root\_turtle.winfo\_screenwidth())

    yoffset = int(1/7 \* home.root\_turtle.winfo\_screenheight())

    root\_save\_configuration.geometry(f"{size[0]}x{size[1]}+{xoffset}+{yoffset}")

    root\_save\_configuration.resizable(width=False, height=False)

    def on\_closing():

        global active\_save\_configuration\_number

        active\_save\_configuration\_number = None

        root\_save\_configuration.destroy()

    root\_save\_configuration.protocol("WM\_DELETE\_WINDOW", on\_closing)

    top\_frame = Frame(root\_save\_configuration)

    top\_frame.pack(side=TOP, fill='x')

    top\_frame.columnconfigure((0,1,2,3,4), weight=1)

    title\_text=StringVar(top\_frame)

    title\_text.set("TITLE")

    title\_entry = Entry(top\_frame, font=("Comic Sans", 18, 'bold'), textvariable = title\_text, relief="groove", bd=3)

    title\_entry.grid(row=0, column=1, columnspan=3, sticky="WE")

    def onSaveButtonClick():

        global active\_save\_configuration\_number

        #Title Validation

        if len(title\_entry.get().rstrip()) > 150:

            messagebox.showerror('Error', 'Max Chars Allowed:\n150')

            return None

        for i in range(len(title\_entry.get().rstrip())):

            if title\_entry.get().rstrip()[i] in ("\"", ",", "(", ")"):

                messagebox.showerror('Error', 'Invalid Chars-\nRound Brackets ( )\nComma ,\nDouble Quotes "')

                return None

        #Notes Validation

        if len(notes\_entry.get(1.0, "end-1c").rstrip()) > 1500:

            messagebox.showerror('Error', 'Max Chars Allowed:\n1500')

            return None

        for i in range(len(notes\_entry.get(1.0, "end-1c").rstrip())):

            if notes\_entry.get(1.0, "end-1c").rstrip()[i] in ("\"", ",", "(", ")"):

                messagebox.showerror('Error', 'Round Brackets\nComma\nDouble Quotes')

                return None

        if database\_functions.check\_connection()==False:

            database\_functions.open\_connection()

        if database\_functions.check\_connection()==True:

            database\_functions.update\_configuration\_saved(replay\_match\_number, config\_no, title\_entry.get().rstrip(), notes\_entry.get(1.0, "end-1c").rstrip())

        else:

            messagebox.showerror("Error", "Check your internet connection and try again.")

            return None

        active\_save\_configuration\_number = None

        root\_save\_configuration.destroy()

        try:

            root\_saved\_configurations.destroy()

            show\_saved\_configurations(replay\_match\_number)

        except:

            pass

    ttk.Button(top\_frame, text="SAVE",command=onSaveButtonClick).grid(row=0, column=4)

    main\_frame = Frame(root\_save\_configuration, relief="ridge", bd=3)

    main\_frame.pack(side=TOP, fill='both', expand=True)

    scrollbar = Scrollbar(main\_frame, orient ="vertical")

    scrollbar.pack(side="right", fill="y")

    notes\_entry = Text(main\_frame, font=("Consolas", 15, 'normal'), wrap = 'word', height = main\_frame.winfo\_height(), yscrollcommand=scrollbar.set)

    notes\_entry.pack(fill="both", expand=True)

    saved\_configuration\_numbers=()

    for i in database\_functions.receive\_configurations\_saved(replay\_match\_number):

        saved\_configuration\_numbers+=(i[1],)

    if config\_no in saved\_configuration\_numbers:

        for i in database\_functions.receive\_configurations\_saved(replay\_match\_number):

            if i[1]==config\_no:

                title\_text.set(i[2])

                title\_entry.configure(textvariable=title\_text)

                notes\_entry.insert(INSERT, i[3])

    else:

        notes\_entry.insert(INSERT, "NOTES")

    scrollbar.config(command = notes\_entry.yview)

def show\_saved\_configurations(replay\_match\_number):

    global root\_saved\_configurations

    try:

        root\_saved\_configurations.destroy()

    except:

        pass

    root\_saved\_configurations = Tk()

    root\_saved\_configurations.title("Saved Configurations")

    #root\_saved\_configurations.iconphoto(False, home.img\_pocket\_chess\_arena\_icon)

    size=(500, 600)

    xoffset = int(3/5.6 \* home.root\_turtle.winfo\_screenwidth())

    yoffset = int(1/15 \* home.root\_turtle.winfo\_screenheight())

    root\_saved\_configurations.geometry(f"{size[0]}x{size[1]}+{xoffset}+{yoffset}")

    root\_saved\_configurations.resizable(width=False, height=True)

    configs=()

    titles=()

    notes=()

    for i in database\_functions.receive\_configurations\_saved(replay\_match\_number):

        configs+=(i[1],)

        titles+=(i[2],)

        notes+=(i[3],)

    #Creating main\_frame in which all the contents of saved\_configuration will be present

    main\_frame = ScrolledFrame(root\_saved\_configurations, max\_height = size[1])

    main\_frame.pack(fill = 'both', expand = True)

    for i in range(len(titles)):

        frame=Frame(main\_frame.viewPort, height=300, relief="raised", bd=5)

        frame.pack(fill='x')

        title\_text=scrolledtext.ScrolledText(frame, font=('Consolas', 16, 'bold'), relief='sunken', bd=3, height=0.5, wrap = 'word')

        title\_text.pack(fill=X)

        notes\_text=scrolledtext.ScrolledText(frame, font=('Consolas', 13,'normal'), height=7, wrap = 'word')

        notes\_text.pack(fill=X)

        notes\_text.insert(INSERT, notes[i].rstrip())

        title\_text.insert(INSERT, titles[i].rstrip())

        title\_text.configure(state='disabled')

        notes\_text.configure(state='disabled')

        #Button\_no are separately numbered for the show, edit and delete buttons. Button\_no starts from 0 just like i.

        def onShowButtonClick(button\_no):

            configure\_pieces(configs[button\_no])

            home.wn.tracer(1)

            root\_saved\_configurations.lift()

            root\_saved\_configurations.attributes("-topmost", True)

        def onEditButtonClick(button\_no):

            save\_configuration(replay\_match\_number, configs[button\_no])

            root\_saved\_configurations.attributes("-topmost", False)

        def onDeleteButtonClick(button\_no):

            database\_functions.delete\_configuration(replay\_match\_number, configs[button\_no])

            root\_saved\_configurations.destroy()

            show\_saved\_configurations(replay\_match\_number)

        button\_frame=Frame(frame)

        show\_button = ttk.Button(button\_frame, text='SHOW', command = partial(onShowButtonClick, i))

        edit\_button = ttk.Button(button\_frame, text='EDIT', command = partial(onEditButtonClick, i))

        delete\_button = ttk.Button(button\_frame, text='DELETE', command = partial(onDeleteButtonClick, i))

        show\_button.grid(row=0, column=0, sticky='WE')

        edit\_button.grid(row=0, column=1, sticky='WE')

        delete\_button.grid(row=0, column=2, sticky='WE')

        button\_frame.pack()

        Label(main\_frame, text="      ").pack(side=TOP, fill='x')

def list\_of\_games\_played\_page():

    global main\_frame, label\_title, frame\_top

    def onButtonClick():

        global error\_alert, main\_frame, label\_title, frame\_top

        try:

            desired\_match\_number = int(match\_number\_replay.get())

        except:

            desired\_match\_number = match\_number\_replay.get()

        if desired\_match\_number in possible\_match\_numbers:

            global root\_internet\_problem

            try:

                error\_alert.grid\_forget()

            except:

                pass

            try:

                root\_internet\_problem.destroy()

            except:

                pass

            if database\_functions.check\_connection()==False:

                database\_functions.open\_connection()

            if database\_functions.check\_connection()==True:

                val = match\_number\_replay.get()

                label\_title.destroy()

                frame\_top.destroy()

                main\_frame.destroy()

                home.root\_turtle.grid\_columnconfigure((0,1,2,3), weight = 0)

                home.root\_turtle.grid\_rowconfigure((0,1,2,3), weight = 0)

                setup\_replay()

                replay\_main(val)

            else:

                root\_internet\_problem = Tk()

                root\_internet\_problem.title("Error")

                size = (250, 70)

                xoffset = int(home.root\_turtle.winfo\_width()/2 - size[0]/2)

                yoffset = int(home.root\_turtle.winfo\_height()/2 - size[1]/2)

                root\_internet\_problem.geometry(f"{size[0]}x{size[1]}+{xoffset}+{yoffset}")

                root\_internet\_problem.resizable(width=False, height=False)

                root\_internet\_problem.attributes('-topmost', True)

                ttk.Label(root\_internet\_problem, text="Check your internet connection and try again.", width = size[0]).pack(fill='x', side='top')

                def onOkayButtonClick():

                    root\_internet\_problem.destroy()

                ttk.Button(root\_internet\_problem, text='OK', command=onOkayButtonClick).pack(side='bottom')

        else:

            error\_alert = Label(entry\_frame, text="Invalid Match Number", fg="red")

            error\_alert.grid(row=0, column=3)

    def on\_back():

        global main\_frame, label\_title

        label\_title.destroy()

        main\_frame.destroy()

        home.root\_turtle.grid\_columnconfigure((0,1,2,3), weight = 0)

        home.root\_turtle.grid\_rowconfigure((0,1,2,3), weight = 0)

        home.main\_menu()

    def on\_turtle\_close():

        home.program\_running = False

        home.root\_turtle.destroy()

        database\_functions.close\_connection()

    home.root\_turtle.protocol("WM\_DELETE\_WINDOW", on\_turtle\_close)

    #Configuring root\_turtle

    home.root\_turtle.configure(bg = constant.DARKBGCLR)

    #Gridding root\_turtle

    home.root\_turtle.grid\_columnconfigure(0, weight = 1)

    home.root\_turtle.grid\_rowconfigure(1, weight = 1)

    root\_turtle\_width = 1250

    root\_turtle\_height = 500

    # Get the screen dimension

    screen\_width = home.root\_turtle.winfo\_screenwidth()

    screen\_height = home.root\_turtle.winfo\_screenheight()

    # find the center point

    center\_x = int(screen\_width/2 - root\_turtle\_width / 2)

    center\_y = 20

    # set the position of the window to the center of the screen

    home.root\_turtle.geometry(f'{root\_turtle\_width}x{root\_turtle\_height}+{center\_x}+{center\_y}')

    home.wn.clearscreen()

    home.wn.bgcolor("#C2CBD1")

    frame\_top = Frame(home.root\_turtle, bg = constant.DARKBGCLR, bd = 0, highlightthickness = 0)

    frame\_top.grid(row = 0, column = 0, sticky = NSEW)

    frame\_top.grid\_columnconfigure(1, weight = 1)

    frame\_top.grid\_rowconfigure(0, weight = 1)

    label\_title = Label(frame\_top, text="GAMES PLAYED", font = ("Algerian", 25, 'bold', 'underline'), bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR)

    label\_title.grid(row = 0, column = 1, sticky = NSEW, padx = 10,)

    #Adding the back button

    img\_back = PhotoImage(file = "./Icons/back.png").subsample(2,2)

    button\_back = Button(frame\_top, image = img\_back, bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, activebackground = constant.DARKBGCLR, activeforeground = constant.DARKBGTEXTCLR, font = ('Consolas', 20, 'bold'), command = on\_back, bd = 0, highlightthickness = 0)

    button\_back.grid(row = 0, column = 0, sticky = W, padx = 10)

    #Creating Main Frame

    main\_frame = ScrolledFrame(home.root\_turtle, bg = constant.DARKBGCLR, max\_height = root\_turtle\_height)

    main\_frame.grid(row = 1, column = 0, sticky = NSEW)

    #Creating a frame to contain the entry box for accepting the match number

    entry\_frame = Frame(main\_frame.viewPort, bg = constant.DARKBGCLR)

    entry\_frame.grid(row = 0, column = 0, columnspan = 10, sticky = NSEW, pady = 2)

    #Gridding entry\_frame

    entry\_frame.grid\_columnconfigure((0,4), weight = 1)

    Label(entry\_frame, text = "MATCH NUMBER: ", font=('Consolas', 15, 'bold'), bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR).grid(row=0, column=0, sticky = E , padx=5 , ipadx=5 , ipady=5 )

    match\_number\_replay=Entry(entry\_frame, font = ('Comic Sans', 16))

    match\_number\_replay.grid(row=0, column=1 , padx=5, pady = 2, ipadx=5 , ipady=2)

    possible\_match\_numbers=()

    for i in database\_functions.receive\_all\_game\_details():

        possible\_match\_numbers+=(i[0],)

    Button(entry\_frame, text="SHOW REPLAY", font = ('Comic Sans', 16), bg = constant.SECONDARYCLR, fg = constant.DARKBGTEXTCLR, command=onButtonClick).grid(row=0, column=2, padx=5 , ipadx=5 , ipady=2)

    #Creating empty space to the left of the canvas

    Label(main\_frame, text="               ", bg = constant.DARKBGCLR).pack(side='left')

    #Creating empty to the right of the canvas

    Label(main\_frame, text="          ", bg = constant.DARKBGCLR).pack(side='right')

    #Creating frame

    main\_frame.viewPort.columnconfigure((0,1,2,3,4,5,6,7,8,9), weight=1)

    #Removing bd and highlightthickness

    main\_frame.configure(bd = 0, highlightthickness = 0)

    main\_frame.viewPort.configure(bd = 0, highlightthickness = 0)

    headings = ("Match No.", "Date", "Start Time", "End Time", "Duration", "White Player", "Black Player", "Result")

    for i in range(len(headings)):

        Label(main\_frame.viewPort, text=headings[i], font = ("Consolas", 18, 'bold'), borderwidth = 3, relief="ridge").grid(row=1, column=i+1, sticky ="WE")

    rownum=2

    for individual\_match\_details in database\_functions.receive\_all\_game\_details():

        colnum=1

        for i in individual\_match\_details[:-2]:

            Label(main\_frame.viewPort, text=i, font = ("Consolas", 16, 'normal'),wraplength=180, justify="center", borderwidth = 1, relief="sunken").grid(row=rownum, column=colnum, sticky ="NSWE")

            colnum+=1

        rownum+=1

    '''

    def onCanvasConfigure(event):

        global frame\_in\_canvas

        try:

            canvas.delete(frame\_in\_canvas)

        except:

            pass

        canvas.update()

        frame.update()

        nwposition = ((canvas.winfo\_width() - frame.winfo\_width())//2,0)

        if frame.winfo\_width() <= canvas.winfo\_width():

            frame\_width = canvas.winfo\_width()

        else:

            frame\_width = frame.winfo\_width()

        frame\_in\_canvas = canvas.create\_window(nwposition, window=frame, anchor='n', width = frame\_width)

        canvas.configure(scrollregion=canvas.bbox("all"),yscrollcommand=scrollbar\_vertical.set, xscrollcommand=scrollbar\_horizontal.set)

    canvas.bind("<Configure>", onCanvasConfigure)

    '''

    home.root\_turtle.mainloop()

def define\_basic\_global\_variables(match\_no, list\_of\_moves, list\_of\_times, min\_time\_, increment\_):

    global size, sqsize, stretch\_square, drift, verticaldrift, boardview, replay\_list2d, list\_piece\_numbers, piece\_count, move\_count, replay\_writeturtle, replay\_match\_number, replay\_writeturtle, moves, times, emp, light\_square\_clr, dark\_square\_clr, min\_time, increment, times\_white, times\_black, last\_sec

    #Initialising a chess for the chess board and a corresponding size for root\_turtle

    length = 600

    size\_canvas\_turtle = int(length) + 15

    size\_root\_turtle = size\_canvas\_turtle + 35

    home.canvas\_turtle.configure(width = size\_canvas\_turtle, height = size\_canvas\_turtle)

    home.canvas\_turtle.update()

    home.root\_turtle.geometry(f"{size\_root\_turtle}x{size\_root\_turtle+95}")

    #Size of the chess board

    size = length

    #Size of 1 UNIT square in the chess board

    sqsize = size/8

    #The value by which each unit square should be stretched.

    stretch\_square = (size/8)/20

    #Initialising drifts

    drift = 0

    verticaldrift = 0

    #Board View (Default is white)

    boardview="white"

    #Initialising the colours of the squares

    light\_square\_clr = constant.LIGHTSQUARECLR

    dark\_square\_clr = constant.DARKSQUARECLR

    #The replay for which match number is to be shown

    replay\_match\_number = match\_no

    #Initialisation of some more variables

    replay\_list2d=logic.copy\_of\_init\_list2d()

    list\_piece\_numbers = [[0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0]]

    piece\_count = 0

    move\_count = 0

    #Writer turtle of the replay page

    replay\_writeturtle = turtle.Turtle()

    replay\_writeturtle.ht()

    replay\_writeturtle.pu()

    #Initialising the list of the moves made and the times taken

    moves = list\_of\_moves

    times = list\_of\_times

    times = [x for x in times if x.total\_seconds() >= 0]

    #Initialsing min\_time and increment

    min\_time = min\_time\_

    increment = increment\_

    #Initialising times\_white and times\_black

    times\_white = times[0::2]

    times\_black = times[1::2]

    #Defining emp which will be used as an element in replay\_list2d

    emp=["", "", 0, 0]

    #last seconds

    last\_sec = 20

def setup\_replay():

    global  canvas\_turtle, var\_board\_scaler, scale\_board, pinned, img\_pin\_outline, img\_pin\_filled, button\_pin, img\_color\_chooser, img\_flip, volume\_toggle, button\_volume\_toggle, img\_volume\_on, img\_volume\_off, button\_extend\_right, img\_save\_configuration, img\_saved\_configurations, img\_back, frame\_top, frame\_bottom, frame\_right, img\_left\_arrow, img\_right\_arrow, label\_white\_timer\_bottom, label\_white\_timer\_top, label\_black\_timer\_bottom, label\_black\_timer\_top, frame\_moves, frame\_arrows

    def on\_scale\_board\_release(e):

        global scale\_board, var\_board\_scaler, size, sqsize, stretch\_square, button\_extend\_right

        size = 600 \* var\_board\_scaler.get() / 50

        sqsize = size / 8

        stretch\_square=(size/8)/20

        configure\_game()

        size\_canvas\_turtle = int(size) + 15

        size\_root\_turtle = size\_canvas\_turtle + 35

        home.canvas\_turtle.configure(width = size\_canvas\_turtle, height = size\_canvas\_turtle)

        home.canvas\_turtle.update()

        width\_root\_turtle = size\_root\_turtle

        if button\_extend\_right['text'] == "<":

            frame\_right.update\_idletasks()

            width\_root\_turtle += frame\_right.winfo\_width()

        home.root\_turtle.geometry(f"{width\_root\_turtle}x{size\_root\_turtle+95}")

    def on\_pin\_toggle():

        global pinned, button\_pin

        if pinned == True:

            pinned = False

            home.root\_turtle.attributes('-topmost',False)

            button\_pin.configure(image = img\_pin\_outline)

        elif pinned == False:

            pinned = True

            home.root\_turtle.attributes('-topmost',True)

            button\_pin.configure(image = img\_pin\_filled)

    def on\_extend\_right():

        global button\_extend\_right, frame\_right

        w = home.root\_turtle.winfo\_width()

        h = home.root\_turtle.winfo\_height()

        frame\_right.update\_idletasks()

        if button\_extend\_right['text'] == ">":

            frame\_right.grid(row = 0, column = 2, rowspan = 3, sticky = NSEW)

            new\_w = w + frame\_right.winfo\_width()

            button\_extend\_right.configure(text = "<")

        elif button\_extend\_right['text'] == "<":

            new\_w = w - frame\_right.winfo\_width()

            frame\_right.grid\_forget()

            button\_extend\_right.configure(text = ">")

        home.root\_turtle.geometry(f"{new\_w}x{h}")

    def on\_color\_chooser():

        global dark\_square\_clr, light\_square\_clr

        try:

            clr\_dark = colorchooser.askcolor(title ="Choose DARK square color", color = dark\_square\_clr)[0]

            if clr\_dark is None:

                return

        except:

            return

        dark\_square\_clr = rgb\_to\_hex(clr\_dark)

        configure\_game()

    def on\_volume\_toggle():

        global volume\_toggle, button\_volume\_toggle

        if volume\_toggle == True:

            volume\_toggle = False

            button\_volume\_toggle.configure(image = img\_volume\_off)

        elif volume\_toggle == False:

            volume\_toggle = True

            button\_volume\_toggle.configure(image = img\_volume\_on)

    def on\_flip():

        global boardview, label\_white\_timer\_bottom, label\_white\_timer\_top, label\_black\_timer\_bottom, label\_black\_timer\_top

        if boardview == "white":

            boardview = "black"

            if move\_count != 0:

                label\_white\_timer\_bottom.grid\_forget()

                label\_white\_timer\_top.grid(row = 0, column = 3, sticky = EW, padx = 5, ipady = 3)

                label\_black\_timer\_top.grid\_forget()

                label\_black\_timer\_bottom.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 3)

        elif boardview == "black":

            boardview = "white"

            if move\_count != 0:

                label\_white\_timer\_top.grid\_forget()

                label\_white\_timer\_bottom.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 3)

                label\_black\_timer\_bottom.grid\_forget()

                label\_black\_timer\_top.grid(row = 0, column = 3, sticky = EW, padx = 5, ipady = 3)

        configure\_game()

    def on\_back():

        global frame\_top, frame\_bottom, frame\_right

        if database\_functions.check\_connection()==False:

            database\_functions.open\_connection()

        if database\_functions.check\_connection()==True:

            try:

                root\_saved\_configurations.destroy()

            except:

                pass

            home.wn.clearscreen()

            frame\_top.destroy()

            frame\_bottom.destroy()

            frame\_right.destroy()

            home.canvas\_turtle.grid(row = 0, column = 0)

            home.root\_turtle.minsize(1050, 550)

            home.root\_turtle.resizable(width = True, height = True)

            home.root\_turtle.grid\_columnconfigure((0,1,2,3), weight = 0)

            home.root\_turtle.grid\_rowconfigure((0,1,2,3), weight = 0)

            list\_of\_games\_played\_page()

        else:

            messagebox.showerror("Error", "Check your internet connection and try again.")

    def on\_show\_saved\_configurations(replay\_match\_number):

        if database\_functions.check\_connection()==False:

            database\_functions.open\_connection()

        if database\_functions.check\_connection()==True:

            show\_saved\_configurations(replay\_match\_number)

        else:

            messagebox.showerror("Error", "Check your internet connection and try again.")

    #Configuring home.root\_turtle

    home.root\_turtle.configure(bg = constant.LIGHTBGCLR)

    #Reconfiguring minsize of the window

    home.root\_turtle.minsize(0,0)

    home.root\_turtle.resizable(width = False, height = False)

    home.root\_turtle.overrideredirect(False)

    home.root\_turtle.withdraw()

    home.root\_turtle.deiconify()

    #Gridding home.root\_turtle

    home.root\_turtle.grid\_rowconfigure(0, weight = 0)

    home.root\_turtle.grid\_columnconfigure(0, weight = 0)

    home.root\_turtle.grid\_rowconfigure(1, weight = 1)

    home.root\_turtle.grid\_columnconfigure(1, weight = 1)

    #Placing canvas\_turtle inside canvas\_game

    home.canvas\_turtle.grid(row = 1, column = 1, sticky = NSEW)

    #Giving a background colour to wn

    home.wn.bgcolor("#000000")

    #Adding all the other elements present during the replay (except timers)

    #Creating all the frames for the 4 sides

    frame\_top = Frame(home.root\_turtle, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    frame\_bottom = Frame(home.root\_turtle, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    frame\_right = Frame(home.root\_turtle, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    #Adding all the frames to the screen

    frame\_top.grid(row = 0, column = 1, sticky = NSEW)

    frame\_bottom.grid(row = 2, column = 1, sticky = NSEW)

    #Gridding all the frames

    frame\_top.grid\_columnconfigure((3,4,5), weight = 1)

    frame\_bottom.grid\_columnconfigure((0,1,2), weight = 1)

    frame\_right.grid\_rowconfigure(0, weight = 1)

    frame\_right.grid\_columnconfigure(0, weight = 1)

    #Adding the left arrow and right arrow key prompt

    img\_left\_arrow = PhotoImage(file = "./Icons/left\_arrow.png")

    img\_right\_arrow = PhotoImage(file = "./Icons/right\_arrow.png")

    #Creating, gridding and adding frame\_arrows to frame\_top

    frame\_arrows = Frame(frame\_top, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    frame\_arrows.grid\_columnconfigure((0,1), weight = 1)

    frame\_arrows.grid\_rowconfigure(0, weight = 1)

    frame\_arrows.grid(row = 0, column = 3, columnspan = 2, sticky = NSEW, padx = 5)

    label\_left\_arrow = Label(frame\_arrows, image = img\_left\_arrow, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    label\_left\_arrow.grid(row = 0, column = 0, sticky = 'nse', padx = 5)

    label\_right\_arrow = Label(frame\_arrows, image = img\_right\_arrow, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    label\_right\_arrow.grid(row = 0, column = 1, sticky = 'nsw', padx = 5)

    button\_extend\_right = Button(frame\_top, text = ">", font = ('consolas', 20, 'bold'), bg = constant.LIGHTBGCLR, fg = constant.LIGHTBGTEXTCLR, activebackground = constant.LIGHTBGCLR, activeforeground = constant.LIGHTBGTEXTCLR, command = on\_extend\_right, bd = 0, highlightthickness = 0)

    button\_extend\_right.grid(row = 0, column = 6, sticky = NSEW, padx = (0,1))

    #Adding frame\_moves to frame\_right

    frame\_moves = ScrolledFrame(frame\_right, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0, max\_height = 1000)

    frame\_moves.grid(row = 0, column = 0, sticky = NSEW, padx = 2, pady = 2, ipadx = 2, ipady = 2)

    #Gridding frame\_moves.viewPort

    frame\_moves.viewPort.grid\_columnconfigure((1,2), weight = 1)

    #Adding the board scaler to frame\_top

    var\_board\_scaler = IntVar()

    scale\_board = ttk.Scale(frame\_top, variable = var\_board\_scaler, from\_ = 30, to = 70, orient = HORIZONTAL)

    var\_board\_scaler.set(50)

    scale\_board.bind("<ButtonRelease-1>", on\_scale\_board\_release)

    scale\_board.grid(row = 0, column = 5, sticky = EW, padx = 5)

    #Adding the pin to frame\_top

    img\_pin\_outline = PhotoImage(file = "./Icons/pin\_outline.png").subsample(15,15)

    img\_pin\_filled = PhotoImage(file = "./Icons/pin\_filled.png").subsample(15,15)

    #Adding a back button to frame\_Top

    img\_back = PhotoImage(file = "./Icons/back.png").subsample(3,3)

    button\_back = Button(frame\_top, image = img\_back, bg = constant.LIGHTBGCLR, fg = constant.LIGHTBGTEXTCLR, activebackground = constant.LIGHTBGCLR, activeforeground = constant.LIGHTBGTEXTCLR, font = ('Consolas', 20, 'bold'), command = on\_back, bd = 0, highlightthickness = 0)

    button\_back.grid(row = 0, column = 1, sticky = W, padx = 10)

    button\_pin = Button(frame\_top, image = img\_pin\_outline, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = on\_pin\_toggle, bd = 0, highlightthickness = 0)

    pinned = False

    button\_pin.grid(row = 0, column = 2, sticky = NSEW)

    #Creating frame\_icons to contain all the icons

    frame\_icons = Frame(frame\_bottom, bg = constant.LIGHTBGCLR, bd = 0, highlightthickness = 0)

    #Adding frame\_icons to the screen

    frame\_icons.grid(row = 0, column = 0, sticky = NSEW, padx = 5)

    #Gridding frame\_icons

    frame\_icons.grid\_columnconfigure((0,1,2,3,4), weight = 1)

    #Adding the color chooser icon to frame\_bottom

    img\_color\_chooser = PhotoImage(file = "./Icons/color\_chooser.png").subsample(3,3)

    button\_color\_chooser = Button(frame\_icons, image = img\_color\_chooser, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = on\_color\_chooser, bd = 0, highlightthickness = 0)

    #Adding volume toggle icon to frame\_bottom

    img\_volume\_on = PhotoImage(file = "./Icons/volume\_on.png").subsample(4,4)

    img\_volume\_off = PhotoImage(file = "./Icons/volume\_off.png").subsample(4,4)

    volume\_toggle = True

    button\_volume\_toggle = Button(frame\_icons, image = img\_volume\_on, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = on\_volume\_toggle, bd = 0, highlightthickness = 0)

    #Adding flip icon to frame\_bottom

    img\_flip = PhotoImage(file = "./Icons/flip.png").subsample(3,3)

    button\_flip = Button(frame\_icons, image = img\_flip, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = on\_flip, bd = 0, highlightthickness = 0)

    #Adding save configuration and saved configurations buttons

    img\_save\_configuration = PhotoImage(file = "./Icons/save.png").subsample(4,4)

    img\_saved\_configurations = PhotoImage(file = "./Icons/folder.png").subsample(4,4)

    button\_save\_configuration = Button(frame\_icons, image = img\_save\_configuration, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = lambda : save\_configuration(replay\_match\_number, move\_count), bd = 0, highlightthickness = 0)

    button\_saved\_configurations = Button(frame\_icons, image = img\_saved\_configurations, bg = constant.LIGHTBGCLR, activebackground = constant.LIGHTBGCLR, command = lambda : on\_show\_saved\_configurations(replay\_match\_number), bd = 0, highlightthickness = 0)

    #Adding all the icons to frame\_icons

    button\_color\_chooser.grid(row = 0, column = 0, sticky = NSEW)

    button\_volume\_toggle.grid(row = 0, column = 1, sticky = NSEW)

    button\_flip.grid(row = 0, column = 2, sticky = NSEW)

    button\_save\_configuration.grid(row = 0, column = 3, sticky = NSEW)

    button\_saved\_configurations.grid(row = 0, column = 4, sticky = NSEW)

    #Adding the timer labels to frame\_top and frame\_bottom

    label\_white\_timer\_top = Label(frame\_top, text = "00:00:00", bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('consolas', 15, 'bold'), bd = 2)

    label\_white\_timer\_bottom = Label(frame\_bottom, text = "00:00:00", bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('consolas', 15, 'bold'), bd = 2)

    label\_black\_timer\_top = Label(frame\_top, text = "00:00:00", bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('consolas', 15, 'bold'), bd = 2)

    label\_black\_timer\_bottom = Label(frame\_bottom, text = "00:00:00", bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('consolas', 15, 'bold'), bd = 2)

    #Adding tooltips for the required widgets

    Tooltip(button\_save\_configuration, text = "Save current configuration")

    Tooltip(button\_saved\_configurations, text = "View saved configurations")

    Tooltip(button\_flip, text = "Switch view")

    Tooltip(button\_volume\_toggle, text = "Toggle sound")

    Tooltip(button\_color\_chooser, text = "Change square colour")

    Tooltip(button\_pin, text = "Toggle pin window")

    Tooltip(scale\_board, text = "Scale board size")

    Tooltip(label\_left\_arrow, text = "Previous move")

    Tooltip(label\_right\_arrow, text = "Next move")

    for \_ in range(2):

        on\_extend\_right()

#Board setup

def board\_setup():

    home.wn.tracer(0)

    for y in range(8):

        for x in range(8):

            globals()[f't{x}{y}'] = turtle.Turtle()

            globals()[f't{x}{y}'].shape('square')

            globals()[f't{x}{y}'].shapesize(stretch\_square, stretch\_square)

            globals()[f't{x}{y}'].up()

            if (x+y)%2==0:

                globals()[f't{x}{y}'].color(light\_square\_clr)

            elif (x+y)%2==1:

                globals()[f't{x}{y}'].color(dark\_square\_clr)

            globals()[f't{x}{y}'].goto((sqsize/2) + sqsize\*(x-4)+drift , -(sqsize/2) + sqsize\*(4-y) + verticaldrift)

    display\_labels(boardview)

def update\_timers(move\_count):

    #moves\_wo\_pp = moves WithOut PawnPromotion

    moves\_wo\_pp = moves[:move\_count+1]

    for i in moves\_wo\_pp:

        if i[0] == "Pawnpromotion":

            moves\_wo\_pp.remove(i)

    no\_white\_moves = len(moves\_wo\_pp[0::2])

    no\_black\_moves = len(moves\_wo\_pp[1::2])

    if min\_time == 0:

        white\_time = (sum(times\_white[0:no\_white\_moves], datetime.timedelta(0)).total\_seconds())

        black\_time = (sum(times\_black[0:no\_black\_moves], datetime.timedelta(0)).total\_seconds())

    else:

        white\_time = ((datetime.timedelta(seconds=min\_time) - sum(times\_white[0:no\_white\_moves], datetime.timedelta(seconds=0.0)) + datetime.timedelta(seconds=increment \* (no\_white\_moves))).total\_seconds())

        black\_time = ((datetime.timedelta(seconds=min\_time) - sum(times\_black[0:no\_black\_moves], datetime.timedelta(seconds=0.0)) + datetime.timedelta(seconds=increment \* (no\_black\_moves))).total\_seconds())

        print(white\_time , black\_time)

    #Updating the timers on screen

    if white\_time < last\_sec:

        label\_white\_timer\_top.configure(text = tfor(white\_time))

        label\_white\_timer\_bottom.configure(text = tfor(white\_time))

    else:

        label\_white\_timer\_top.configure(text = tfor(white\_time)[:-2])

        label\_white\_timer\_bottom.configure(text = tfor(white\_time)[:-2])

    if black\_time < last\_sec:

        label\_black\_timer\_top.configure(text = tfor(black\_time))

        label\_black\_timer\_bottom.configure(text = tfor(black\_time))

    else:

        label\_black\_timer\_top.configure(text = tfor(black\_time)[:-2])

        label\_black\_timer\_bottom.configure(text = tfor(black\_time)[:-2])

#This function is called everytime the user presses the LEFT arrow key. Its function is to reverse the last move made on the screen and also update replay\_list2d.

def previous\_move():

    global move\_count, replay\_list2d, list\_piece\_numbers, piece\_count, moves

    if move\_count!=0:

        move\_count-=1

    else:

        return None

    home.wn.onkeypress(None,"Right")

    home.wn.onkeypress(None, "Left")

    #PLaying Sounds

    if volume\_toggle == True:

        playsound("./Sounds/move.mp3",False)

    if moves[move\_count][0] != "Pawnpromotion":

        update\_timers(move\_count-1)

    if moves[move\_count][0] not in ("Castling", "Enpassant", "Pawnpromotion"):

        #Changes in replay\_list2d

        startadd = moves[move\_count][1]

        endadd = moves[move\_count][0]

        replay\_list2d[endadd[0]][endadd[1]]=replay\_list2d[startadd[0]][startadd[1]]

        replay\_list2d[startadd[0]][startadd[1]]=moves[move\_count][2]

        replay\_list2d[endadd[0]][endadd[1]][2]=endadd[0]

        replay\_list2d[endadd[0]][endadd[1]][3]=endadd[1]

        globals()[f'piece{list\_piece\_numbers[startadd[0]][startadd[1]]}'].goto(coord\_from\_add(endadd[0], endadd[1]))

        list\_piece\_numbers[endadd[0]][endadd[1]] = list\_piece\_numbers[startadd[0]][startadd[1]]

        if moves[move\_count][2] != emp:

            create\_chess\_piece(startadd[0],startadd[1], moves[move\_count][2][0].title(), logic.colour(moves[move\_count][2]))

            home.wn.tracer(1)

            list\_piece\_numbers[startadd[0]][startadd[1]]=piece\_count

        elif moves[move\_count][2] == emp:

            list\_piece\_numbers[startadd[0]][startadd[1]]=0

    elif moves[move\_count][0] == "Castling":

        king\_startadd, king\_endadd = moves[move\_count][1][1], moves[move\_count][1][0]

        rook\_startadd, rook\_endadd = moves[move\_count][2][1], moves[move\_count][2][0]

        for startadd, endadd in ((king\_startadd, king\_endadd), (rook\_startadd, rook\_endadd)):

            #Changes in replay\_list2d

            replay\_list2d[endadd[0]][endadd[1]]=replay\_list2d[startadd[0]][startadd[1]]

            replay\_list2d[startadd[0]][startadd[1]]=["", "", 0, 0]

            replay\_list2d[endadd[0]][endadd[1]][2]=endadd[0]

            replay\_list2d[endadd[0]][endadd[1]][3]=endadd[1]

            #Making the piece move on the board

            globals()[f'piece{list\_piece\_numbers[startadd[0]][startadd[1]]}'].goto(coord\_from\_add(endadd[0], endadd[1]))

            #Modifying the list\_piecenumbers

            list\_piece\_numbers[endadd[0]][endadd[1]] = list\_piece\_numbers[startadd[0]][startadd[1]]

            list\_piece\_numbers[startadd[0]][startadd[1]]=0

    elif moves[move\_count][0] == "Enpassant":

        #Changes in replay\_list2d

        startadd, endadd = moves[move\_count][2], moves[move\_count][1]

        victimadd = moves[move\_count][3]

        replay\_list2d[endadd[0]][endadd[1]]=replay\_list2d[startadd[0]][startadd[1]]

        replay\_list2d[startadd[0]][startadd[1]]=["", "", 0, 0]

        replay\_list2d[endadd[0]][endadd[1]][2]=endadd[0]

        replay\_list2d[endadd[0]][endadd[1]][3]=endadd[1]

        replay\_list2d[victimadd[0]][victimadd[1]] = moves[move\_count][4]

        #Modifying the list\_piecenumbers of the victim piece

        piece\_count+=1

        list\_piece\_numbers[victimadd[0]][victimadd[1]] = piece\_count

        #Making the piece move on the board

        globals()[f'piece{list\_piece\_numbers[startadd[0]][startadd[1]]}'].goto(coord\_from\_add(endadd[0], endadd[1]))

        create\_chess\_piece(victimadd[0],victimadd[1], "Pawn", logic.colour(moves[move\_count][4]))

        home.wn.tracer(1)

        #Modifying the list\_piecenumbers of the stronger attacking piece

        list\_piece\_numbers[endadd[0]][endadd[1]] = list\_piece\_numbers[startadd[0]][startadd[1]]

        list\_piece\_numbers[startadd[0]][startadd[1]] = 0

    elif  moves[move\_count][0] == "Pawnpromotion":

        #Changes in replay\_list2d

        if logic.colour(moves[move\_count][2])=="white":

            color\_W\_or\_B = "\*"

        else:

            color\_W\_or\_B=""

        replay\_list2d[moves[move\_count][1][0]][moves[move\_count][1][1]] = ["pawn", f"{color\_W\_or\_B}^", moves[move\_count][2][2], moves[move\_count][2][3]]

        #Modifying the shape of the piece

        globals()[f'piece{list\_piece\_numbers[moves[move\_count][1][0]][moves[move\_count][1][1]]}'].shape("Pawn")

        previous\_move()

    home.wn.onkeypress(previous\_move, "Left")

    home.wn.onkeypress(next\_move, "Right")

#This function is called everytime the user presses the RIGHT arrow key. Its function is to display the move on the screen and also update replay\_list2d.

def next\_move():

    global move\_count, replay\_list2d, list\_piece\_numbers, times, boardview, frame\_arrows

    if move\_count == 0:

        frame\_arrows.destroy()

        if boardview == "black":

            label\_white\_timer\_top.grid(row = 0, column = 3, sticky = EW, padx = 5, ipady = 3)

            label\_black\_timer\_bottom.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 3)

        elif boardview == "white":

            label\_white\_timer\_bottom.grid(row = 0, column = 2, sticky = EW, padx = 5, ipady = 3)

            label\_black\_timer\_top.grid(row = 0, column = 3, sticky = EW, padx = 5, ipady = 3)

    if move\_count == len(moves):

        return None

    home.wn.onkeypress(None,"Right")

    home.wn.onkeypress(None, "Left")

    if moves[move\_count][0] != "Pawnpromotion":

        update\_timers(move\_count)

    if moves[move\_count][0] not in ("Castling", "Enpassant", "Pawnpromotion"):

        #Changes in replay\_list2d

        startadd = moves[move\_count][0]

        endadd = moves[move\_count][1]

        replay\_list2d[endadd[0]][endadd[1]]=replay\_list2d[startadd[0]][startadd[1]]

        replay\_list2d[startadd[0]][startadd[1]]=["", "", 0, 0]

        replay\_list2d[endadd[0]][endadd[1]][2]=endadd[0]

        replay\_list2d[endadd[0]][endadd[1]][3]=endadd[1]

        #Playing Sounds

        if volume\_toggle == True:

            if moves[move\_count][2] != emp:

                playsound("./Sounds/capture.mp3",False)

            elif moves[move\_count][2] == emp:

                playsound("./Sounds/move.mp3",False)

        #Making the piece move on the board

        globals()[f'piece{list\_piece\_numbers[startadd[0]][startadd[1]]}'].goto(coord\_from\_add(endadd[0], endadd[1]))

        if moves[move\_count][2] != emp:

            globals()[f'piece{list\_piece\_numbers[endadd[0]][endadd[1]]}'].ht()

            del globals()[f'piece{list\_piece\_numbers[endadd[0]][endadd[1]]}']

        #Modifying the list\_piecenumbers

        list\_piece\_numbers[endadd[0]][endadd[1]] = list\_piece\_numbers[startadd[0]][startadd[1]]

        list\_piece\_numbers[startadd[0]][startadd[1]]=0

    elif moves[move\_count][0] == "Castling":

        king\_startadd, king\_endadd = moves[move\_count][1][0], moves[move\_count][1][1]

        rook\_startadd, rook\_endadd = moves[move\_count][2][0], moves[move\_count][2][1]

        for startadd, endadd in ((king\_startadd, king\_endadd), (rook\_startadd, rook\_endadd)):

            #Changes in replay\_list2d

            replay\_list2d[endadd[0]][endadd[1]]=replay\_list2d[startadd[0]][startadd[1]]

            replay\_list2d[startadd[0]][startadd[1]]=["", "", 0, 0]

            replay\_list2d[endadd[0]][endadd[1]][2]=endadd[0]

            replay\_list2d[endadd[0]][endadd[1]][3]=endadd[1]

            #Playing Sounds

            if volume\_toggle == True:

                playsound("./Sounds/castling.mp3",False)

            #Making the piece move on the board

            globals()[f'piece{list\_piece\_numbers[startadd[0]][startadd[1]]}'].goto(coord\_from\_add(endadd[0], endadd[1]))

            #Modifying the list\_piecenumbers

            list\_piece\_numbers[endadd[0]][endadd[1]] = list\_piece\_numbers[startadd[0]][startadd[1]]

            list\_piece\_numbers[startadd[0]][startadd[1]]=0

    elif moves[move\_count][0] == "Enpassant":

        #Changes in replay\_list2d

        startadd, endadd = moves[move\_count][1], moves[move\_count][2]

        victimadd = moves[move\_count][3]

        replay\_list2d[endadd[0]][endadd[1]]=replay\_list2d[startadd[0]][startadd[1]]

        replay\_list2d[startadd[0]][startadd[1]]=["", "", 0, 0]

        replay\_list2d[endadd[0]][endadd[1]][2]=endadd[0]

        replay\_list2d[endadd[0]][endadd[1]][3]=endadd[1]

        replay\_list2d[victimadd[0]][victimadd[1]] = emp

        #Playing Sounds

        if volume\_toggle == True:

            playsound("./Sounds/capture.mp3",False)

        #Making the piece move on the board

        globals()[f'piece{list\_piece\_numbers[startadd[0]][startadd[1]]}'].goto(coord\_from\_add(endadd[0], endadd[1]))

        globals()[f'piece{list\_piece\_numbers[victimadd[0]][victimadd[1]]}'].ht()

        del globals()[f'piece{list\_piece\_numbers[victimadd[0]][victimadd[1]]}']

        #Modifying the list\_piecenumbers

        list\_piece\_numbers[endadd[0]][endadd[1]] = list\_piece\_numbers[startadd[0]][startadd[1]]

        list\_piece\_numbers[startadd[0]][startadd[1]] = 0

        list\_piece\_numbers[victimadd[0]][victimadd[1]] = 0

    elif  moves[move\_count][0] == "Pawnpromotion":

        #Changes in replay\_list2d

        replay\_list2d[moves[move\_count][1][0]][moves[move\_count][1][1]] = moves[move\_count][2]

        #Modifying the shape of the piece

        globals()[f'piece{list\_piece\_numbers[moves[move\_count][1][0]][moves[move\_count][1][1]]}'].shape(moves[move\_count][2][0].title())

    move\_count+=1

    try:

        if moves[move\_count][0] == "Pawnpromotion":

            next\_move()

    except:

        pass

    home.wn.onkeypress(previous\_move, "Left")

    home.wn.onkeypress(next\_move, "Right")

#Displays A,B,C, ...H and 1,2,3...8 on the side of the board for whiteview and blackview taking into account the value of boardview

def display\_labels(view):

    home.wn.tracer(0)

    #Deletes the OLD labels

    try:

        home.show\_labels(None, None, None, None, False, True)

    except:

        pass

    try:

        home.show\_labels(None, None, None, None, True, True)

    except:

        pass

    home.wn.tracer(1)

    #Creates the NEW labels

    distance=17

    startdistance=(sqsize/2)+8

    if view=="white":

        home.show\_labels((-size/2)-distance,(-size/2)-distance + verticaldrift, sqsize, startdistance, False)

    elif view=="black":

        home.show\_labels((-size/2)-distance,(-size/2)-distance + verticaldrift, sqsize, startdistance, True)

#Can be used to create a SINGLE piece at the desired board location (UNLIKE pieces\_setup in game.py which creates ALL the pieces in the BASE configuration)

def create\_chess\_piece(row,col,identity, colour):

    global piece\_count, list\_piece\_numbers

    piece\_count+=1

    list\_piece\_numbers[row][col] = piece\_count

    home.wn.tracer(0)

    globals()[f'piece{piece\_count}']=turtle.Turtle()

    globals()[f'piece{piece\_count}'].ht()

    globals()[f'piece{piece\_count}'].pu()

    globals()[f'piece{piece\_count}'].shape(identity.title())

    stretch\_piece = stretch\_square \* 0.2

    border\_width = int(stretch\_piece \* 4)

    globals()[f'piece{piece\_count}'].shapesize(stretch\_piece,stretch\_piece,border\_width)

    globals()[f'piece{piece\_count}'].speed(4)

    if colour == "black":

        globals()[f'piece{piece\_count}'].color(constant.BLACKPIECECLR)

        globals()[f'piece{piece\_count}'].pencolor("black")

    elif colour == "white":

        globals()[f'piece{piece\_count}'].color(constant.WHITEPIECECLR)

        globals()[f'piece{piece\_count}'].pencolor("black")

    globals()[f'piece{piece\_count}'].goto(coord\_from\_add(row, col))

    globals()[f'piece{piece\_count}'].st()

#Return: Central pixel coordinates as a tuple (IMP: Using the global variable boardview, it also acknowledges the changes in BOARD VIEW and returns the coordinates accordingly)

def coord\_from\_add(row, col):

    step=size/8

    if boardview=="white":

        return ((col\*step)-(4\*step)+(step/2)+drift,(4\*step)-(row\*step)-(step/2)+verticaldrift)

    elif boardview=="black":

        return ((-1 \* ((col\*step)-(4\*step)+(step/2)))+drift,(-1 \* ((4\*step)-(row\*step)-(step/2)))+verticaldrift)

def configure\_list(config\_num, update\_moves = False):

    global moves

    temp\_list2d = logic.copy\_of\_init\_list2d()

    mcount = 0

    row\_count = 0

    chess\_not = None

    turn = "white"

    while mcount<config\_num:

        temp\_list2d\_start\_of\_current\_move=[]

        for i in temp\_list2d:

            temp=[]

            for j in i:

                temp.append(j.copy())

            temp\_list2d\_start\_of\_current\_move.append(temp)

        #Modifying temp\_list2d to take it to the desired configuration

        if moves[mcount][0] not in ("Castling", "Enpassant", "Pawnpromotion"):

            srow, scol = moves[mcount][0]

            erow, ecol = moves[mcount][1]

            #Getting the chess notation of the move

            cn\_srow, cn\_scol, cn\_erow, cn\_ecol = srow, scol, erow, ecol

            cn\_castle = False

            if mcount != (len(moves)-1) and moves[mcount+1][0] == "Pawnpromotion":

                cn\_pppiece = moves[mcount+1][2][0]

            else:

                cn\_pppiece = ''

            temp\_list2d[erow][ecol]=temp\_list2d[srow][scol]

            temp\_list2d[srow][scol]=["", "", 0, 0]

            temp\_list2d[erow][ecol][2] = erow

            temp\_list2d[erow][ecol][3] = ecol

        elif moves[mcount][0] == "Castling":

            #Getting the chess notation of the move

            kcol, rcol = moves[mcount][1][1], moves[mcount][2][1]

            cn\_srow, cn\_scol, cn\_erow, cn\_ecol = None, kcol, None, rcol

            cn\_castle = True

            cn\_pppiece = ''

            king\_startadd, king\_endadd = moves[mcount][1][0], moves[mcount][1][1]

            rook\_startadd, rook\_endadd = moves[mcount][2][0], moves[mcount][2][1]

            for startadd, endadd in ((king\_startadd, king\_endadd), (rook\_startadd, rook\_endadd)):

                temp\_list2d[endadd[0]][endadd[1]]=temp\_list2d[startadd[0]][startadd[1]]

                temp\_list2d[startadd[0]][startadd[1]]=["", "", 0, 0]

                temp\_list2d[endadd[0]][endadd[1]][2]=endadd[0]

                temp\_list2d[endadd[0]][endadd[1]][3]=endadd[1]

        elif moves[mcount][0] == "Enpassant":

            srow, scol = moves[mcount][1]

            erow, ecol = moves[mcount][2]

            vicrow, viccol = moves[mcount][3]

            #Getting the chess notation of the move

            cn\_srow, cn\_scol, cn\_erow, cn\_ecol = srow, scol, erow, ecol

            cn\_castle = False

            cn\_pppiece = ''

            temp\_list2d[erow][ecol]=temp\_list2d[srow][scol]

            temp\_list2d[srow][scol]=["", "", 0, 0]

            temp\_list2d[erow][ecol][2] = erow

            temp\_list2d[erow][ecol][3] = ecol

            temp\_list2d[vicrow][viccol] = emp

        elif  moves[mcount][0] == "Pawnpromotion":

            temp\_list2d[moves[mcount][1][0]][moves[mcount][1][1]] = moves[mcount][2]

            mcount+=1

            continue

        mcount += 1

        if not update\_moves:

            continue

        cn\_checkinfo = logic.get\_game\_situation(temp\_list2d)[:2]

        chess\_not = get\_chess\_notation(temp\_list2d\_start\_of\_current\_move, cn\_srow, cn\_scol, cn\_erow, cn\_ecol, cn\_checkinfo, cn\_castle, cn\_pppiece)

        #Adding the move to frame\_moves

        label\_chess\_not = Label(frame\_moves.viewPort, text = chess\_not, bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('Comic Sans', 15, 'bold'), bd = 2, highlightthickness = 2, highlightbackground = constant.LIGHTBGTEXTCLR)

        Label(frame\_moves.viewPort, text = row\_count + 1, bg = constant.DARKBGCLR, fg = constant.DARKBGTEXTCLR, font = ('Comic Sans', 15, 'bold'), bd = 2, highlightthickness = 2, highlightbackground = constant.LIGHTBGTEXTCLR).grid(row = row\_count, column = 0, sticky = NSEW, padx = 1, pady = 1, ipady = 5)

        if turn == "white":

            label\_chess\_not.grid(row = row\_count, column = 1, sticky = NSEW, padx = 1, pady = 1, ipady = 5)

            turn = "black"

        elif turn == "black":

            label\_chess\_not.grid(row = row\_count, column = 2, sticky = NSEW, padx = 1, pady = 1, ipady = 5)

            turn = "white"

            row\_count += 1

    return temp\_list2d

#Deletes all the existing turtle chess pieces on the board, creates new turtle chess pieces for the desired configuration. Updates/Resets replay\_list2d, list\_of\_piece\_numbers, piece\_count and move\_count as well.

def configure\_pieces(config\_num):

    global replay\_list2d, list\_piece\_numbers, piece\_count, move\_count, frame\_moves, moves

    #Reconfiguring the board

    home.wn.tracer(0)

    for i in range(1,piece\_count+1):

        try:

            globals()[f'piece{i}'].ht()

            globals()[f'piece{i}'].clear()

            del globals()[f'piece{i}']

        except:

            pass

    #Initialisation of some variables

    try:

        del replay\_list2d, list\_piece\_numbers, piece\_count, move\_count

    except:

        pass

    list\_piece\_numbers = [[0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0]]

    piece\_count = 0

    replay\_list2d = configure\_list(config\_num, False).copy()

    move\_count = config\_num

    update\_timers(config\_num - 1)

    for i in range(8):

        for j in range(8):

            if replay\_list2d[i][j] != emp:

                create\_chess\_piece(i, j, replay\_list2d[i][j][0], logic.colour(replay\_list2d[i][j]))

    home.wn.tracer(1)

    home.wn.listen()

    home.wn.onkeypress(previous\_move, "Left")

    home.wn.onkeypress(next\_move, "Right")

def configure\_game():

    home.wn.tracer(0)

    #Hiding and Deleting all the board turtles on the screen

    for x in range(8):

        for y in range(8):

            try:

                globals()[f't{x}{y}'].ht()

                globals()[f't{x}{y}'].clear()

                del globals()[f't{x}{y}']

            except:

                pass

    for x in range(8):

        for y in range(8):

            try:

                globals()[f'tb{x}{y}'].ht()

                globals()[f'tb{x}{y}'].clear()

                del globals()[f'tb{x}{y}']

            except:

                pass

    board\_setup()

    configure\_pieces(move\_count)

    home.wn.tracer(1)

def switch\_view():

    global boardview

    if boardview == "black":

        boardview="white"

        display\_labels("white")

    elif boardview == "white":

        boardview="black"

        display\_labels("black")

    configure\_pieces(move\_count)

def replay\_main(mat\_no):

    def on\_turtle\_close():

        home.program\_running = False

        home.root\_turtle.destroy()

        try:

            root\_save\_configuration.destroy()

        except:

            pass

        try:

            root\_saved\_configurations.destroy()

        except:

            pass

        database\_functions.close\_connection()

    home.root\_turtle.protocol("WM\_DELETE\_WINDOW", on\_turtle\_close)

    #Receiving all the details of the desired match from the database

    date, start\_time, end\_time, duration, white\_player, black\_player, winner, moves, times, min\_time, increment = database\_functions.receive\_game\_details(mat\_no)

    home.wn.clearscreen()

    home.wn.bgcolor("black")

    white\_name = white\_player.title()

    black\_name = black\_player.title()

    if winner.lower() == 'w':

        home.root\_turtle.title(f'\*{white\_name} vs {black\_name}')

    elif winner.lower() == 'b':

        home.root\_turtle.title(f'{white\_name} vs \*{black\_name}')

    elif winner.lower() == 'd':

        home.root\_turtle.title(f'\*{white\_name} vs \*{black\_name}')

    elif winner.lower() == 's':

        home.root\_turtle.title(f'#{white\_name} vs #{black\_name}')

    print(moves)

    print(times)

    logic.define\_basic\_global\_variables()

    define\_basic\_global\_variables(mat\_no, moves, times, min\_time, increment)

    board\_setup()

    configure\_pieces(0)

    #Adding chess notations of all moves to frame\_moves

    configure\_list(len(moves), True)

    home.wn.tracer(1)

logic.py

import logmessage

def define\_basic\_global\_variables():

    global blrook1, blhor1, blbish1, blqueen, blking, blbish2, blhor2, blrook2, p1, p2, p3, p4, p5, p6, p7, p8,emp,P1, P2, P3, P4, P5, P6, P7, P8, whrook1, whhor1, whbish1, whqueen, whking, whbish2, whhor2, whrook2, colour, list2d, List\_of\_Moves, chance, enpassant\_possible, pawnnum

    #Creating White pieces

    whrook1=["rook", "\*][", 7, 0, False] #Identity(no shortforms. Full name), Symbol(It will be modified based on the "piece" choice given by the user), Rownumber, Columnnumber, Moved?

    whrook2=["rook", "\*][", 7, 7, False] #The numbers for rownumber and columnumber are based on 2D list indexing and not Usual Chess Address

    whhor1=["horse", "\*/>", 7, 1]

    whhor2=["horse", "\*/>", 7, 6]

    whbish1=["bishop", "\*A", 7, 2]

    whbish2=["bishop", "\*A", 7, 5]

    whqueen=["queen", "\*Q", 7, 3]

    whking=["king", "\*$", 7, 4, False]

    P1=["pawn", "\*^", 6, 0, False] #False indicating that the pawn cannot be a victim of en passant capture. When it is susceptible, it will be changed to True.

    P2=["pawn", "\*^", 6, 1, False]

    P3=["pawn", "\*^", 6, 2, False]

    P4=["pawn", "\*^", 6, 3, False]

    P5=["pawn", "\*^", 6, 4, False]

    P6=["pawn", "\*^", 6, 5, False]

    P7=["pawn", "\*^", 6, 6, False]

    P8=["pawn", "\*^", 6, 7, False]

    #Creating Black pieces

    blrook1=["rook", "][", 0, 0, False] #Identity(no shortforms. Full name), Symbol(It will be modified based on the "piece" choice given by the user), Rownumber, Columnnumber

    blrook2=["rook", "][", 0, 7, False] #The numbers for rownumber and columnumber are based on 2D list indexing and not Usual Chess Address

    blhor1=["horse", "/>", 0, 1]

    blhor2=["horse", "/>", 0, 6]

    blbish1=["bishop", "A", 0, 2]

    blbish2=["bishop", "A", 0, 5]

    blqueen=["queen", "Q", 0, 3]

    blking=["king", "$", 0, 4, False]

    p1=["pawn", "^", 1, 0, False]

    p2=["pawn", "^", 1, 1, False]

    p3=["pawn", "^", 1, 2, False]

    p4=["pawn", "^", 1, 3, False]

    p5=["pawn", "^", 1, 4, False]

    p6=["pawn", "^", 1, 5, False]

    p7=["pawn", "^", 1, 6, False]

    p8=["pawn", "^", 1, 7, False]

    emp=["", "", 0, 0] #Empty

    list2d=[[blrook1, blhor1, blbish1, blqueen, blking, blbish2, blhor2, blrook2],[p1, p2, p3, p4, p5, p6, p7, p8],[emp]\*8,[emp]\*8,[emp]\*8,[emp]\*8,[P1, P2, P3, P4, P5, P6, P7, P8],[whrook1, whhor1, whbish1, whqueen, whking, whbish2, whhor2, whrook2]]

    List\_of\_Moves=[]

    chance="white"

    enpassant\_possible=False #Is an en passant capture possible in the next move?

    pawnnum=1 #A number used to identify the pieces which are formed after a pawnpromotion

def copy\_of\_init\_list2d():

    #None of the below variables are global variables. ALL of them are LOCAL variables.

    #Creating White pieces

    whrook1=["rook", "\*][", 7, 0, False] #Identity(no shortforms. Full name), Symbol(It will be modified based on the "piece" choice given by the user), Rownumber, Columnnumber, Moved?

    whrook2=["rook", "\*][", 7, 7, False] #The numbers for rownumber and columnumber are based on 2D list indexing and not Usual Chess Address

    whhor1=["horse", "\*/>", 7, 1]

    whhor2=["horse", "\*/>", 7, 6]

    whbish1=["bishop", "\*A", 7, 2]

    whbish2=["bishop", "\*A", 7, 5]

    whqueen=["queen", "\*Q", 7, 3]

    whking=["king", "\*$", 7, 4, False]

    P1=["pawn", "\*^", 6, 0, False] #False indicating that the pawn cannot be a victim of en passant capture. When it is susceptible, it will be changed to True.

    P2=["pawn", "\*^", 6, 1, False]

    P3=["pawn", "\*^", 6, 2, False]

    P4=["pawn", "\*^", 6, 3, False]

    P5=["pawn", "\*^", 6, 4, False]

    P6=["pawn", "\*^", 6, 5, False]

    P7=["pawn", "\*^", 6, 6, False]

    P8=["pawn", "\*^", 6, 7, False]

    #Creating Black pieces

    blrook1=["rook", "][", 0, 0, False] #Identity(no shortforms. Full name), Symbol(It will be modified based on the "piece" choice given by the user), Rownumber, Columnnumber

    blrook2=["rook", "][", 0, 7, False] #The numbers for rownumber and columnumber are based on 2D list indexing and not Usual Chess Address

    blhor1=["horse", "/>", 0, 1]

    blhor2=["horse", "/>", 0, 6]

    blbish1=["bishop", "A", 0, 2]

    blbish2=["bishop", "A", 0, 5]

    blqueen=["queen", "Q", 0, 3]

    blking=["king", "$", 0, 4, False]

    p1=["pawn", "^", 1, 0, False]

    p2=["pawn", "^", 1, 1, False]

    p3=["pawn", "^", 1, 2, False]

    p4=["pawn", "^", 1, 3, False]

    p5=["pawn", "^", 1, 4, False]

    p6=["pawn", "^", 1, 5, False]

    p7=["pawn", "^", 1, 6, False]

    p8=["pawn", "^", 1, 7, False]

    emp=["", "", 0, 0] #Empty

    return [[blrook1, blhor1, blbish1, blqueen, blking, blbish2, blhor2, blrook2],[p1, p2, p3, p4, p5, p6, p7, p8],[emp]\*8,[emp]\*8,[emp]\*8,[emp]\*8,[P1, P2, P3, P4, P5, P6, P7, P8],[whrook1, whhor1, whbish1, whqueen, whking, whbish2, whhor2, whrook2]]

#generation of a requests link

def generate\_code():

    return 1234

def encrypt(x):

    return x

#Aim: To display any list for debugging purposes

def displaylist(lst):

    colwd=25

    print("-"\*(int(5\*colwd)))

    for i in lst:

        for j in i:

            if colour(j)!="empty":

                clr=colour(j)

            else:

                clr=""

            print(clr+" "+j[0], end=" "\*((colwd//2)-len(j[0]+" "+clr))+"|" + " "\*(colwd//7))

        print()

        print("-"\*(int(5\*colwd)))

#Parameters: Starting address and the ending address as tuples

#Function: In list2d, the piece present in the starting address is moved to the ending address making 3 important changes

def move(add1, add2):

    global list2d

    list2d[add2[0]][add2[1]]=list2d[add1[0]][add1[1]]

    list2d[add1[0]][add1[1]]=emp

    list2d[add2[0]][add2[1]][2]=add2[0]

    list2d[add2[0]][add2[1]][3]=add2[1]

#Parameters: Starting address and the ending address as tuples, the list in which the move is to be made

#Function: In the list given as a parameter, the piece present in the starting address is moved to the ending address making 3 important changes

def validatemove(add1,add2,testf2d):

    testf2d[add2[0]][add2[1]]=testf2d[add1[0]][add1[1]]

    testf2d[add1[0]][add1[1]]=["", "", 0, 0]

    testf2d[add2[0]][add2[1]]=[testf2d[add2[0]][add2[1]][0], testf2d[add2[0]][add2[1]][1], add2[0], add2[1]]

#Aim: To get the colour of the piece

#Parameter: Piece

#Result: "black", "white" or "empty"

def colour(piece):

    if piece[1]=="":

        return "empty"

    elif piece[1][0]=="\*":

        return "white"

    else:

        return "black"

#Given a color, the opposite colour is returned

def oppcolour(clr): #Opposite Colour

    if clr=="black":

        return "white"

    elif clr=="white":

        return "black"

#Aim: To check if two pieces belong to the same team

#Parameters: Example: whrook1, whbish2

#Return: True if they belong to the same team and False if they belong to different teams or if one of the pieces is ["", "", 0, 0]-->empty space

#Explanation: whrook1=["rook", "\*...", rownumber, columnnumber], whbish2=["bishop", "\*...", rownumber, columnnumber]. If the zeroth index of the 1st index of each of the two pieces is equal to \*, both are white. If both do not contain \*, they are both black. If only one of them contain, they are opposite teams. whrook1 and whbish2 belong to the same team.

def sameteam(pc1, pc2):

    if pc1[1]!="" and pc2[1]!="":

        if pc1[1][0]=="\*" and pc2[1][0]=="\*":

            return True

        elif pc1[1][0]!="\*" and pc2[1][0]!="\*":

            return True

        else:

            return False

    else:

        return False

#Aim: (To detect a CHECK, To check if a move will be intercepted by a piece) Given a starting address(based on 2D list indexing) and an ending address(same indexing), finding out all the pieces in between the two addresses on the CHESSBOARD STARTING from the starting address and moving to the ending address.

#Parameters: starting address[given as a tuple], ending address[given as a tuple], a 2 dimensional list

#Note: If the starting address and the ending address lie in the same row, same column or same diagonal, the operation can be done. Otherwise, it cannot be done.

#Return: if pieces present in between return the Pieces in a tuple; if no pieces present, return empty tuple; if operation cannot be carried out, return "Invalid"

#Example: parameters --> (0,1), (7,1), list2d  ;  output --> (value of p2, value of P2, value of whhor1)   ==> Output is a tuple not a list AND the element/piece present in the starting address is NOT given in the output

#Explanation: [Refer to the chessboard on google docs] Between the two addresses given, p2, P2, whhor1 are present. blhor1 is NOT included.

def piecesbetween(add1, add2, f2d):

    if add1==add2:

        return ()

    if add1[0]==add2[0] or add1[1]==add2[1] or abs(add1[0] - add2[0])==abs(add1[1]- add2[1]):

        path=()

        if add1[0]==add2[0]:

            stepr=0

        else:

            stepr=(add2[0]-add1[0])//abs(add2[0]-add1[0])

        if add1[1]==add2[1]:

            stepc=0

        else:

            stepc=(add2[1]-add1[1])//abs(add2[1]-add1[1])

        curadd=(add1[0]+stepr, add1[1]+stepc) #Current address at which we are present and checking for pieces

        complete=False

        while complete==False:

            if curadd==add2:

                complete=True

            if f2d[curadd[0]][curadd[1]] ==["", "", 0, 0]:

                pass

            else:

                path+=(f2d[curadd[0]][curadd[1]],)

            curadd=(curadd[0]+stepr, curadd[1]+stepc) #Changing the value of the current address to the next address which we should check

        return path

    else:

        return "Invalid"

#Returns the rowstepcount and columnstepcount between the starting address and the ending address (NOT in a tuple)

#Possible return values are: 0,1 ; 0,-1 ; 1,0 ; -1,0 ; 1,1 ; 1,-1 ; -1,1 ; -1,-1

def stepcount(add1, add2):

    if add1[0]==add2[0]:

        stepr=0

    else:

        stepr=(add2[0]-add1[0])//abs(add2[0]-add1[0])

    if add1[1]==add2[1]:

        stepc=0

    else:

        stepc=(add2[1]-add1[1])//abs(add2[1]-add1[1])

    return stepr, stepc

#Aim: to check if a move given by the user is valid

#Parameters: Starting address(2D Indexing)(tuple), Ending Address(2D Indexing)(tuple), 2 dimensional list

#Return: True if the move is valid and False if the move is invalid.

def rookmove(add1, add2, f2d):

    level=0

    #1st Level

    if add1[0]==add2[0]: #row numbers

        level+=1

    elif add1[1]==add2[1]: #Column Numbers

        level+=1

    else:

        return False

    #2nd Level

    pathpieces=piecesbetween(add1, add2, f2d)

    if pathpieces ==():

        level+=1

    elif len(pathpieces)==1 and (pathpieces[0][2], pathpieces[0][3])==add2 and sameteam(pathpieces[0], f2d[add1[0]][add1[1]])==False:

        level+=1

    if level==2:

        return True

    else:

        return False

#For a horse, we need not worry about whether there is a piece in between or not.

def hormove(add1, add2, f2d):

    level=0

    #1st Level

    if (add1[0]==add2[0]+1 or add1[0]==add2[0]-1) and (add1[1]==add2[1]+2 or add1[1]==add2[1]-2):

        level+=1

    elif (add1[0]==add2[0]+2 or add1[0]==add2[0]-2) and (add1[1]==add2[1]+1 or add1[1]==add2[1]-1):

        level+=1

    else:

        return False

    #2nd Level

    if sameteam(f2d[add2[0]][add2[1]], f2d[add1[0]][add1[1]])==False:

        level+=1

    if level==2:

        return True

    else:

        return False

def bishmove(add1, add2, f2d):

    level=0

    #1st Level

    if abs(add1[0]-add2[0])==abs(add1[1]-add2[1]):

        level+=1

    else:

        return False

    #2nd Level

    pathpieces=piecesbetween(add1, add2, f2d)

    if pathpieces ==():

        level+=1

    elif len(pathpieces)==1 and (pathpieces[0][2], pathpieces[0][3])==add2 and sameteam(pathpieces[0], f2d[add1[0]][add1[1]])==False:

        level+=1

    if level==2:

        return True

    else:

        return False

def queenmove(add1, add2, f2d):

    level=0

    #1st Level

    if add1[0]==add2[0] or add1[1]==add2[1]:

        level+=1

    elif abs(add1[0]-add2[0])==abs(add1[1]-add2[1]):

        level+=1

    else:

        return False

    #2nd Level

    pathpieces=piecesbetween(add1, add2, f2d)

    if pathpieces ==():

        level+=1

    elif len(pathpieces)==1 and (pathpieces[0][2], pathpieces[0][3])==add2 and sameteam(pathpieces[0], f2d[add1[0]][add1[1]])==False:

        level+=1

    if level==2:

        return True

    else:

        return False

def kingmove(add1, add2, f2d):

    level=0

    #1st Level

    #(add1[0]==add2[0]+1 or add1[0]==add2[0]-1 or add1[0]==add2[0]) and (add1[1]==add2[1] or add1[1]==add2[1]-1 or add1[1]==add2[1]+1)

    if (add1[0]-add2[0])\*\*2 + (add1[1]-add2[1])\*\*2<=2:

        level+=1

    else :

        return False

    #2nd Level

    pathpieces=piecesbetween(add1, add2, f2d)

    if pathpieces ==():

        level+=1

    elif len(pathpieces)==1 and (pathpieces[0][2], pathpieces[0][3])==add2 and sameteam(pathpieces[0], f2d[add1[0]][add1[1]])==False:

        level+=1

    if level==2:

        return True

    else:

        return False

#Parameters: Two addresses are given as tuples, 2 dimensional list

#Will return True if castling is possible, False if not possible

def castle(add1, add2, f2d):

    if not(0<=add1[0]<=7 and 0<=add1[1]<=7 and 0<=add2[0]<=7 and 0<=add2[1]<=7):

        return (False, ((0,0), (0,0)), ((0,0), (0,0)))

    #Converting add2 to possible add of rook

    if add2[1] > add1[1]:

        add2 = (add2[0], add2[1] + 1)

    elif add2[1] < add1[1]:

        add2 = (add2[0], add2[1] - 2)

    #Getting the relevant pieces

    pc1=f2d[add1[0]][add1[1]]

    pc2=f2d[add2[0]][add2[1]]

    caslevel1=False

    if pc1[0]=="king" and pc2[0]=="rook":

        caslevel1=True

    logmessage.log("       Caslevel1: (King has been moved correctly for a castle) ", caslevel1)

    caslevel1\_1=False

    if caslevel1==True:

        if abs(add1[1]-add2[1])==3:

            if f2d[add1[0]][pc2[3]-1]==["", "", 0, 0] and f2d[add1[0]][pc2[3]-2]==["", "", 0, 0]:

                caslevel1\_1=True

        elif abs(add1[1]-add2[1])==4:

            if f2d[add1[0]][pc2[3]+1]==["", "", 0, 0] and f2d[add1[0]][pc2[3]+2]==["", "", 0, 0] and f2d[add1[0]][pc2[3]+3]==["", "", 0, 0] :

                caslevel1\_1=True

    if caslevel1==True:

        logmessage.log("       Caslevel1\_1: (Squares in between are empty) ", caslevel1\_1)

    caslevel2=False

    if caslevel1\_1==True and check((pc1[2], pc1[3]), f2d)[0]==False:

        caslevel2=True

    if caslevel1\_1==True:

        logmessage.log("       Caslevel2: (Castling king is not facing a check) ", caslevel2)

    caslevel3=False

    if caslevel2==True and sameteam(pc1, pc2)==True and pc1[4]==False and pc2[4]==False and len(piecesbetween(add1, add2, f2d))==1:

        caslevel3=True

    if caslevel2==True:

        logmessage.log("       Caslevel3: (Both the pieces belong to the same team) ", caslevel3)

    caslevel4=False

    if caslevel3==True:

        clist2d=[]

        for i in f2d:

            temp=[]

            for j in i:

                temp.append(j.copy())

            clist2d.append(temp)

        if abs(add1[1]-add2[1])==3:

            validatemove((pc1[2], pc1[3]), (pc1[2], pc1[3]+1), clist2d)

            if check((pc1[2], pc1[3]+1), clist2d)[0]==False:

                caslevel4=True

                newkingadd=(pc1[2], pc1[3]+2)

                newrookadd=(pc2[2], pc2[3]-2)

        elif abs(add1[1]-add2[1])==4:

            validatemove((pc1[2], pc1[3]), (pc1[2], pc1[3]-1), clist2d)

            if check((pc1[2], pc1[3]-1), clist2d)[0]==False:

                caslevel4=True

                newkingadd=(pc1[2], pc1[3]-2)

                newrookadd=(pc2[2], pc2[3]+3)

    if caslevel4==True:

        castleinfo=(True, ((pc1[2], pc1[3]), newkingadd), ((pc2[2], pc2[3]), newrookadd))

        return (True, ((pc1[2], pc1[3]), newkingadd), ((pc2[2], pc2[3]), newrookadd)) #Castle is possible or not?, King's starting and ending add, rook's S and E add

    elif caslevel4==False:

        return (False, ((0,0), (0,0)), ((0,0), (0,0)))

#Movement of the pawn is slightly complicated. Depending on whether it is black or white, it can only move in ONE direction. It can move CROSS only if something can be attacked.

#Parameters: Starting address[2D indexing], ending address[2D indexing], 2 dimensional list

#Return: if the move is straight and there is 'NO piece in the ending address' return True; if the move is cross there is a piece of the OPPOSITE team in the ending address return True

#Incorporated starting double move and En Passant also

#The returned tuple consists of 3 boolean values and 1 tuple: Is the move possible?, Is the moving pawn a possible en passant victim in the next move?, Is the moving pawn killing another pawn by en passant?, The address of the pawn which has to be killed.

def pawnmove(add1, add2, f2d):

    if f2d[add1[0]][add1[1]][1]=="\*^": #White Checking

        if  add2[0]-add1[0]==-1: #For white, row should decrease.

            if add1[1]==add2[1]: #Checking if the column is the same

                if f2d[add2[0]][add2[1]][1]=="": #Checking if the ending address is empty

                    return (True, False, False, None)

                else:

                    return (False, False,False, None)

            elif abs(add2[1]-add1[1])==1: #Checking if the pawn has moved cross

                if sameteam(f2d[add1[0]][add1[1]], f2d[add2[0]][add2[1]])==False and f2d[add2[0]][add2[1]]!=["", "", 0, 0]: #Checking if the ending address is occupied by a black piece

                    return (True, False,False, None)

                elif f2d[add2[0]][add2[1]]==["", "", 0, 0] and f2d[add1[0]][add2[1]][0]=="pawn" and f2d[add1[0]][add2[1]][4]==True:

                    return (True, False, True, (add1[0],add2[1]))

                else:

                    return (False, False,False, None)

        elif add2[0]-add1[0]==-2 and add1[0]==6:

            if add1[1]==add2[1]: #Checking if the column is the same

                if f2d[add2[0]][add2[1]][1]=="": #Checking if the ending address is empty

                    return (True, True,False, None)

                else:

                    return (False, False,False, None)

        else:

            return (False, False, False, None)

    elif f2d[add1[0]][add1[1]][1]=="^": #Black Checking

        if  add2[0]-add1[0]==1: #For black, row should increase.

            if add1[1]==add2[1]: #Checking if the column is the same

                if f2d[add2[0]][add2[1]][1]=="": #Checking if the ending address is empty

                    return (True, False,False, None)

                else:

                    return (False, False,False, None)

            elif abs(add2[1]-add1[1])==1: #Checking if the pawn has moved cross

                if sameteam(f2d[add1[0]][add1[1]], f2d[add2[0]][add2[1]])==False and f2d[add2[0]][add2[1]]!=["", "", 0, 0]: #Checking if the ending address is occupied by a white piece

                    return (True, False,False, None)

                elif f2d[add2[0]][add2[1]]==["", "", 0, 0] and f2d[add1[0]][add2[1]][0]=="pawn" and f2d[add1[0]][add2[1]][4]==True:

                    return (True, False, True,(add1[0],add2[1]))

                else:

                    return (False, False,False, None)

        elif add2[0]-add1[0]==2 and add1[0]==1:

            if add1[1]==add2[1]: #Checking if the column is the same

                if f2d[add2[0]][add2[1]][1]=="": #Checking if the ending address is empty

                    return (True, True,False, None)

                else:

                    return (False, False,False, None)

        else:

            return (False, False,False, None)

    return (None, None,False, None)

#Parameters: First address, Second address, 2 dimensional list

#Function: Given that a piece CAN MOVE from the first address to the second address, extend return the addresses (tuples) of all the squares following the first address along the line(first add, second add) that are either free or occupied by a piece opposite to the piece present in the starting address.

#In short, it gives the reach of the piece present in the first address given that it can move from the first address to the second address.

def extend(add1, add2, f2d):

    r,c=add1[0], add1[1]

    if colour(f2d[r][c])=="white":

        for m in f2d:

            for n in m:

                if n[1]=="\*$":

                    extendking=n #King of the piece which is being moved. We need to see if the king of the piece which is moving suffers a check or not

    elif colour(f2d[r][c])=="black":

        for m in f2d:

            for n in m:

                if n[1]=="$":

                    extendking=n

    stepr, stepc=stepcount(add1, add2)

    piece=f2d[add1[0]][add1[1]]

    rowcounter, colcounter=add2[0], add2[1]

    tup=()

    while True:

        if 0<=rowcounter<=7 and 0<=colcounter<=7:

            if sameteam(piece, f2d[rowcounter][colcounter])==False:

                templist2d=[]

                for l in f2d:

                    temp=[]

                    for k in l:

                        temp.append(k.copy())

                    templist2d.append(temp)

                validatemove((r,c), (rowcounter,colcounter), templist2d)

                if check((extendking[2], extendking[3]), templist2d)[0]==False:

                    tup+=((rowcounter, colcounter),)

                if colour(f2d[rowcounter][colcounter])=="white" or colour(f2d[rowcounter][colcounter])=="black":

                    break

            else:

                break

            rowcounter+=stepr

            colcounter+=stepc

        else:

            break

    return tup

#Given an address and a 2 dimensional list, it returns the addresses of all the squares that the piece present in the starting address can go to as a TUPLE.

def legal(add,f2d):

    if colour(f2d[add[0]][add[1]])=="empty":

        return (False, ())

    global legaladdresses

    legaladdresses=()

    r,c=add[0], add[1]

    if colour(f2d[r][c])=="white":

        for m in f2d:

            for n in m:

                if n[1]=="\*$":

                    king=n #King of the piece which is being moved. We need to see if the king of the piece which is moving suffers a check or not

    elif colour(f2d[r][c])=="black":

        for m in f2d:

            for n in m:

                if n[1]=="$":

                    king=n

    def aroundcheck(add, f2d, a1, a2, a3, a4, a5, a6, a7, a8):

        global legaladdresses

        r,c=add[0], add[1]

        pc=f2d[r][c][0]

        ru=cl=8

        if r>=1:

            ru=r-1

        if c>=1:

            cl=c-1

        around=[a1,a2,a3,a4,a5,a6,a7,a8]

        addresses=[(r, c+1), (ru, c+1), (ru, c), (ru, cl), (r, cl), (r+1, cl), (r+1, c), (r+1, c+1)]

        for i in range(8):

            if around[i]==False:

                addresses[i]=(None, None)

        for i,j in addresses:

            try:

                if i!=None:

                    templist2d=[]

                    for l in f2d:

                        temp=[]

                        for k in l:

                            temp.append(k.copy())

                        templist2d.append(temp)

                    if pc=="pawn":

                        level1=pawnmove((r,c), (i,j), templist2d)[0] #Pawn attacking and moving has different rules.

                    else:

                        level1=not(sameteam(f2d[r][c], f2d[i][j])) #All other pieces have the same rules of attacking and moving

                    if level1!=True:

                        continue

                    validatemove((r,c), (i,j), templist2d)

                    if colour(f2d[r][c])=="white":

                        for m in templist2d:

                            for n in m:

                                if n[1]=="\*$":

                                    teamking=n #King of the piece which is being moved. We need to see if the king of the piece which is moving suffers a check or not

                    elif colour(f2d[r][c])=="black":

                        for m in templist2d:

                            for n in m:

                                if n[1]=="$":

                                    teamking=n

                    if check((teamking[2], teamking[3]), templist2d)[0]==False:

                        if pc=="king" or pc=="horse" or pc=="pawn":

                            legaladdresses+=((i,j),)

                        elif pc=="queen" or pc=="bishop" or pc=="rook":

                            if colour(f2d[i][j])=="empty":

                                for k in extend(add, (i,j), f2d):

                                    legaladdresses+=(k,)

                            else:

                                legaladdresses+=((i,j),)

                    if pc=="pawn" and colour(f2d[i][j])=="empty":

                        stepr=i-r

                        nextr=i+stepr

                        templist2d=[]

                        for l in f2d:

                            temp=[]

                            for k in l:

                                temp.append(k.copy())

                            templist2d.append(temp)

                        validatemove((r,c), (nextr, j), templist2d)

                        if pawnmove((r,c), (nextr, j), f2d)[0]==True:

                            if check((teamking[2], teamking[3]), templist2d)[0]==False:

                                legaladdresses+=((nextr, j),)

            except IndexError:

                pass

        if legaladdresses==():

            return (False, legaladdresses)

        else:

            return (True, legaladdresses)

    def aroundhorsecheck(add, f2d):

        global legaladdresses

        r=add[0]

        c=add[1]

        rowu=rowuu=coll=colll=8 #rowup, rowupup, columnleft, columnleftleft

        if r-1>=0:

            rowu=r-1

        if r-2>=0:

            rowuu=r-2

        if c-1>=0:

            coll=c-1

        if c-2>=0:

            colll=c-2

        addresses=((rowu, c+2),(r+1, c+2),(rowuu, c+1),(r+2, c+1),(rowu, colll),(r+1, colll),(rowuu, coll),(r+2, coll))

        for i, j in addresses:

            try:

                templist2d=[]

                for l in f2d:

                    temp=[]

                    for k in l:

                        temp.append(k.copy())

                    templist2d.append(temp)

                level1=not(sameteam(f2d[r][c], f2d[i][j]))

                if level1!=True:

                        continue

                validatemove((r,c), (i,j), templist2d)

                if colour(f2d[r][c])=="white":

                    for m in templist2d:

                        for n in m:

                            if n[1]=="\*$":

                                teamking=n #King of the piece which is being moved. We need to see if the king of the piece which is moving suffers a check or not

                elif colour(f2d[r][c])=="black":

                    for m in templist2d:

                        for n in m:

                            if n[1]=="$":

                                teamking=n

                if check((teamking[2], teamking[3]), templist2d)[0]==False:

                    legaladdresses+=((i,j),)

            except IndexError:

                pass

        if legaladdresses==():

            return (False, legaladdresses)

        else:

            return (True, legaladdresses)

    if check((king[2], king[3]), f2d)[0]==True and (f2d[r][c][0]=="rook" or f2d[r][c][0]=="bishop" or f2d[r][c][0]=="queen"):

        attpcs=check((king[2], king[3]), f2d)[1] #Attacking Pieces

        logmessage.log("Legal: Check=True: attpcs ", attpcs)

        for indattpc in attpcs: #Individual Attacking Piece

            rowf=indattpc[2] #Row number of the attacking piece

            colf=indattpc[3] #Column number of the attacking piece

            rowstep, colstep=stepcount((king[2], king[3]), (rowf, colf)) #Gives us the rowstep value and the colstep value to gradually move from the king's address to the attackers's address

            rowcounter=king[2]+rowstep

            colcounter=king[3]+colstep

            locations=()

            if indattpc[0]=="horse":

                locations+=(indattpc,)

            else:

                while True:

                    if rowcounter==rowf and colcounter==colf:

                        locations+=(indattpc,)

                        break

                    bwsquare=["", "", rowcounter, colcounter] #Between square

                    locations+=(bwsquare,)

                    rowcounter+=rowstep

                    colcounter+=colstep

            for indsquares in locations: #Individual squares in between the king and the attacking species

                templist2d=[]

                for l in f2d:

                    temp=[]

                    for k in l:

                        temp.append(k.copy())

                    templist2d.append(temp)

                checklevel1=False

                if f2d[r][c][0]=="rook":

                    if rookmove((r,c), (indsquares[2], indsquares[3]), templist2d)==True:

                        validatemove((r,c), (indsquares[2], indsquares[3]), templist2d)

                        checklevel1=True

                elif f2d[r][c][0]=="bishop":

                    if bishmove((r,c), (indsquares[2], indsquares[3]), templist2d)==True:

                        validatemove((r,c), (indsquares[2], indsquares[3]), templist2d)

                        checklevel1=True

                elif f2d[r][c][0]=="queen":

                    if queenmove((r,c), (indsquares[2], indsquares[3]), templist2d)==True:

                        validatemove((r,c), (indsquares[2], indsquares[3]), templist2d)

                        checklevel1=True

                if checklevel1==True:

                    if colour(f2d[r][c])=="white":

                        for m in templist2d:

                            for n in m:

                                if n[1]=="\*$":

                                    teamking=n #King of the piece which is being moved. We need to see if the king of the piece which is moving suffers a check or not

                    elif colour(f2d[r][c])=="black":

                        for m in templist2d:

                            for n in m:

                                if n[1]=="$":

                                    teamking=n

                    if check((teamking[2], teamking[3]), templist2d)[0]==False:

                        legaladdresses+=((indsquares[2], indsquares[3]),)

    else:

        if f2d[r][c][0]=="king":

            return aroundcheck((r,c), f2d, True, True, True, True, True, True, True, True)

        elif f2d[r][c][0]=="queen":

            return aroundcheck((r,c), f2d, True, True, True, True, True, True, True, True)

        elif f2d[r][c][0]=="rook":

            return aroundcheck((r,c), f2d, True, False, True, False, True, False, True, False)

        elif f2d[r][c][0]=="bishop":

            return aroundcheck((r,c), f2d, False, True, False, True, False, True, False, True)

        elif f2d[r][c][0]=="pawn":

            return aroundcheck((r,c), f2d, False, True, True, True,False,True,True,True)

        elif f2d[r][c][0]=="horse":

            return aroundhorsecheck((r,c), f2d)

    if legaladdresses==():

        return (False, legaladdresses)

    else:

        return (True, legaladdresses)

#Aim: Detection of a Check

#Parameters: king's position as a tuple, 2 dimensional list

#Result: if a check is possible, return (True, all the attacking pieces in a tuple). If not return (False, ())

def check(add,f2d):

    attack=()

    #Checking all pieces except Horse

    leftadd=(add[0], 0)

    left=piecesbetween(add, leftadd, f2d)

    rightadd=(add[0], 7)

    right=piecesbetween(add, rightadd, f2d)

    topadd=(0, add[1])

    top=piecesbetween(add, topadd, f2d)

    bottomadd=(7, add[1])

    bottom=piecesbetween(add, bottomadd, f2d)

    neadd=(add[0]-min(abs(add[0]-0), abs(7-add[1])),add[1]+min(abs(add[0]-0), abs(7-add[1])))  #North east - ne

    ne=piecesbetween(add, neadd, f2d)

    nwadd=(add[0]-min(abs(add[0]-0), abs(0-add[1])),add[1]-min(abs(add[0]-0), abs(0-add[1]))) #North west - nw

    nw=piecesbetween(add, nwadd, f2d)

    swadd=(add[0]+min(abs(add[0]-7), abs(0-add[1])),add[1]-min(abs(add[0]-7), abs(0-add[1]))) #South west - sw

    sw=piecesbetween(add, swadd, f2d)

    seadd=(add[0]+min(abs(add[0]-7), abs(7-add[1])),add[1]+min(abs(add[0]-7), abs(7-add[1]))) #South east - se

    se=piecesbetween(add, seadd, f2d)

    alldir=[left, right, top, bottom, ne, nw, se, sw]

    count=0

    for i in alldir:

        if i==():

            pass

        else:

            if i[0][0]=="horse":

                pass

            else:

                if sameteam(i[0], f2d[add[0]][add[1]])==False:

                    if (i[0][0]=="rook" or i[0][0]=="queen") and count in range(4):

                        attack+=(i[0],)

                    elif (i[0][0]=="bishop" or i[0][0]=="queen") and count in (4,5,6,7):

                        attack+=(i[0],)

                    elif i[0][0]=="pawn" and ((add[0]-i[0][2])\*\*2 + (add[1]-i[0][3])\*\*2)==2:

                        if i[0][1][0]=="\*" and count in (6,7):

                            attack+=(i[0],)

                        elif i[0][1][0]!="\*" and count in (4,5):

                            attack+=(i[0],)

                    elif i[0][0]=="king" and (i[0][2]-add[0])\*\*2 + (i[0][3]-add[1])\*\*2<=2:

                        return (True, (i[0],))

        count+=1

    #Checking Horse

    r=add[0]

    c=add[1]

    rowu=rowuu=coll=colll=8 #rowup, rowupup, columnleft, columnleftleft

    if r-1>=0:

        rowu=r-1

    if r-2>=0:

        rowuu=r-2

    if c-1>=0:

        coll=c-1

    if c-2>=0:

        colll=c-2

    addresses=((rowu, c+2),(r+1, c+2),(rowuu, c+1),(r+2, c+1),(rowu, colll),(r+1, colll),(rowuu, coll),(r+2, coll))

    for tup in addresses:

        try:

            if f2d[tup[0]][tup[1]][0]=="horse" and sameteam(f2d[tup[0]][tup[1]], f2d[r][c])==False:

                attack+=(f2d[tup[0]][tup[1]],)

        except IndexError:

            pass

    if attack==():

        return (False, attack)

    else:

        return (True, attack)

#Aim: [Underlying aim is to detect a checkmate] Given an address and its status(empty or non-empty), being able to detect whether any piece of the SPECIFIED colour can enter the address either by just occupying or attacking

#Parameters: address(tuple), empty=True or False, clr=what is the colour of the piece you are looking for to occupy the given address, 2 dimensional list

#Note: Either of the kings cannot be considered for moving into the required square.

#Return: if some piece satisfying the GIVEN conditions is present, return (True, pieces in a tuple), If ot, return (False, ())

def capture(add, empty, clr, f2d):

    attack=()

    leftadd=(add[0], 0)

    left=piecesbetween(add, leftadd, f2d)

    rightadd=(add[0], 7)

    right=piecesbetween(add, rightadd, f2d)

    topadd=(0, add[1])

    top=piecesbetween(add, topadd, f2d)

    bottomadd=(7, add[1])

    bottom=piecesbetween(add, bottomadd, f2d)

    neadd=(add[0]-min(abs(add[0]-0), abs(7-add[1])),add[1]+min(abs(add[0]-0), abs(7-add[1])))  #North east - ne

    ne=piecesbetween(add, neadd, f2d)

    nwadd=(add[0]-min(abs(add[0]-0), abs(0-add[1])),add[1]-min(abs(add[0]-0), abs(0-add[1]))) #North west - nw

    nw=piecesbetween(add, nwadd, f2d)

    swadd=(add[0]+min(abs(add[0]-7), abs(0-add[1])),add[1]-min(abs(add[0]-7), abs(0-add[1]))) #South west - sw

    sw=piecesbetween(add, swadd, f2d)

    seadd=(add[0]+min(abs(add[0]-7), abs(7-add[1])),add[1]+min(abs(add[0]-7), abs(7-add[1]))) #South east - se

    se=piecesbetween(add, seadd, f2d)

    alldir=[left, right, top, bottom, ne, nw, se, sw]

    count=0

    for i in alldir:

        if i==():

            pass

        else:

            if i[0][0]=="horse":

                pass

            else:

                if colour(i[0])==clr:

                    if (i[0][0]=="rook" or i[0][0]=="queen") and count in range(4):

                        attack+=(i[0],)

                    elif (i[0][0]=="bishop" or i[0][0]=="queen") and count in (4,5,6,7):

                        attack+=(i[0],)

                    elif i[0][0]=="pawn":

                        if empty==True and count in (2,3):

                            if clr=="black" and (add[0]-i[0][2]==1 or (add[0]-i[0][2]==2 and i[0][2]==1)):

                                attack+=(i[0],)

                            elif clr=="white" and (add[0]-i[0][2]==-1 or (add[0]-i[0][2]==-2 and i[0][2]==6)):

                                attack+=(i[0],)

                        elif empty==False:

                            if clr=="black" and count in (4,5) and ((add[0]-i[0][2])\*\*2 + (add[1]-i[0][3])\*\*2 ==2):

                                attack+=(i[0],)

                            elif clr=="white" and count in (6,7) and ((add[0]-i[0][2])\*\*2 + (add[1]-i[0][3])\*\*2 ==2):

                                attack+=(i[0],)

        count+=1

    #Checking Horse

    r=add[0]

    c=add[1]

    rowu=rowuu=coll=colll=8 #rowup, rowupup, columnleft, columnleftleft

    if r-1>=0:

        rowu=r-1

    if r-2>=0:

        rowuu=r-2

    if c-1>=0:

        coll=c-1

    if c-2>=0:

        colll=c-2

    addresses=((rowu, c+2),(r+1, c+2),(rowuu, c+1),(r+2, c+1),(rowu, colll),(r+1, colll),(rowuu, coll),(r+2, coll))

    for tup in addresses:

        try:

            if f2d[tup[0]][tup[1]][0]=="horse" and colour(f2d[tup[0]][tup[1]])==clr:

                attack+=(f2d[tup[0]][tup[1]],)

        except IndexError:

            pass

    if attack==():

        return (False, attack)

    else:

        return (True, attack)

#Aim: Detecting a Checkmate

#Parameters: Address of the king for whom you would like to validate for checkmate(tuple), 2 dimensional list

#Return: True if it is a checkmate and return False if it is not a checkmate.

def checkmate(add, f2d):

    attack=check(add,f2d)[1]

    if attack==():

        return False

    logmessage.log("\nCHECK!")

    logmessage.log("Pieces attacking the king:", attack)

    oppclr=oppcolour(colour(f2d[add[0]][add[1]]))  #The colour opposite to the king's colour which the king can attack

    kingclr=colour(f2d[add[0]][add[1]])

    logmessage.log("Attacked King colour: ", kingclr)

    logmessage.log("Opposite King Colour: ", oppclr)

    #Checking whether the king can move to a SAFE square around it either by just moving or by attacking another piece

    r,c=add[0], add[1]

    ru=8

    cl=8

    if r-1>=0:

        ru=r-1

    if c-1>=0:

        cl=c-1

    addresses=((r,c+1), (ru, c+1) ,(ru, c) ,(ru, cl) ,(r, cl) ,(r+1, cl) ,(r+1, c) ,(r+1, c+1))

    logmessage.log(addresses)

    for tup in addresses:

        testf2d=[]

        for i in f2d:

            temp=[]

            for j in i:

                temp.append(j.copy())

            testf2d.append(temp)

        try:

            if colour(testf2d[tup[0]][tup[1]])=="empty" or colour(testf2d[tup[0]][tup[1]])==oppclr:

                validatemove((r,c), tup, testf2d)

                logmessage.log("Pieces which can capture the king if it moves to the given adjacent square: ", tup, capture((tup[0], tup[1]), False, oppclr, testf2d))

                capturecheck=True

                if kingclr=="white":

                    #rok and cok-row of opposite king and column of opposite king

                    for i in testf2d:

                        for j in i:

                            if j[1]=="$":

                                rok, cok=j[2], j[3]

                    if (rok-tup[0])\*\*2 + (cok-tup[1])\*\*2<=2:

                        capturecheck=False

                elif kingclr=="black":

                    #rok and cok-row of opposite king and column of opposite king

                    for i in testf2d:

                        for j in i:

                            if j[1]=="\*$":

                                rok, cok=j[2], j[3]

                    if (rok-tup[0])\*\*2 + (cok-tup[1])\*\*2<=2:

                        capturecheck=False

                if capturecheck==True and capture((tup[0], tup[1]), False, oppclr, testf2d)[0]==False:

                    return False

        except IndexError:

            pass

    if len(attack)>=2:

        #If there are multiple pieces attacking the king, the king has no option but to move simply or by attacking a piece.(This has already been validated in the beginning of this function)

        pass

    else:

        if attack[0][0]=="horse":

            #If there is a horse attacking a king, the king has no option but to move simply or by attacking a piece. Or the horse which is attacking can be killed.(This has already been validated in the beginning of this function)

            if capture((attack[0][2], attack[0][3]), False, kingclr,f2d)[0]==True:

                for i in capture((attack[0][2], attack[0][3]), False, kingclr,f2d)[1]:

                    #duplicate f2

                    dupf2d=[]

                    for k in f2d:

                        temp=[]

                        for j in k:

                            temp.append(j.copy())

                        dupf2d.append(temp)

                    validatemove((i[2], i[3]),(attack[0][2],attack[0][3]), dupf2d)

                    if check((r,c), dupf2d)[0]==False:

                        return False

        else:

            #If only one piece is attacking a king, the king can move, attack or some piece from the king's team can come in between and block OR some piece of the king's team can kill the piece giving a check

            rowf=attack[0][2] #Row number of the attacking piece

            colf=attack[0][3] #Column number of the attacking piece

            rowstep, colstep=stepcount((r,c), (rowf, colf)) #Gives us the rowstep value and the colstep value to gradually move from the king's address to the attackers's address

            rowcounter=r+rowstep

            colcounter=c+colstep

            success=False

            while success==False:

                if rowcounter==rowf and colcounter==colf:

                    success=True

                    if capture((rowcounter, colcounter), False, kingclr, f2d)[0]==True:

                        logmessage.log("Pieces which can attack the attacking piece: ",(rowcounter, colcounter), capture((rowcounter, colcounter), False, kingclr, f2d))

                        for i in capture((rowcounter, colcounter), False, kingclr, f2d)[1]:

                            #duplicate f2d

                            dupf2d=[]

                            for k in f2d:

                                temp=[]

                                for j in k:

                                    temp.append(j.copy())

                                dupf2d.append(temp)

                            validatemove((i[2], i[3]),(rowf,colf), dupf2d)

                            if check((r,c), dupf2d)[0]==False:

                                return False

                else:

                    if capture((rowcounter, colcounter), True, kingclr, f2d)[0]==True:

                        logmessage.log("Pieces which can block the attacking piece: ",(rowcounter, colcounter) ,capture((rowcounter, colcounter), True, kingclr, f2d))

                        for i in capture((rowcounter, colcounter), True, kingclr, f2d)[1]:

                            #duplicate f2d

                            dupf2d=[]

                            for k in f2d:

                                temp=[]

                                for j in k:

                                    temp.append(j.copy())

                                dupf2d.append(temp)

                            validatemove((i[2], i[3]),(rowcounter,colcounter), dupf2d)

                            if check((r,c), dupf2d)[0]==False:

                                return False

                rowcounter+=rowstep

                colcounter+=colstep

    return True

#Aim: Detecting a Stalemate

#Parameters: 2 dimensional list, Is any king under check?

#Return: True if it is a stalemate and return False if it is not a stalemate.

def stalemate(f2d, checkval):

    #Stalemate Checking

    if checkval==True:

        return False

    for i in f2d:

        for j in i:

            if chance==colour(j) and legal((j[2], j[3]), f2d)[0]==True:

                return False

    return True

#The main function which is called by the display subsection of the program. This functions unifies all the functions present in Logic.py

#Return a long tuple containing all the information which the display subsection of the program would need.

#Returns: (Validity of Move, starting address, ending address, is it a check?, is it a checkmate?, is pawnpromotion possible? , Is castling possible?, are any legal moves possible for the piece present in the starting address? ,Possible legal addresses as a tuple)

def gameprocessing(add1, add2, update=False): #Starting address, Ending address

    #list2d ==> Live Board

    #newlist2d ==> To be used for checks and checkmates

    global chance, list2d, list2d\_start\_of\_current\_move, enpassant\_possible, enpassant\_possiblevictim

    enpassantcapture=False #WIll be changed to True if a pawn is en passant capturing another pawn in this move. False is a default value

    enpassant\_killedpawn\_add=None

    if add2==(None, None):

        return (False, add1, add2, False, False, False, False, legal(add1, list2d)[0],legal(add1, list2d)[1], False, None, False)

    if add1==add2:

        return (False, add1, add2, False, False, False, False, legal(add1, list2d)[0],legal(add1, list2d)[1], False, None, False)

    newlist2d=[]

    for i in list2d:

        temp=[]

        for j in i:

            temp.append(j.copy())

        newlist2d.append(temp)

    list2d\_start\_of\_current\_move = []

    for i in list2d:

        temp=[]

        for j in i:

            temp.append(j.copy())

        list2d\_start\_of\_current\_move.append(temp)

    pc=list2d[add1[0]][add1[1]] #piece - pc

    logmessage.log("")

    logmessage.log("Chance: ", chance)

    level1=False

    if colour(pc)==chance:

        level1=True

    logmessage.log("Level1: (The correct player is playing) ", level1)

    level2=False

    if level1==True:

        if pc[0]=="rook" and rookmove(add1, add2, list2d)==True:

            level2=True

        elif pc[0]=="horse" and hormove(add1, add2, list2d)==True:

            level2=True

        elif pc[0]=="bishop" and bishmove(add1, add2, list2d)==True:

            level2=True

        elif pc[0]=="queen" and queenmove(add1, add2, list2d)==True:

            level2=True

        elif pc[0]=="king" and kingmove(add1, add2, list2d)==True:

            level2=True

        elif pc[0]=="pawn" and pawnmove(add1, add2, list2d)[0]==True:

            enpassantcapture=pawnmove(add1, add2, list2d)[2]

            enpassant\_killedpawn\_add=pawnmove(add1, add2, list2d)[3]

            level2=True

    if level1==True:

        logmessage.log("Level2: (The selected piece is CAPABLE of moving to the specified location) ", level2)

    level2alter=False #Alternate level2 incorporating castling

    if level1==True:

        if level2 == False:

            try:

                castleinfo=castle(add1, add2, list2d)

                if castleinfo[0]==True:

                    level2alter=True

            except:

                pass

    if level2==True:

        logmessage.log("Level2alter: (Alternate Level2) (Castling is Possible) ", level2alter)

    #Changing newlist2d to use it for check and checkmate

    if level2==True:

        validatemove(add1, add2, newlist2d)

    elif level2alter==True:

        #Converting the end row, col click to rook square row, col

        if add2[1] > add1[1]:

            add2 = (add2[0], add2[1] + 1)

        elif add2[1] < add1[1]:

            add2 = (add2[0], add2[1] - 2)

        validatemove(castleinfo[1][0], castleinfo[1][1], newlist2d)

        validatemove(castleinfo[2][0], castleinfo[2][1], newlist2d)

    for i in newlist2d:

        for j in i:

            if j[1]=="\*$":

                newwhking=j

            elif j[1]=="$":

                newblking=j

    level3=False

    if level2==True or level2alter==True:

        if chance=="white":

            if check((newwhking[2], newwhking[3]),newlist2d)[0]==False:

                level3=True

        if chance=="black":

            if check((newblking[2], newblking[3]), newlist2d)[0]==False:

                level3=True

    if level2alter==True:

        logmessage.log("Level3: (Player's king is safe after the move) ", level3)

    level4=False

    if level3==True:

        if ((newwhking[2]-newblking[2])\*\*2 + (newwhking[3]-newblking[3])\*\*2 >2):

            level4=True

    if level3==True:

        logmessage.log("Level4: (Kings are NOT too close) ", level4)

    if level4==True:

        #Incorporating En Passant: Making changes to the pawn's detail list and changing the value of enpassant\_possible

        if enpassant\_possible==True: #This statement can be passed only if pawnmove(add1, add2, list2d)[1] was passed in the previous move

            enpassant\_possible=False

            enpassant\_possiblevictim[4]=False

            enpassant\_possiblevictim=[None, None]

            del enpassant\_possiblevictim

        if pawnmove(add1, add2, list2d)[1]==True:

            pc[4]=True

            enpassant\_possiblevictim=pc #The pawn which is susceptible to an enpassant capture in the next move

            enpassant\_possible=True #An en passant capture is possible in the next move

        if level2==True:

            if pc[0]=="king": #Indicating that the king has started moving

                pc[4]=True

            elif pc[0]=="rook":# Indicating that the rook has started moving

                pc[4]=True

            piece\_at\_endinglocation=list2d[add2[0]][add2[1]]

            move(add1, add2)

            if enpassantcapture==True:

                piece\_at\_victimlocation = list2d[enpassant\_killedpawn\_add[0]][enpassant\_killedpawn\_add[1]]

                list2d[enpassant\_killedpawn\_add[0]][enpassant\_killedpawn\_add[1]]=emp

        elif level2alter==True:

            list2d[add1[0]][add1[1]][4]=True

            list2d[add2[0]][add2[1]][4]=True

            move(castleinfo[1][0], castleinfo[1][1])

            move(castleinfo[2][0], castleinfo[2][1])

        #Checking for Pawn Promotion: The move has already been validated

        if chance=="white":

            pawnprom=False

            if pc[0]=="pawn" and pc[2]==0:

                pawnprom=True

        if chance=="black":

            pawnprom=False

            if pc[0]=="pawn" and pc[2]==7:

                pawnprom=True

        if chance=="white":

            checkval=check((blking[2], blking[3]), list2d)[0]

            checkmateval=checkmate((blking[2], blking[3]), list2d)

            chance="black"

        elif chance=="black":

            checkval=check((whking[2], whking[3]), list2d)[0]

            checkmateval=checkmate((whking[2], whking[3]), list2d)

            chance="white"

        logmessage.log("\n")

        logmessage.log(list2d, "\n")

        if logmessage.DEBUG==True:

            displaylist(list2d)

            for i in list2d:

                for j in i:

                    print(j, end=" ")

                print("\n\n")

        stalemateval = stalemate(list2d, checkval)

        #Adding the move made to "List\_of\_Moves"

        if level2alter==True: #Castling

            List\_of\_Moves.append(("Castling", castleinfo[1], castleinfo[2]))

        elif enpassantcapture==True: #Enpassant capture

            List\_of\_Moves.append(("Enpassant", add1, add2, enpassant\_killedpawn\_add, piece\_at\_victimlocation))

        else:

            List\_of\_Moves.append((add1, add2, piece\_at\_endinglocation))

        #print('in logic: ',List\_of\_Moves , '\n')

        if level2alter==True:#Level2alter is for castling

            return (True, castleinfo[1], castleinfo[2], checkval, checkmateval, pawnprom, castleinfo[0], legal(add1, list2d)[0],legal(add1, list2d)[1], False, None, stalemateval)

        elif level2==True:

            return (True, add1, add2, checkval, checkmateval, pawnprom, False, legal(add1, list2d)[0],legal(add1, list2d)[1], enpassantcapture, enpassant\_killedpawn\_add, stalemateval)

    elif level4==False:

        return(False, add1, add2, False, False, False, False, legal(add1, list2d)[0], legal(add1, list2d)[1], False, None, False)

#Given the address where the pawn is present and the identity of the piece to which it has to be converted to, the required changes in list2d are made.

def pawnpromotion(add, identity):#Address of the pawn as a tuple, identity-what should the pawn be changed to? (queen, rook, horse, bishop)

    global list2d

    pc=list2d[add[0]][add[1]]

    if identity==None:

        return None

    if colour(pc)=="white":

        if identity=="queen":

            sym="\*Q"

        elif identity=="bishop":

            sym="\*A"

        elif identity=="horse":

            sym="\*/>"

        elif identity=="rook":

            sym="\*]["

    elif colour(pc)=="black":

        if identity=="queen":

            sym="Q"

        elif identity=="bishop":

            sym="A"

        elif identity=="horse":

            sym="/>"

        elif identity=="rook":

            sym="]["

    if identity=="rook":

        globals()['{clr[0:2]}{identity}pp{pawnnum}']=[identity, sym, pc[2], pc[3], True]

        List\_of\_Moves.append(("Pawnpromotion", add, [identity, sym, pc[2], pc[3], True]))

    else:

        globals()['{clr[0:2]}{identity}pp{pawnnum}']=[identity, sym, pc[2], pc[3]]

        List\_of\_Moves.append(("Pawnpromotion", add, [identity, sym, pc[2], pc[3]]))

    list2d[add[0]][add[1]]=globals()['{clr[0:2]}{identity}pp{pawnnum}']

#Parameter: Possible winner

#Return: 0 or 1

def winner\_on\_flag(clr):

    alive\_pieces = []

    for row in list2d:

        for piece in row:

            if colour(piece) == clr:

                alive\_pieces.append(piece[0])

    if len(alive\_pieces) == 1 and alive\_pieces == ['king']:

        return 0

    elif len(alive\_pieces) == 2:

        alive\_pieces.remove('king')

        if alive\_pieces[0] in ('bishop', 'horse'):

            return 0

        else:

            return 1

    else:

        return 1

def get\_game\_situation(f2d):

    checkv, checkmatev, stalematev = False, False, False

    for row in f2d:

        for piece in row:

            if piece[1]=="\*$":

                whkadd = tuple(piece[2:4])

                checkv = check(whkadd, f2d)[0]

                if checkv:

                    checkmatev = checkmate(whkadd, f2d)

                    if not checkv:

                        stalematev = stalemate(f2d, False)

                    return checkv , checkmatev , stalematev

            elif piece[1]=="$":

                blkadd = tuple(piece[2:4])

                checkv = check(blkadd, f2d)[0]

                if checkv:

                    checkmatev = checkmate(blkadd, f2d)

                    if not checkv:

                        stalematev = stalemate(f2d, False)

                    return checkv , checkmatev , stalematev

    return checkv, checkmatev, stalematev

database\_functions.py

import mysql.connector

import configparser

from tkinter import messagebox

#mysql -h mydbinstance.cm4fylzmmaab.us-east-2.rds.amazonaws.com  -u admin -p

def open\_connection():

    global chessdb, mycur

    try:

        config = configparser.ConfigParser()

        config.read('config.ini')

        host\_val = config['DATABASE DETAILS']['HOST']

        user\_val = config['DATABASE DETAILS']['USER']

        passwd\_val = config['DATABASE DETAILS']['PASSWORD']

        db\_val = config['DATABASE DETAILS']['DATABASE']

        #chessdb=mysql.connector.connect(host="localhost", user="root", passwd="sri21sql04#$", database="chessarena")

        chessdb=mysql.connector.connect(host=host\_val, user=user\_val, passwd=passwd\_val, database=db\_val)

        mycur=chessdb.cursor()

    except:

        pass

def update\_game\_details(date,start\_time, end\_time, duration, white\_player, black\_player, winner, List\_of\_Moves, List\_of\_Times, min\_time, increment):

    try:

        #Updating the TABLE Games\_Played

        mycur.execute("INSERT INTO Games\_Played VALUES(NULL,'" + str(date) + "','" + str(start\_time) + "','" + str(end\_time) + "','" + str(duration) + "','" + white\_player + "','" + black\_player + "','" + winner + "','" + str(min\_time) + "','" + str(increment) + "')")

        #Getting the Match number

        mycur.execute("SELECT Match\_Number from Games\_Played")

        mat\_no=mycur.fetchall()[-1][0]

        #Updating the TABLE Moves\_Made

        for i,j in zip(List\_of\_Moves, List\_of\_Times):

            mycur.execute("INSERT INTO Moves\_Made VALUES('" + str(mat\_no) + "',\"" + str(i) + "\",\""+ str(j) + "\")")

        chessdb.commit()

    except:

        pass

def receive\_game\_details(mat\_no):

    try:

        #Receiving all the details from the database for the match number given as parameter

        mycur.execute("SELECT \* from Games\_Played where match\_number="+str(mat\_no))

        mat\_no,date,start\_time, end\_time, duration, white\_player, black\_player, winner, min\_time, increment = mycur.fetchall()[0]

        mycur.execute("SELECT Move from Moves\_Made where match\_number="+str(mat\_no))

        moves=[]

        for i in mycur.fetchall():

            moves.append(eval(i[0]))

        mycur.execute("SELECT Time\_Taken from Moves\_Made where match\_number="+str(mat\_no))

        times=[]

        for i in mycur.fetchall():

            times.append(i[0])

        return date,start\_time, end\_time, duration, white\_player, black\_player, winner, moves, times, min\_time, increment

    except:

        pass

def receive\_all\_game\_details():

    try:

        mycur.execute("SELECT \* from Games\_Played")

        return mycur.fetchall()[0:]

    except:

        pass

def update\_configuration\_saved(mat\_no, config\_no, title, notes):

    try:

        mycur.execute("DELETE FROM Configurations\_Saved WHERE Match\_Number = "+str(mat\_no)+" and Config\_No ="+str(config\_no))

        mycur.execute("INSERT INTO Configurations\_Saved VALUES("+str(mat\_no)+","+str(config\_no)+",\""+title.rstrip()+"\",\""+notes.rstrip()+"\")")

        chessdb.commit()

    except:

        pass

def delete\_configuration(mat\_no, config\_no):

    try:

        mycur.execute("DELETE FROM Configurations\_Saved WHERE Match\_Number = "+str(mat\_no)+" and Config\_No ="+str(config\_no))

        chessdb.commit()

    except:

        pass

def receive\_configurations\_saved(mat\_no):

    try:

        mycur.execute("SELECT \* from Configurations\_Saved WHERE Match\_Number = "+ str(mat\_no))

        return mycur.fetchall()

    except mysql.connector.Error as err:

        if err.errno in (2006, 2013):

            pass

        else:

            messagebox.showerror("Database Error", err)

            return "Unknown Error"

    except:

        pass

def close\_connection():

    try:

        mycur.close()

        chessdb.close()

    except:

        pass

def check\_connection():

    \_ = receive\_configurations\_saved(0) #This has been given only to check if there is a connection. Match Numbers start from 1.

    if \_ == None:

        return False

    else:

        return True

utils.py

#Import of Modules

from logmessage import log

from math import floor

from tkinter import \*

from random import randint

from playsound import playsound

from colorsys import rgb\_to\_hls, hls\_to\_rgb

#Import of created files

import game

import logic

import constant

class Tooltip:

    def \_\_init\_\_(self, widget, \*, bg='#18191C', fg = "#FFFFFF", pad=(8, 6, 6, 5), text='widget info', waittime=400, wraplength=250):

        self.waittime = waittime  # in miliseconds, originally 500

        self.wraplength = wraplength  # in pixels, originally 180

        self.widget = widget

        self.text = text

        self.widget.bind("<Enter>", self.onEnter)

        self.widget.bind("<Leave>", self.onLeave)

        self.widget.bind("<ButtonPress>", self.onLeave)

        self.bg = bg

        self.fg = fg

        self.pad = pad

        self.id = None

        self.tw = None

    def onEnter(self, event=None):

        self.schedule()

    def onLeave(self, event=None):

        self.unschedule()

        self.hide()

    def schedule(self):

        self.unschedule()

        self.id = self.widget.after(self.waittime, self.show)

    def unschedule(self):

        id\_ = self.id

        self.id = None

        if id\_:

            self.widget.after\_cancel(id\_)

    def show(self):

        def tip\_pos\_calculator(widget, label, \*, tip\_delta=(10, 5), pad=(5, 3, 5, 3)):

            w = widget

            s\_width, s\_height = w.winfo\_screenwidth(), w.winfo\_screenheight()

            width, height = (pad[0] + label.winfo\_reqwidth() + pad[2],

                             pad[1] + label.winfo\_reqheight() + pad[3])

            mouse\_x, mouse\_y = w.winfo\_pointerxy()

            x1, y1 = mouse\_x + tip\_delta[0], mouse\_y + tip\_delta[1]

            x2, y2 = x1 + width, y1 + height

            x\_delta = x2 - s\_width

            if x\_delta < 0:

                x\_delta = 0

            y\_delta = y2 - s\_height

            if y\_delta < 0:

                y\_delta = 0

            offscreen = (x\_delta, y\_delta) != (0, 0)

            if offscreen:

                if x\_delta:

                    x1 = mouse\_x - tip\_delta[0] - width

                if y\_delta:

                    y1 = mouse\_y - tip\_delta[1] - height

            offscreen\_again = y1 < 0  # out on the top

            if offscreen\_again:

                # No further checks will be done.

                # TIP:

                # A further mod might automagically augment the

                # wraplength when the tooltip is too high to be

                # kept inside the screen.

                y1 = 0

            return x1, y1

        fg = self.fg

        bg = self.bg

        pad = self.pad

        widget = self.widget

        # creates a toplevel window

        self.tw = Toplevel(widget)

        self.root = self.tw.winfo\_toplevel()

        self.root.attributes('-topmost', True)

        self.root.attributes('-alpha', 0.9)

        # Leaves only the label and removes the app window

        self.tw.wm\_overrideredirect(True)

        win = Frame(self.tw,

                       background=bg,

                       borderwidth=0)

        label = Label(win,

                          text=self.text,

                          justify=LEFT,

                          font = ('Comic Sans', 13, 'bold'),

                          background=bg,

                          foreground=fg,

                          relief=SOLID,

                          borderwidth=0,

                          wraplength=self.wraplength)

        label.grid(padx=(pad[0], pad[2]),

                   pady=(pad[1], pad[3]),

                   sticky=NSEW)

        win.grid()

        x, y = tip\_pos\_calculator(widget, label)

        self.tw.wm\_geometry("+%d+%d" % (x, y))

    def hide(self):

        tw = self.tw

        if tw:

            tw.destroy()

        self.tw = None

class CustomButton(Button):

    def \_\_init\_\_(self, master, hover = False, hover\_text = '', hover\_bg = "#000000", hover\_fg = "#FFFFFF", \*args, \*\*kwargs):

        super().\_\_init\_\_(master, \*args, \*\*kwargs, activebackground = constant.ACTIVEBGCLR, activeforeground = constant.ACTIVEFGCLR)

        if hover:

            self.bind("<Enter>", lambda e: self.onHoverShowButton())

            self.hover\_text = hover\_text

            self.hover\_bg = hover\_bg

            self.hover\_fg = hover\_fg

        else:

            self.bind("<Enter>", lambda e: self.onEnter())

    def onEnter(self):

        org\_bg = self['bg']

        conv\_org\_bg = tuple(int(org\_bg.lstrip("#")[i:i+2], 16) for i in (0, 2, 4))

        new\_bg = '#%02x%02x%02x' % darken\_color(\*conv\_org\_bg, 0.2)

        self.config(bg = new\_bg)

        def onLeave(e):

            if new\_bg == self['bg']:

                self.config(bg = org\_bg)

            self.unbind("<Leave>")

        self.bind("<Leave>", onLeave)

    def onHoverShowButton(self):

        org\_bg = self['bg']

        new\_bg = self.hover\_bg

        org\_fg = self['fg']

        new\_fg = self.hover\_fg

        org\_text = self['text']

        def onLeave(e):

            self.config(text = org\_text, bg = org\_bg, fg = org\_fg)

            self.unbind("<Leave>")

        self.config(text = self.hover\_text, bg = new\_bg, fg = new\_fg)

        self.bind("<Leave>", onLeave)

class ScrolledFrame(Frame):

    def \_\_init\_\_(self, parent, max\_height, \*args, \*\*kwargs):

        super().\_\_init\_\_(parent, \*args, \*\*kwargs) # create a frame (self)

        #Storing the value of max\_height

        self.max\_height = max\_height

        #place canvas on self

        self.canvas = Canvas(self, \*args, \*\*kwargs)

        #place a frame on the canvas, this frame will hold the child widgets

        self.viewPort = Frame(self.canvas, \*args, \*\*kwargs)

        #place a vertical scrollbar on self

        self.vsb = Scrollbar(self, orient= VERTICAL)

        #Setting the command for vertical scrollbar

        self.vsb.config(command = self.canvas.yview)

        #pack the vertical scrollbar to right of self

        self.vsb.pack(side = RIGHT, fill = Y)

        #attach scrollbar action to scroll of canvas

        self.canvas.configure(yscrollcommand = self.vsb.set)

        #pack canvas to left of self and expand to fil

        self.canvas.pack(side = TOP, fill = 'both', expand = True)

        self.canvas\_window = self.canvas.create\_window((4,4), window=self.viewPort, anchor="nw")

        #bind an event whenever the size of the viewPort frame changes.

        self.viewPort.bind("<Configure>", self.onFrameConfigure)

        #bind an event whenever the size of the viewPort frame changes.

        self.canvas.bind("<Configure>", self.onCanvasConfigure)

        #perform an initial stretch on render, otherwise the scroll region has a tiny border until the first resize

        self.onFrameConfigure(None)

        self.viewPort.bind('<Enter>', self.\_bound\_to\_mousewheel)

        self.viewPort.bind('<Leave>', self.\_unbound\_to\_mousewheel)

    def \_bound\_to\_mousewheel(self, event):

        self.canvas.bind\_all("<MouseWheel>", self.\_on\_mousewheel)

    def \_unbound\_to\_mousewheel(self, event):

        self.canvas.unbind\_all("<MouseWheel>")

    def \_on\_mousewheel(self, event):

        try:

            self.canvas.yview\_scroll(int(-1\*(event.delta/120)), "units")

        except:

            pass

    def onFrameConfigure(self, event):

        viewPort\_ht = self.viewPort.winfo\_height()

        if viewPort\_ht < self.max\_height:

            self.canvas.config(height = viewPort\_ht)

        else:

            self.canvas.config(height = self.max\_height)

        #whenever the size of the frame changes, alter the scroll region respectively.

        self.canvas.configure(scrollregion=self.canvas.bbox("all"))

    def onCanvasConfigure(self, event):

        '''Reset the canvas window to encompass inner frame when required'''

        canvas\_width = event.width

        #whenever the size of the canvas changes alter the window region respectively.

        self.canvas.itemconfig(self.canvas\_window, width = canvas\_width)

    def config(self, \*\*kwargs):

        self.viewPort.config(\*\*kwargs)

        self.canvas.config(\*\*kwargs)

        self.viewPort.update\_idletasks()

        self.canvas.update\_idletasks()

    def configure(self, \*\*kwargs):

        self.viewPort.config(\*\*kwargs)

        self.canvas.config(\*\*kwargs)

        self.viewPort.update\_idletasks()

        self.canvas.update\_idletasks()

#time format, n in seconds

def tfor(n):

    s = n%60

    m = int((n//60)%60)

    h = int((n//60)//60)

    sec = "0"\*(2-len(str(int(round(s,1))))) + str(float(round(s,1)))

    time = f'{"0"\*(2-len(str(h)))+str(h)} : {"0"\*(2-len(str(m)))+str(m)} : {sec}'

    return time

#Parameters: Exact Pixels on the screen

#Return: Assuming the board to just be a MATRIX, it returns the row num and the col num (Doesn't care about WV and BV.)

def get(x,y):

    return floor(3 - y//game.sqsize), floor(x//game.sqsize + 4)

#Return: central pixel coordinates as a tuple

def antiget(row,col): #index2d ==> 2dindex tuple, l-side length of the board in pixels

    step=game.size/8

    return ((col\*step)-(4\*step)+(step/2),(4\*step)-(row\*step)-(step/2))

#Parameters: WV-Row num,  WV-Col num

#Return: Chess Naming for the square (Eg: A6, E4 etc)

def info(row, col):

    return chr(col + 65) + str(8-row)

#Converting x coordinate to centre of the board

#While Processing, it is easier deal with the board as if it was actaully in the centre. Now, x and y work like as though the board is at the centre

def shift\_to\_centre(x):

    if game.white==True:

        x = x - game.drift

    elif game.black==True:

        x = x + game.drift

    else:

        x=0

    return x

def whiteview\_blackview\_equivalents(row,col):

    if game.white==True:

        whrow=row

        whcol=col

        blrow, blcol=7-row, 7-col

    elif game.black==True:

        blrow, blcol = row, col

        whrow=7-row

        whcol=7-col

    else:

        whrow, whcol, blrow, blcol=0,0,0,0

    return whrow, whcol, blrow, blcol

#Aim: Speak out every move

#Parameters: Starting row, Starting column, Ending row, Ending column (All these values are wrt white view)

def speak(srow, scol, erow, ecol, enpassant, castle):

    piece = game.name[logic.list2d\_start\_of\_current\_move[srow][scol][0]]

    start = info(srow, scol)

    end = info(erow, ecol)

    try:

        victim = game.name[logic.list2d\_start\_of\_current\_move[erow][ecol][0]]

        partner = game.name[logic.list2d\_start\_of\_current\_move[erow][ecol][0]]

    except:

        pass

    if enpassant == True:

        game.speaker.say(f'{piece} {start} to {end} en passant')

    elif castle == True:

        game.speaker.say(f'{piece} {start} {partner} {end} castle')

    elif logic.list2d\_start\_of\_current\_move[erow][ecol] != logic.emp:

        game.speaker.say(f'{piece} {start} takes {victim} {end}')

    else:

        game.speaker.say(f'{piece} {start} to {end}')

    game.speaker.runAndWait()

def find(pinfo, f2d\_start\_of\_current\_move):

    l = f2d\_start\_of\_current\_move

    altpieces = []

    for y in range(8):

        for x in range(8):

            if l[y][x] == pinfo:

                continue

            elif l[y][x][1] == pinfo[1]:

                altpieces += [l[y][x]]

    return altpieces

def get\_chess\_notation(f2d\_start\_of\_current\_move, srow,scol,erow,ecol,checkinfo,castle,pppiece=''):

    check = '+' if checkinfo[0] and not checkinfo[1] else ''

    checkmate = '#' if checkinfo[1] else ''

    check\_str = check + checkmate

    if castle:

        return ('O-O' if ecol > scol else 'O-O-O') + check\_str

    pinfo = f2d\_start\_of\_current\_move[srow][scol]

    piece = pinfo[0][0].upper() if pinfo[0] != 'pawn' else 'abcdefgh'[scol]

    piece = 'N' if piece == 'H' else piece

    process = '' if not f2d\_start\_of\_current\_move[erow][ecol][0] else 'x'

    suffix = f'{"abcdefgh"[ecol]}{8-erow}'

    if piece in ('Q','N','B','R'):

        altpieces = find(pinfo, f2d\_start\_of\_current\_move)

        if altpieces:

            filtered\_altpieces = []

            for r in range(len(altpieces)):

                otherp = altpieces[r]

                sr,sc = otherp[2],otherp[3]

                possible = logic.legal((sr,sc),f2d\_start\_of\_current\_move)[1]

                if (erow,ecol) in possible:

                    filtered\_altpieces += [altpieces[r]]

            altpieces = filtered\_altpieces

            row = col = False

            prefix = ''

            for opiece in altpieces:

                prefix = f'{piece}{"abcdefgh"[scol]}'

                col = True if opiece[3] == ecol else col

                row = True if opiece[2] == erow else row

                if row or col:

                    prefix = ''

                if row and col:

                    break

            if not prefix:

                prefix = f'{piece}{"abcdefgh"[scol]}' if row else piece

                prefix = f'{piece}{8-srow}' if col else prefix

                prefix = f'{piece}{"abcdefgh"[scol]}{8-srow}' if row and col else prefix

        else:

            prefix = piece

    else:

        if piece.lower() == piece: #Pawn

            prefix = piece if scol != ecol  else ''

        else: #King

            prefix = piece

    if pinfo[0] == "pawn" and scol != ecol:

        if pppiece:

            return f'{prefix}x{suffix}={pppiece[0].upper()}' + check\_str

        else:

            return f'{prefix}x{suffix}' + check\_str

    if pppiece:

        return f'{prefix}{process}{suffix}={pppiece[0].upper()}' + check\_str

    return f'{prefix}{process}{suffix}' + check\_str

#Aim: Play sound effects for movement, captures and check

#Parameters: Ending row, Ending column, enpassant capture?, castling?, check?, checkmate?, stalemate?

def sounds(erow, ecol, enpassant, castle,  check, checkmate, stalemate):

    if check!= True and checkmate!=True and stalemate!=True:

        if castle==True:

            playsound("./Sounds/castling.mp3", False)

        elif enpassant==True:

            playsound("./Sounds/capture.mp3", False)

        else:

            if logic.list2d\_start\_of\_current\_move[erow][ecol] == logic.emp:

                playsound("./Sounds/move.mp3", False)

            elif logic.list2d\_start\_of\_current\_move[erow][ecol] != logic.emp:

                playsound("./Sounds/capture.mp3", False)

    elif check == True and checkmate != True:

        playsound("./Sounds/check.mp3", False)

def adjust\_color\_lightness(r, g, b, factor):

    h, l, s = rgb\_to\_hls(r / 255.0, g / 255.0, b / 255.0)

    l = max(min(l \* factor, 1.0), 0.0)

    r, g, b = hls\_to\_rgb(h, l, s)

    return int(r \* 255), int(g \* 255), int(b \* 255)

def lighten\_color(r, g, b, factor=0.1):

    r = int(r)

    g = int(g)

    b = int(b)

    return adjust\_color\_lightness(r, g, b, 1 + factor)

def darken\_color(r, g, b, factor=0.1):

    return adjust\_color\_lightness(r, g, b, 1 - factor)

def hex\_to\_rgb(value):

    value = value.lstrip('#')

    length = len(value)

    return tuple(int(value[i:i + length // 3], 16) for i in range(0, length, length // 3))

def rgb\_to\_hex(rgb):

    rgb = tuple(int(i) for i in rgb)

    return '#%02x%02x%02x' % rgb

constant.py

#Creating Chess Pieces

pcdriftx={"rook": -56, "horse":-48, "bishop": -50, "queen": -48, "king":-48, "pawn":-45}

pcdrifty={"rook": -55, "horse":-40, "bishop": -47, "queen": -45, "king":-42, "pawn":-42}

tiltedcoord={"rook": (), "horse": (), "bishop": (), "queen": (), "king": (), "pawn": ()}

COORD={"rook": (), "horse": (), "bishop": (), "queen": (), "king": (), "pawn": ()}

pieces=("rook", "horse", "bishop", "queen", "king", "pawn")

tiltedcoord["rook"]=((96,83),(86,83),(78,69),(43,69),(40,75),(37,77),(20,77),(20,67),(28,67),(28,59),(17,59),(17,44),(28,44),(28,36),(20,36),(20,26),(37,26),(40,29),(43,34),(78,34),(86,20),(96,20))

tiltedcoord["horse"]=((82,63),(74,63),(67,55),(64,55),(48,39),(43,39),(40,40),(42,41),(47,49),(47,53),(53,59),(53,64),(50,70),(44,70),(32,61),(27,61),(22,58),(17,51),(11,51),(11,47),(17,42),(18,35),(21,28),(27,22),(34,17),(41,15),(55,15),(67,17),(75,11),(82,11))

tiltedcoord["bishop"]=((84,72),(75,72),(69,61),(66,63),(61,66),(56,68),(50,68),(45,67),(39,65),(34,62),(24,53),(20,54),(16,54),(11,50),(11,45),(16,41),(20,44),(25,47),(34,49),(45,50),(50,50),(50,43),(45,43),(34,41),(27,38),(34,32),(39,29),(45,27),(50,26),(56,26),(61,28),(66,31),(69,33),(75,21),(84,21))

tiltedcoord["queen"]=((82,70),(73,70),(67,63),(39,74),(37,78),(33,80),(29,79),(26,75),(26,71),(29,67),(33,66),(37,66),(46,58),(25,58),(22,61),(18,63),(14,62),(11,58),(10,55),(11,52),(14,49),(18,48),(21,49),(24,51),(42,44),(24,39),(21,41),(18,42),(15,41),(11,38),(10,35),(10,34),(11,31),(15,28),(19,27),(23,28),(25,31),(46,31),(37,23),(33,24),(30,23),(27,21),(25,17),(26,13),(29,11),(33,10),(37,11),(39,15),(67,26),(73,19),(82,19))

tiltedcoord["king"]=((80,68),(72,68),(66,61),(48,76),(44,77),(44,62),(54,54),(54,49),(41,49),(38,53),(38,59),(43,62),(43,77),(36,77),(25,65),(25,56),(27,47),(21,47),(21,53),(13,53),(13,47),(6,47),(6,38),(13,38),(13,32),(21,32),(21,37),(27,37),(25,29),(25,18),(36,8),(43,8),(43,22),(38,26),(38,32),(41,36),(54,36),(54,31),(44,22),(44,8),(48,9),(66,24),(72,16),(80,16))

tiltedcoord["pawn"]=((77,64),(69,64),(51,47),(46,47),(46,53),(43,55),(40,54),(36,48),(33,50),(31,51),(28,52),(23,52),(20,50),(17,47),(15,44),(15,37),(17,34),(20,31),(23,29),(28,29),(31,30),(33,31),(36,33),(40,27),(43,26),(46,28),(46,34),(51,34),(69,17),(77,17))

for i in pieces:

    for tup in tiltedcoord[i]:

        COORD[i]+=((tup[0]+pcdriftx[i], tup[1]+pcdrifty[i]),)

#Colour Selections

LIGHTSQUARECLR="#E8E8E8"

DARKSQUARECLR="#78A7B7"

CHECKSQUARECLR=(224, 111, 111)

SELECTEDLIGHTSQUARECLR="#D8D14D"

SELECTEDDARKSQUARECLR="#C3BC3B"

LEGALLIGHTSQUARECLR="#B0B0D6"

LEGALDARKSQUARECLR="#8888D7"

BLACKPIECECLR='#313339'

WHITEPIECECLR="white"

PAWNPROMOTIONWINDOWCLR="#E2C35C"

#Colours specific to GUI

DARKBGCLR = "#1F1F1F" #

LIGHTBGCLR = "#CCD1D1" #ACACAC

DARKBGTEXTCLR = "#FFFFFF" #d9d9d9

LIGHTBGTEXTCLR = "#000000"

ACTIVEBGCLR = "#FFFFFF"

ACTIVEFGCLR = "#000000"

PRIMARYCLR = "#CCD1D1"

SECONDARYCLR = "#8BC34A"

TERTIARYCLR = "#FFA726"

#Drift of white\_view board in Double View

DRIFT\_WV\_BOARD = -300

DRIFT\_BOARD\_END\_OF\_GAME = -250

ABOUT\_INFO = '''Pocket Chess Arena, a cross platform desktop application, enables you to play casual unrated chess games with your friends offline and online and also revisit past games. Using the generated PGN, you can analyse your games on top chess platforms. The objective was to develop a pocket-sized application to help users quickly strike up a game with a friend in the midst of using another application.

It was built using tkinter, turtle and MySQL by 3 students of NPS Indiranagar, Bangalore as part of the Grade 11 and 12 Computer Science project.

Please feel free to reach out for any questions, comments or feedback.

Developers:

Sriram Srinivasan

Subham Patra

Gurumurthy V

Email address: pocketchessarena@gmail.com'''

spear.py

import flask

from threading import Thread

from random import \*

import time

server = flask.Flask(\_\_name\_\_)

rooms = {} #code: [code,me,move]

timers = {} #code: {(whitetime,running), (blacktime,running) ,mintime ,inc, movenumber}

inccheck = {} #code : [whiteinc bool , blackinc bool]

move\_durations = {} #code : move duration

offers = {} # code : drawoffers or gameover

connection = {}# code : [[white connection , black connection]]

debug = ''

def clean(code):

    try :

        del rooms[code] , timers[code] , inccheck[code] , move\_durations[code] , connection[code] , offers[code]

    except:

        pass

def generate\_code():

    number = list(str(randint(100,999)))

    characters = [choice([chr(x + 64) for x in range(1,27)]) for t in range(2)]

    l = number + characters

    shuffle(l)

    code = ''.join(l)

    if code in rooms:

        return generate\_code()

    return code

def handle\_room(code,timer=True):

    global oldb, oldw, connection

    t0 = int(time.time())

    while code in rooms:

        if timer:

            if timers[code]['black'][0] <= 0:

                timers[code]['black'][0] = 0

                offers[code] = {'status':'gameover', 'resignations':None, 'draw offers':None, 'flagged':'black', 'lost connection':None, 'game result':'W'}

                break

            elif timers[code]['white'][0] <= 0:

                timers[code]['white'][0] = 0

                offers[code] = {'status':'gameover', 'resignations':None, 'draw offers':None, 'flagged':'white', 'lost connection':None, 'game result':'B'}

                break

            if inccheck[code][0]:

                timers[code]['white'][0] += timers[code]['inc']

                inccheck[code][0] = False

            elif inccheck[code][1]:

                timers[code]['black'][0] += timers[code]['inc']

                inccheck[code][1] = False

            if timers[code]['white'][1]:

                timers[code]['white'][0] = oldw - time.time()

            elif timers[code]['black'][1]:

                timers[code]['black'][0] = oldb - time.time()

        if int(time.time())%30 == 15 and t0 != int(time.time()):

            t0 = int(time.time())

            if connection[code][0]-connection[code][1] >= 1:

                offers[code] = {'status':'gameover', 'resignations':None, 'draw offers':None, 'flagged':None, 'lost connection':'black', 'game result':'W'}

                break

            elif connection[code][1]-connection[code][0] >= 1:

                offers[code] = {'status':'gameover', 'resignations':None, 'draw offers':None, 'flagged':None, 'lost connection':'white', 'game result':'B'}

                break

@server.route('/connect',methods=['POST','GET'])

def on\_connect():

    return generate\_code()

@server.route(f'/rooms',methods=['POST','GET'])

def host():

    global rooms, oldb, oldw, timers , debug, connection

    if flask.request.method == 'POST':

        info = dict(flask.request.form)

        code = info['code']

        if 'status' in rooms:

            pass

        if code not in rooms:

            rooms[code] = info

            mintime = info['mintime']

            timer = False

            connection[code] = [0,0]

            offers[code] = {'status':'running', 'resignations':None, 'draw offers':None, 'flagged':None, 'lost connection':None, 'game result':None}

            if mintime.lower() != 'unlimited':

                mintime = int(mintime)\*60

                timers[code] = {'white' : [mintime , False] , 'black' : [mintime , False] , 'inc':int(info['inc']) , 'moveno':0 , 'mintime':mintime}

                inccheck[code] = [False, False]

                move\_durations[code] = []

                timer=True

            Thread(target=handle\_room,args=(code,timer)).start()

        else :

            rooms[code] = info

            try :

                if info['side'] == 'white':

                    oldb = timers[code]['black'][0] + time.time()

                    timers[code]['white'][1] = False

                    timers[code]['black'][1] = True

                    inccheck[code][0] = True

                    timers[code]['moveno'] += 1

                    move\_durations[code] += [int(timers[code]['mintime'] - timers[code]['white'][0] + timers[code]['inc']\*timers[code]['moveno'] - sum(move\_durations[code][:-1:2]))]

                elif info['side'] == 'black':

                    oldw = timers[code]['white'][0] + time.time()

                    timers[code]['black'][1] = False

                    timers[code]['white'][1] = True

                    inccheck[code][1] = True

                    move\_durations[code] += [int(timers[code]['mintime'] - timers[code]['black'][0] + timers[code]['inc']\*timers[code]['moveno'] - sum(move\_durations[code][1:-1:2]))]

            except Exception as er:

                pass

                #print(f'Some Error : {er}')

        return 'null'#flask.render\_template()

    else:

        code = flask.request.args.get('code')

        try:

            return rooms[code]

        except :

            return 'null'

@server.route('/connection',methods=['POST','GET'])

def on\_connection\_recv():

    global connection

    if flask.request.method == 'GET':

        side = flask.request.args.get('side')

        code = flask.request.args.get('code')

        if code is None:

            pass

        else:

            connectoption = (flask.request.args.get('connectop') == 'True')

            try :

                if connectoption:

                    if side == 'white':

                        connection[code][0] += 1

                    else:

                        connection[code][1] += 1

            except :

                pass

    elif flask.request.method == 'POST':

        info = dict(flask.request.form)

        code = info['code']

        if 'resign' in info:

            offers[code]['status'] = 'gameover'

            offers[code]['resignations'] = info['side']

        elif 'draw' in info:

            if not offers[code]:

                offers[code]['draw offers'] = info['side']

            else:

                offers[code]['draw offers'] = info['draw']

                if info['draw'] == 'accepted':

                    offers[code]['status'] = 'gameover'

                    offers[code]['game result'] = 'D'

        elif 'checkmate' in info:

            offers[code] = {'status':'gameover', 'resignations':None, 'draw offers':None, 'flagged':None, 'lost connection':None, 'game result':info['side'][0].upper()}

        elif 'reset' in info:

            offers[code] = {'status':'running', 'resignations':None, 'draw offers':None, 'flagged':None, 'lost connection':None, 'game result':None}

        elif 'clean' in info:

            clean(code)

    try :

        return {'connection':connection[code] , 'timers':timers[code] , 'offers':offers[code] , 'move durations':move\_durations[code]}

    except:

        return 'null'

@server.route('/buffer',methods=['POST','GET'])

def debug\_():

    global connection

    return connection

server.run(debug=1,host='localhost',port=7979)

test\_chess.py

#The purpose of test\_chess.py is to test multiple CHECKMATE configurations all at once using pytest.

import pytest

from logic import \*

iemp=["", "", 0, 0]

#The reason we use @pytest.fixture() is because we need to INITIALISE the 2dlist AND all the pieces every time we go to a new test case. In a new test case, we can't use the values of the 2dlist and the pieces which we finally arrived at in the previous testcase.

#Each test case is independent.

#NOTE: In this testing file, the variable names which we use for the 2-dimensional list and the pieces are different.

@pytest.fixture()

def il(): #Initialisng the 2d List

    iwhrook1=["rook", "\*][", 7, 0]

    iwhrook2=["rook", "\*][", 7, 7]

    iwhhor1=["horse", "\*/>", 7, 1]

    iwhhor2=["horse", "\*/>", 7, 6]

    iwhbish1=["bishop", "\*A", 7, 2]

    iwhbish2=["bishop", "\*A", 7, 5]

    iwhqueen=["queen", "\*Q", 7, 3]

    iwhking=["king", "\*$", 7, 4]

    iP1=["pawn", "\*^", 6, 0]

    iP2=["pawn", "\*^", 6, 1]

    iP3=["pawn", "\*^", 6, 2]

    iP4=["pawn", "\*^", 6, 3]

    iP5=["pawn", "\*^", 6, 4]

    iP6=["pawn", "\*^", 6, 5]

    iP7=["pawn", "\*^", 6, 6]

    iP8=["pawn", "\*^", 6, 7]

    #Creating Black pieces

    iblrook1=["rook", "][", 0, 0]

    iblrook2=["rook", "][", 0, 7]

    iblhor1=["horse", "/>", 0, 1]

    iblhor2=["horse", "/>", 0, 6]

    iblbish1=["bishop", "A", 0, 2]

    iblbish2=["bishop", "A", 0, 5]

    iblqueen=["queen", "Q", 0, 3]

    iblking=["king", "$", 0, 4]

    ip1=["pawn", "^", 1, 0]

    ip2=["pawn", "^", 1, 1]

    ip3=["pawn", "^", 1, 2]

    ip4=["pawn", "^", 1, 3]

    ip5=["pawn", "^", 1, 4]

    ip6=["pawn", "^", 1, 5]

    ip7=["pawn", "^", 1, 6]

    ip8=["pawn", "^", 1, 7]

    return [[iblrook1, iblhor1, iblbish1, iblqueen, iblking, iblbish2, iblhor2, iblrook2],[ip1, ip2, ip3, ip4, ip5, ip6, ip7, ip8],[iemp]\*8,[iemp]\*8,[iemp]\*8,[iemp]\*8,[iP1, iP2, iP3, iP4, iP5, iP6, iP7, iP8],[iwhrook1, iwhhor1, iwhbish1, iwhqueen, iwhking, iwhbish2, iwhhor2, iwhrook2]]

#By using this function, the value at the starting address will be changed to iemp. The value at the ending address will be changed to the piece which was originally at the starting address. And, the rownumber and the columnnumber of the piece which is moving is changed.

def tmove(tl, add1, add2):

    tl[add2[0]][add2[1]]=tl[add1[0]][add1[1]] #Ending address is filled

    tl[add1[0]][add1[1]]=iemp #Starting address is emptied

    tl[add2[0]][add2[1]][2]=add2[0]

    tl[add2[0]][add2[1]][3]=add2[1]

###Sample Code for Testing

##def test\_check\_\*(il):  #\* should be filled with the name of the checkmate or if the checkmate has no name, give some numbers along with your name (i.e, Guru, Subham)

##    iemp=emp=["", "", 0, 0]

##    tmove(il, \* , \*) #\*-Starting address and ending address as a tuple,

##    tmove(il, \*, \*)

##        .

##        .

##        .

##

##    displaylist(il) #In case the test case is failed, this command will display the 2d list so that you can CROSSCHECK if the configuration is same as the reference configuration.

##    add = \*  #\*-King's location as a tuple

##    result=checkmate(add, il)

##    assert result==True

def test\_check1(il):

    displaylist(il)

    add = (0,4)

    result = checkmate(add, il)

    assert result == False #I have given False because this configuration is actually not a checkmate

def test\_check2(il):

    displaylist(il)

    tmove(il, (6,4), (4,4))

    tmove(il, (1,4), (3,4))

    tmove(il, (7,5), (4,2))

    tmove(il, (1,5), (2,5))

    tmove(il, (7,3), (3,7))

    tmove(il, (0,2), (5,5))

    tmove(il, (7,2), (4,1))

    tmove(il, (1,6), (3,6))

    displaylist(il)

    add = (0,4)

    result = checkmate(add, il)

    assert result == False

def test\_check\_foolsmate(il):

    tmove(il, (6,6), (4,6))

    tmove(il, (6,5), (5,5))

    tmove(il, (1,4), (3,4))

    tmove(il, (0,3), (4,7))

    displaylist(il)

    add=(7,4)

    result=checkmate(add, il)

    assert result==True

def test\_check\_scholarsmate(il):

    tmove(il,(6,4), (4,4))

    tmove(il,(1,4), (3,4))

    tmove(il,(7,3), (1,5))

    tmove(il, (0,6), (2,5))

    tmove(il, (7,5), (4,2))

    tmove(il, (0,5), (3,2))

    displaylist(il)

    add=(0,4)

    result=checkmate(add,il)

    assert result==True

def test\_check\_smotheredmate(il):

    iemp=emp=["", "", 0, 0]

    tmove(il, (1,4), (3,4))

    tmove(il, (6,4), (4,4))

    tmove(il, (7,1), (5,2))

    tmove(il, (7,6), (6,4))

    tmove(il, (0,1), (5,5))

    tmove(il, (6,6), (5,6))

    displaylist(il)

    add=(7,4)

    result=checkmate(add, il)

    assert result==True

def test\_check\_hippopotamusmate(il):

    tmove(il, (1,4), (3,4))

    tmove(il, (6,4), (4,4))

    tmove(il, (7,1), (5,2))

    tmove(il, (7,6), (6,4))

    tmove(il, (0,1), (5,5))

    tmove(il, (0,3), (3,6))

    tmove(il, (7,2), (3,6))

    tmove(il, (6,6), (5,6))

    il[6][3]=iemp

    displaylist(il)

    add=(7,4)

    result=checkmate(add, il)

    assert result==True

def test\_check\_blackburneshillingmate(il):

    tmove(il, (7,5), (6,4))

    tmove(il, (7,6), (1,5))

    tmove(il, (7,7), (7,5))

    tmove(il, (0,3), (4,4))

    tmove(il, (0,1), (5,5))

    displaylist(il)

    il[1][4]=iemp

    il[6][6]=iemp

    add=(7,4)

    result=checkmate(add, il)

    assert result==True

def test\_check\_legallsmate(il):

    tmove(il, (1,6), (2,6))

    tmove(il, (7,1), (3,3))

    tmove(il, (7,6), (3,4))

    tmove(il,(6,4), (4,4))

    tmove(il, (0,2), (7,3))

    tmove(il, (1,3), (2,3))

    tmove(il, (0,4), (1,4))

    tmove(il, (7,5), (1,5))

    displaylist(il)

    add=(1,4)

    result=checkmate(add, il)

    assert result==True

def test\_check\_smotheredmatequeenspawn(il):

    tmove(il, (0,1), (2,2))

    tmove(il, (0,3), (1,4))

    tmove(il, (0,6), (5,3))

    il[0][5]=iemp

    tmove(il, (7,1), (6,3))

    tmove(il, (7,6), (5,5))

    tmove(il, (6,0), (4,1))

    tmove(il, (6,2), (4,2))

    tmove(il, (7,2), (4,5))

    displaylist(il)

    add=(7,4)

    result=checkmate(add, il)

    assert result==True

def test\_check\_seacadetmate(il):

    tmove(il, (1,3), (2,3))

    tmove(il, (7,1), (3,3))

    tmove(il, (7,6), (3,4))

    tmove(il, (0,4), (1,4))

    tmove(il, (7,5), (1,5))

    tmove(il, (7,4), (7,6))

    tmove(il, (7,7), (7,5))

    tmove(il, (0,2), (7,3))

    il[6][2]=iemp

    il[6][3]=iemp

    tmove(il, (6,4), (4,4))

    il[0][1]=iemp

    displaylist(il)

    add=(1,4)

    result=checkmate(add, il)

    assert result==True

def test\_check\_twopawnmate(il):

    il=[[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp] ]

    iblking=["king", "$", 0, 4]

    iwhking=["king", "\*$", 2, 4]

    iP4=["pawn", "\*^", 1, 3]

    iP5=["pawn", "\*^", 1, 4]

    il[0][4] = iblking

    il[1][3] = iP4

    il[1][4] = iP5

    il[2][4] = iwhking

    displaylist(il)

    add = (0, 4)

    result = checkmate(add, il)

    assert result==True

def test\_check\_diagonalmate(il):

    il=[[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp] ]

    iblrook1=["rook", "][", 0, 0]

    iblrook2=["rook", "][", 0, 5]

    iblking=["king", "$", 0, 6]

    iwhqueen=["queen", "\*Q", 1, 6]

    iwhking=["king", "\*$", 7, 6]

    iP2=["pawn", "\*^", 5, 1]

    iwhbish1=["bishop", "\*A", 7, 0]

    iwhrook1=["rook", "\*][", 7, 5]

    ip1=["pawn", "^", 1, 0]

    ip2=["pawn", "^", 1, 1]

    ip3=["pawn", "^", 1, 2]

    ip6=["pawn", "^", 1, 5]

    ip7=["pawn", "^", 1, 6]

    iP1=["pawn", "\*^", 6, 0]

    iP6=["pawn", "\*^", 6, 5]

    iP7=["pawn", "\*^", 6, 6]

    iP8=["pawn", "\*^", 6, 7]

    il[0][0] = iblrook1

    il[1][0] = ip1

    il[1][1] = ip2

    il[1][2] = ip3

    il[0][5] = iblrook2

    il[0][6] = iblking

    il[1][5] = ip6

    il[1][6] = ip7

    il[1][6] = iwhqueen

    il[6][0] = iP1

    il[5][1] = iP2

    il[6][5] = iP6

    il[6][6] = iP7

    il[6][7] = iP8

    il[7][0] = iwhbish1

    il[7][5] = iwhrook1

    il[7][6] = iwhking

    displaylist(il)

    add = (0, 6)

    result = checkmate(add, il)

    assert result==True

def test\_check\_fianchettomate(il):

    tmove(il, (7,2), (2,5))

    tmove(il, (6,1), (5,1))

    tmove(il, (7,7), (7,5))

    tmove(il, (7,4), (7,6))

    tmove(il, (7,1), (2,7))

    tmove(il, (0,7), (0,5))

    tmove(il, (0,4), (0,6))

    tmove(il, (1,6), (2,6))

    il[0][1] = il[0][2] = il[0][3] = il[1][3] = il[1][4] = emp

    il[6][2] = il[6][3] = il[6][4] = il[7][0] = il[7][3] = emp

    displaylist(il)

    add = (0, 6)

    result = checkmate(add, il)

    assert result==True

def test\_check\_subhamsriramgame1(il):

    il=[[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp],[emp,emp, emp, emp, emp, emp, emp, emp] ]

    iwhking=["king", "\*$", 7, 5, True]

    iblking=["king", "$", 1, 2]

    iblrook1=["rook", "][", 7,0]

    iblqueen=["queen", "Q", 6, 1]

    iblbish2=["bishop", "A", 3, 4]

    il[7][5]=iwhking

    il[1][2]=iblking

    il[7][0]=iblrook1

    il[6][1]=iblqueen

    il[3][4]=iblbish2

    displaylist(il)

    add = (7,5)

    print(checkmate(add, il))

    result = checkmate(add, il)

    assert result==True

log\_message.py

DEBUG = False

def log(\*msg):

    if DEBUG:

        for i in msg:

            print(i, end="")

        print()

main.py

if \_\_name\_\_ == "\_\_main\_\_":

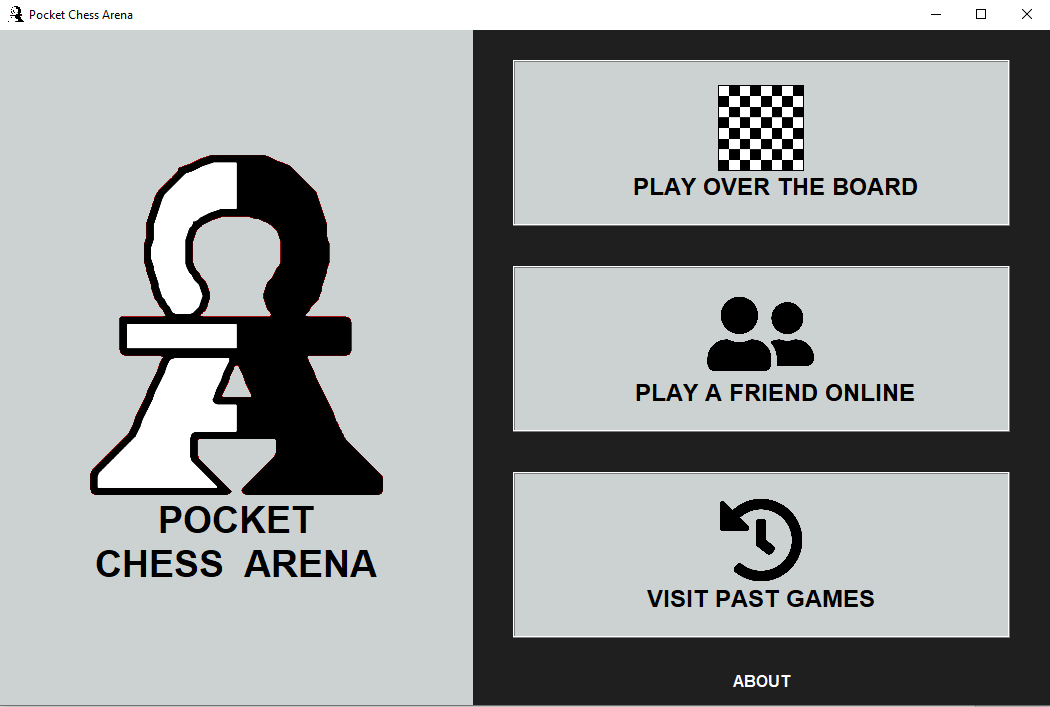
    import home

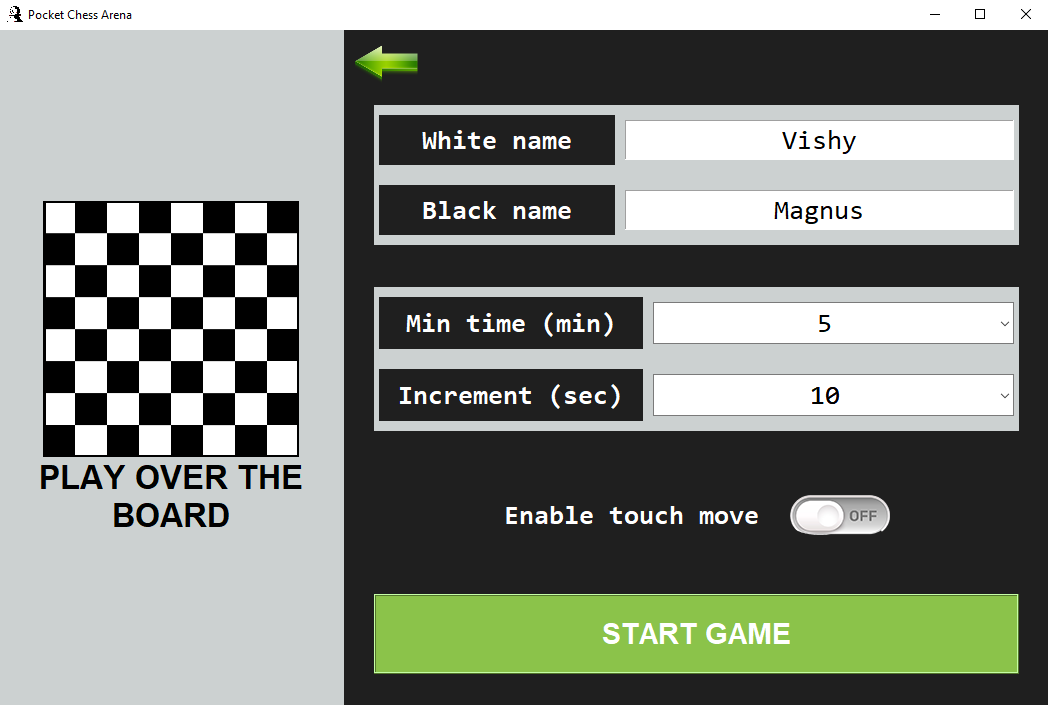
    #All the required modules and required created files for every file in the program has been imported.

    home.main()

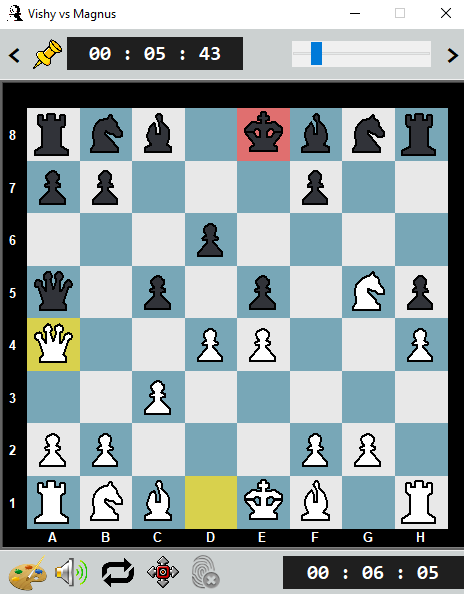
**SCREENSHOTS**

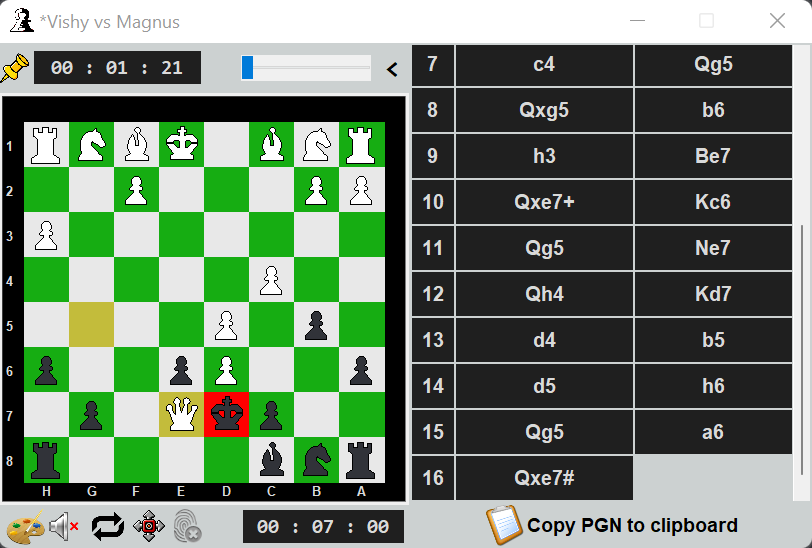
**Main Menu**:

**Settings:**



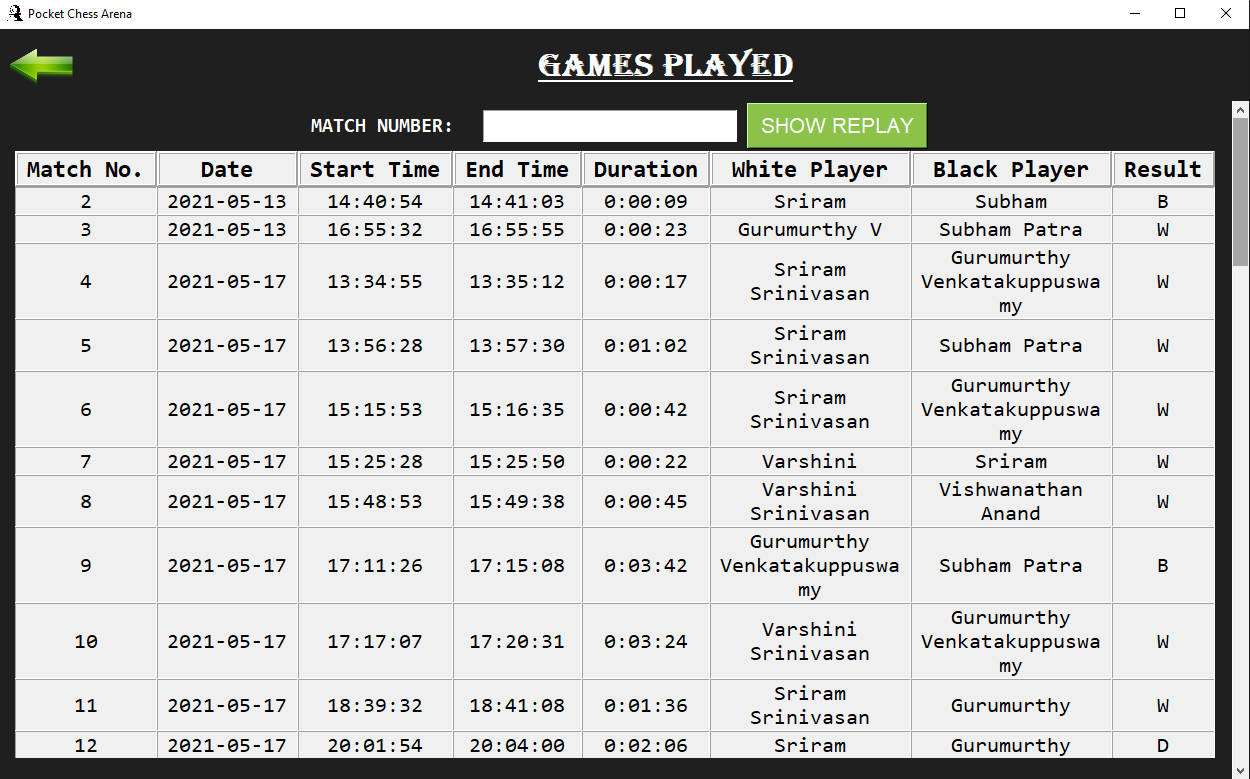
**Game:**

****

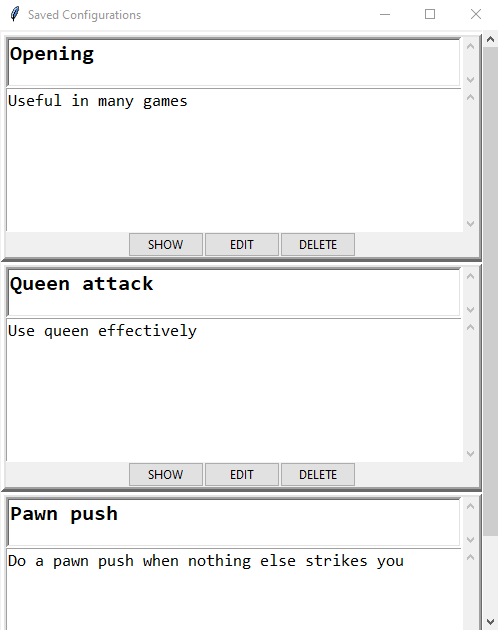
****

**Play Online:**

**Games Played:**



**Replay:**

**BIBLIOGRAPHY**

* Stackoverflow - [www.stackoverflow.com](http://www.stackoverflow.com)
* Turtle and tkinter resources-
  + <https://docs.python.org/3/library/turtle.html>
  + <https://docs.python.org/3/library/tk.html>
  + <https://tkdocs.com/tutorial/styles.html>
  + <https://wiki.tcl-lang.org/page/List+of+ttk+Themes>
  + <https://www.cs.mcgill.ca/~hv/classes/MS/TkinterPres/>
  + <https://realpython.com/python-gui-tkinter/#understanding-widget-naming-conventions>
* Flask and requests documentation - <https://flask.palletsprojects.com/en/2.0.x/>
* Microsoft Azure servers (for hosting the server)
* Pixilart (<https://www.pixilart.com>)
* PGN Utilities (<https://onlinepngtools.com>)