# Import statements

import tkinter as tk # Import tkinter modules to workspace

from tkinter import ttk, messagebox

from PIL import ImageTk, Image

# UNIVERSAL CONSTANTS

GAMEMODES = ["Classic","Fantasy"] # Available gamemode variations

UNIT\_STRENGTH = ['1','9','8','7','6','5','4','3','2','S','F','T'] # Array holding all unit strengths (used to create player sheet rosters)

# Dictionary holding the names, strength, and quantity of units for the "Classic" gamemode variation in the format strength:[unitName,quantity]

CLASSIC\_ROSTER = {'1':["Marshal",1],'9':["General",1],'8':["Colonel",2],'7':["Major",3],'6':["Captain",4],

                  '5':["Lieutenant",4],'4':["Sergeant",4],'3':["Miner",5],'2':["Scout",8],'S':["Spy",1],'F':["Flag",1],'T':["Bomb",6]}

# Dictionary holding the names, strength, and quantity of units for the "Classic" gamemode variation in the format strength:[unitName,quantity]

FANTASY\_ROSTER = {'1':["Dragon",1],'9':["Mage",1],'8':["Knight",2],'7':["Beast Rider",3],'6':["Sorceress",2],

                  '5':["Elemental",2],'4':["Elf",2],'3':["Dwarf",5],'2':["Scout",4],'S':["Slayer",1],'F':["Flag",1],'T':["Trap",6]}

# Initial global values

continuationCheck = False # Checks if system should be starting a new game (False) or loading existing game (True)

sound = True # Enables(True)/disables(False) game sound (default = True)

music = True # Enables(True)/disables(False) background music (default = True)

confirmationPrompt = True # Enables(True)/disables(False) pop-up prompt after making a valid move (default = True)

activePlayer = 'R' # Tracks the current active player for display and piece control (default = 'R')

arrayDict = {} # Stores button and backend data sub-dictionaries for main board and both player sheets

selectedButton = None # Indicates which button the user has clicked on (excluding GUI/navigational buttons)

lastBG = None # Stores the previous background color of the selected button

tileContent = "" # Stores the backend data of what was positioned on the tile selected by the user

deploymentComplete = [] # Stores the identity of players that have confirmed their deployment

oldX, oldY = 0, 0 # Tracks the coordinates of the "source" tile when determining movement

unitAbility = "None" # Tracks which unit ability is in use to determine legal movement tiles

winningPlayer = "" # Stores identity of winning player, if any

class Window(tk.Toplevel): # Base window class

    def \_\_init\_\_(self, parent, title, type):

        super().\_\_init\_\_(parent)

        self.title(title) # Sets title of created window

        '''def get\_image(unit):

            path = "images"

            if selectedGamemode.get() == "Classic":

                path += "\\classic"

            else:

                path += "\\fantasy"

            if unit[-1:] == 'R':

                path += "\\red"

            else:

                path += "\\blue"

            path += ("\\"+unit[1]+".png")

            return ImageTk.PhotoImage(Image.open(path))'''

        '''def get\_images(tileArray,tileStateArray):

                for y in range(len(arrayDict[tileArray])):

                    for x in range(len(arrayDict[tileArray][y])):

                        img=ImageTk.PhotoImage(Image.open("%s.png"%arrayDict[tileStateArray][y][x]))

                        arrayDict[tileArray][y][x].configure(image=img)'''

        class BoardFunctionality(): # Controls various functions related to board interactions

            # Changes ownership of active player status

            def end\_turn(player):

                global activePlayer

                if player == 'R': # Ending Red's turn

                    activePlayer = 'B' # Change active player

                    turnIndicator.configure(bg="blue") # Change turn indicator

                    if len(deploymentComplete) != 2: # While deployment incomplete

                        for child in root.winfo\_children():

                            if child.winfo\_name() == "!window4": # Hide previous active player's sheet

                                child.wm\_attributes('-disabled',True)

                                child.withdraw()

                            elif child.winfo\_name() != "!window": # Unhide all but mainMenu

                                child.wm\_attributes('-disabled',False)

                                child.deiconify()

                else: # Ending Blue's turn

                    activePlayer = 'R' # Change active player

                    turnIndicator.configure(bg="red") # Change turn indicator

                    if len(deploymentComplete) != 2: # While deployment incomplete

                        for child in root.winfo\_children():

                            if child.winfo\_name() == "!window5": # Hide previous active player's sheet

                                child.wm\_attributes('-disabled',True)

                                child.withdraw()

                            elif child.winfo\_name() != "!window": # Unhide all but mainMenu

                                child.wm\_attributes('-disabled',False)

                                child.deiconify()

                if (len(deploymentComplete) == 0 or len(deploymentComplete) == 2) and winningPlayer == "": # True when both players are finished deploying, neither are, and neither player has won

                    Root.open\_window(self,"Change Player", "blocker") # Create privacy blocker to allow trade of control

                    BoardFunctionality.change\_visibility('tileArrayM','tileStateArrayM') # Hide identity of inactive units, reveal active units

                elif winningPlayer != "": # If a player has won

                    Root.open\_window(self,"Game Over","gameOver") # Create game over screen

                if activePlayer in deploymentComplete and len(deploymentComplete) != 2: # If only one player has finished deployment, skip finished player's deployment step

                    BoardFunctionality.end\_turn(activePlayer)

                BoardFunctionality.update\_deployment\_restrictions('tileArrayM') # Update board display

                if len(deploymentComplete) == 2: # While deployment complete for both players, destroy both playerSheets

                    for child in root.winfo\_children():

                        if child.winfo\_name() == "!window4":

                            child.destroy()

                        elif child.winfo\_name() == "!window5":

                            child.destroy()

            # Resolves combat between attacking and defending units

            def resolve\_combat(attacker,defender):

                global winningPlayer

                if selectedGamemode.get() == "Classic": # Checks gamemode for correct name list

                    atkPiece = CLASSIC\_ROSTER[attacker[1]][0] # Gets name of attacking piece

                    defPiece = CLASSIC\_ROSTER[defender[1]][0] # Gets name of defending piece

                    # Checks if slayer/spy is attacking

                    try:

                        atkStrength = int(attacker[1]) # Assigns strength of unit

                    except: # Slayer/spy is attacking piece

                        if defender[1] == '1': # If the defender is the dragon/marshal

                            atkStrength = 11 # Slayer/spy defeats only the dragon/marshal on attack

                            print("Marshal Assassinated")

                        else: # All other defenders

                            atkStrength = 0 # Slayer/spy loses to any other unit, except flag

                    if atkStrength == 1: # Corrects dragon/marshal strength

                        atkStrength = 10

                    # Checks if slayer/spy, trap/bomb, or flag is defending

                    try:

                        defStrength = int(defender[1]) # Assigns strength of unit

                    except:

                        if defender[1] == 'F': # Flag is defender

                            defStrength = -1 # Flag always loses

                            winningPlayer = activePlayer # Declare capturing player as winner

                            print("Flag Captured")

                        elif defender[1] == 'T' and atkStrength == 3: # Trap/bomb is defender and attacker can difuse

                            defStrength = 0 # Trap loses

                            print("Bomb Captured")

                        elif defender[1] == 'T': # Trap/bomb is defender and attacker cannot difuse

                            defStrength = 11 # Trap always wins

                            print("Bomb Triggered")

                        else: # Slayer/spy is defender

                            defStrength = 0 # Slayer/spy loses to any piece on defense

                    if defStrength == 1: # Corrects dragon/marshal strength

                        defStrength = 10

                    # Resolves correct message for combat result

                    if activePlayer == 'R': # Red is active player

                        if atkStrength > defStrength: # Attacker wins result

                            messagebox.showinfo(title="Attack Summary",message="While attacking, Red's %s captured Blue's %s." % (atkPiece,defPiece)) # Results pop-up message

                            if attacker[-1:] == "\*": return attacker # If not already revealed, reveal surviving unit

                            else: return attacker+"\*" # Return survivor

                        elif atkStrength < defStrength: # Defender wins result

                            messagebox.showinfo(title="Attack Summary",message="While defending, Blue's %s captured Red's %s." % (defPiece,atkPiece)) # Results pop-up message

                            if defender[-1:] == "\*": return defender # If not already revealed, reveal surviving unit

                            else: return defender+"\*" # Return survivor

                        else: # Draw result

                            messagebox.showinfo(title="Attack Summary",message="In combat, both Red's %s and Blue's %s were captured." % (atkPiece,defPiece)) # Results pop-up message

                            return "Empty" # Return tile as empty

                    else: # Blue is active player

                        if atkStrength > defStrength: # Attacker wins result

                            messagebox.showinfo(title="Attack Summary",message="While attacking, Blue's %s captured Red's %s." % (atkPiece,defPiece)) # Results pop-up message

                            if attacker[-1:] == "\*": return attacker # If not already revealed, reveal surviving unit

                            else: return attacker+"\*" # Return survivor

                        elif atkStrength < defStrength: # Defender wins result

                            messagebox.showinfo(title="Attack Summary",message="While defending, Red's %s captured Blue's %s." % (defPiece,atkPiece)) # Results pop-up message

                            if defender[-1:] == "\*": return defender # If not already revealed, reveal surviving unit

                            else: return defender+"\*" # Return survivor

                        else: # Draw result

                            messagebox.showinfo(title="Attack Summary",message="In combat, both Blue's %s and Red's %s were captured." % (atkPiece,defPiece)) # Results pop-up message

                            return "Empty" # Return tile as empty

                else: # Checks gamemode for correct name list

                    atkPiece = FANTASY\_ROSTER[attacker[1]][0] # Gets name of attacking piece

                    defPiece = FANTASY\_ROSTER[defender[1]][0] # Gets name of defending piece

                    # Checks if slayer/spy is attacking

                    try:

                        atkStrength = int(attacker[1]) # Assigns strength of unit

                    except: # Slayer/spy is attacking piece

                        if defender[1] == '1': # If the defender is the dragon/marshal

                            atkStrength = 11 # Slayer/spy defeats only the dragon/marshal on attack

                            print("Dragon Slayed")

                        else: # All other defenders

                            atkStrength = 0 # Slayer/spy loses to any other unit, except flag

                    if atkStrength == 1: # Corrects dragon/marshal strength

                        atkStrength = 10

                    # Checks if slayer/spy, trap/bomb, or flag is defending

                    try:

                        defStrength = int(defender[1]) # Assigns strength of unit

                    except:

                        if defender[1] == 'F': # Flag is defender

                            defStrength = -1 # Flag always loses

                            winningPlayer = activePlayer # Declare capturing player as winner

                            print("Flag Captured")

                        elif defender[1] == 'T' and atkStrength == 3: # Trap/bomb is defender and attacker can difuse

                            defStrength = 0 # Trap/bomb loses

                            print("Trap Captured")

                        elif defender[1] == 'T': # Trap/bomb is defender and attacker cannot difuse

                            defStrength = 11 # Trap/bomb always wins

                            print("Trap Triggered")

                        else: # Slayer/spy is defender

                            defStrength = 0 # Slayer/spy loses to any piece on defense

                    if defStrength == 1: # Corrects dragon/marshal strength

                        defStrength = 10

                    # Resolves correct message for combat result

                    if activePlayer == 'R': # Red is active player

                        if atkStrength > defStrength: # Attacker wins result

                            messagebox.showinfo(title="Attack Summary",message="While attacking, Red's %s captured Blue's %s." % (atkPiece,defPiece)) # Results pop-up message

                            if attacker[-1:] == "\*": return attacker # If not already revealed, reveal surviving unit

                            else: return attacker+"\*" # Return survivor

                        elif atkStrength < defStrength: # Defender wins result

                            messagebox.showinfo(title="Attack Summary",message="While defending, Blue's %s captured Red's %s." % (defPiece,atkPiece)) # Results pop-up message

                            if defender[-1:] == "\*": return defender # If not already revealed, reveal surviving unit

                            else: return defender+"\*" # Return survivor

                        else: # Draw result

                            messagebox.showinfo(title="Attack Summary",message="In combat, both Red's %s and Blue's %s were captured." % (atkPiece,defPiece)) # Results pop-up message

                            return "Empty" # Return tile as empty

                    else: # Blue is active player

                        if atkStrength > defStrength: # Attacker wins result

                            messagebox.showinfo(title="Attack Summary",message="While attacking, Blue's %s captured Red's %s." % (atkPiece,defPiece)) # Results pop-up message

                            if attacker[-1:] == "\*": return attacker # If not already revealed, reveal surviving unit

                            else: return attacker+"\*" # Return survivor

                        elif atkStrength < defStrength: # Defender wins result

                            messagebox.showinfo(title="Attack Summary",message="While defending, Red's %s captured Blue's %s." % (defPiece,atkPiece)) # Results pop-up message

                            if defender[-1:] == "\*": return defender # If not already revealed, reveal surviving unit

                            else: return defender+"\*" # Return survivor

                        else: # Draw result

                            messagebox.showinfo(title="Attack Summary",message="In combat, both Blue's %s and Red's %s were captured." % (atkPiece,defPiece)) # Results pop-up message

                            return "Empty" # Return tile as empty

            # Hide identity of inactive units, reveal active units

            def change\_visibility(tileArray,tileStateArray):

                for y in range(len(arrayDict[tileArray])):

                    for x in range(len(arrayDict[tileArray][y])):

                        curTile = arrayDict[tileStateArray][y][x] # Checks every piece on chosen board

                        if curTile[0] != activePlayer and curTile != "Empty" and curTile != "Impassible" and curTile[-1:] != "\*": # Tile has an unrevealed, inactive unit

                            '''if activePlayer == 'R':

                                arrayDict[tileArray][y][x].configure(text="?",image=ImageTk.PhotoImage(Image.open("images\\blueHidden.png"))) # Hide tile identity

                            else:

                                arrayDict[tileArray][y][x].configure(text="?",image=ImageTk.PhotoImage(Image.open("images\\redHidden.png"))) # Hide tile identity'''

                            arrayDict[tileArray][y][x].configure(text="?") # Hide tile identity

                            '''elif curTile == "Empty":

                                #arrayDict[tileArray][y][x].configure(text=curTile,image=None) # Reveal tile identity

                            elif curTile == "Impassible":

                                #arrayDict[tileArray][y][x].configure(text=curTile,image=None) # Reveal tile identity'''

                        else: # All other tiles

                            #arrayDict[tileArray][y][x].configure(text=curTile,image=ImageTk.PhotoImage(Image.open(get\_image(curTile)))) # Reveal tile identity

                            arrayDict[tileArray][y][x].configure(text=curTile) # Reveal tile identity

            # Updates board to only allow players to deploy units in their own deployment zone

            def update\_deployment\_restrictions(tileArray):

                for y in range(len(arrayDict[tileArray])):

                    for x in range(len(arrayDict[tileArray][y])): # Checks every piece on chosen board

                        if selectedGamemode.get() == "Classic": i = 4 # Gamemode determines which tiles are in the middle of the board

                        else: i = 3

                        if len(deploymentComplete) != 2: # Both players not finished deploying

                            if activePlayer == 'R': # Red player is deploying

                                if y < i: # Blue deployment zone (Top 3-4 rows)

                                    arrayDict[tileArray][y][x].configure(state="disabled") # Disable Blue deployment zone

                                elif y >= (len(arrayDict[tileArray])-i): # Red deployment zone (Bottom 3-4 rows)

                                    arrayDict[tileArray][y][x].configure(state="normal") # Enable Red deployment zone

                            else: # Blue player is deploying

                                if y < i: # Blue deployment zone (Top 3-4 rows)

                                    arrayDict[tileArray][y][x].configure(state="normal") # Disable Red deployment zone

                                elif y >= (len(arrayDict[tileArray])-i): # Red deployment zone (Bottom 3-4 rows)

                                    arrayDict[tileArray][y][x].configure(state="disabled") # Enable Blue deployment zone

                        else:

                            arrayDict[tileArray][y][x].configure(state="normal") # When both players finished deploying, enable all squares

            # Controls activation of special unit abilities or lackthereof

            def activate\_unit\_ability():

                global unitAbility, tileContent

                abilityButton.configure(state="disabled") # Disallow another press with currently held unit

                if tileContent[-1:] != "\*": # Checks that unit is not revealed

                    if selectedGamemode.get() == "Classic": # Checks current gamemode to get names

                        if tileContent[1] == '2': # Selected unit is a scout

                            unitAbility = CLASSIC\_ROSTER[tileContent[1]][0] # Assign unit ability to scout ability

                            tileContent += "\*" # Mark unit as revealed

                        elif tileContent[1] == 'S' or tileContent[1] == '3' or tileContent[1] == 'T' or tileContent[1] == 'F': # Spy, Miner, Bomb, and Flag units have special rules that are always active

                            messagebox.showinfo(title="Passive Abilty",message="The selected unit's special rules are always active.") # Passive ability warning pop-up message

                        else: # Any other non-scout unit

                            messagebox.showinfo(title="No Abilty",message="The selected unit has no active ability.") # No ability warning pop-up message

                    else: # Fantasy gamemode variation

                        if tileContent[1] == 'S' or tileContent[1] == '3' or tileContent[1] == 'T' or tileContent[1] == 'F': # Slayer, Dwarf, Trap, and Flag units have special rules that are always active

                            messagebox.showinfo(title="Passive Abilty",message="The selected unit's special rules are always active.") # Passive ability warning pop-up message

                        else: # Any other unit

                            unitAbility = FANTASY\_ROSTER[tileContent[1]][0] # Assign unit ability to unit's name

                            tileContent += "\*" # Mark unit as revealed

                else: # If unit is revealed

                    messagebox.showinfo(title="Abilty Depleted",message="The selected unit is revealed, and thus cannot use its ability.") # Unit revealed warning pop-up message

            # Checks that moves are legal, including special movement abilities

            def position\_check(x,y):

                if unitAbility == "Scout":

                    if not(abs(oldX-x) > 1 and abs(oldY-y) > 1):

                        if abs(oldX-x) > 1:

                            if oldX-x > 0:

                                for tile in range(x+1,oldX): # Moving left

                                    print("tile="+str(tile)+" oldX="+str(oldX)+" x="+str(x))

                                    if arrayDict["tileStateArrayM"][y][tile] != "Empty":

                                        return False

                                return True

                            else:

                                for tile in range(oldX,x-1): # Moving right

                                    print("tile="+str(tile)+" oldX="+str(oldX)+" x="+str(x))

                                    if arrayDict["tileStateArrayM"][y][tile] != "Empty":

                                        return False

                                return True

                        else:

                            if oldY-y > 0:

                                for tile in range(y+1,oldY): # Moving up

                                    print("tile="+str(tile)+" oldY="+str(oldY)+" y="+str(y))

                                    if arrayDict["tileStateArrayM"][tile][x] != "Empty":

                                        return False

                                return True

                            else:

                                for tile in range(oldY,y-1): # Moving down

                                    print("tile="+str(tile)+" oldY="+str(oldY)+" y="+str(y))

                                    if arrayDict["tileStateArrayM"][tile][x] != "Empty":

                                        return False

                                return True

                elif unitAbility == "Elf" or unitAbility == "Sorceress" or unitAbility == "Mage":

                    return not(abs(oldX-x)+abs(oldY-y) > 4)

                elif unitAbility == "Knight" or unitAbility == "Beast Rider":

                    return not(abs(oldX-x)+abs(oldY-y) > 2)

                elif unitAbility == "Dragon":

                    if not(abs(oldX-x) > 1 and abs(oldY-y) > 1):

                        if abs(oldX-x) > 1:

                            if oldX-x > 0:

                                for tile in range(x,oldX):

                                    if arrayDict["tileStateArrayM"][y][tile] == "Empty":

                                        return False

                                return True

                            else:

                                for tile in range(oldX,x):

                                    if arrayDict["tileStateArrayM"][y][tile] == "Empty":

                                        return False

                                return True

                        else:

                            if oldY-y:

                                for tile in range(y,oldY):

                                    if arrayDict["tileStateArrayM"][tile][x] == "Empty":

                                        return False

                                return True

                            else:

                                for tile in range(oldY,y):

                                    if arrayDict["tileStateArrayM"][tile][x] == "Empty":

                                        return False

                                return True

                else: # Default standard move/attack

                    return not(abs(oldX-x)+abs(oldY-y) > 1) and not(abs(oldX-x)+abs(oldY-y) == 0)

            # Reads the contents of the cell background data and moves pieces as allowed by valid moves, then updates the board buttons

            def move\_selection(tileArray,tileStateArray,y,x):

                global tileContent, oldX, oldY, unitAbility

                selectedTile = arrayDict[tileStateArray][y][x] # Indicates which tile is being accessed

                if selectedTile != "Impassible": # Checks that both destination and source tiles aren't impassible

                    if tileContent != "": # Runs if selected tile is second tile clicked (ie, selected tile is the destination tile)

                        if activePlayer in deploymentComplete: # If deployment is complete, follow normal movement rules and enable combat

                            if BoardFunctionality.position\_check(x,y): # Checks that unit making a legal move

                                # ADD UNIT ABILITIES TO ELSE IF HERE

                                if selectedTile[0] == activePlayer: # Checks that destination tile does not contain friendly unit

                                    messagebox.showinfo(title="Invalid Selection",message="You cannot end your movement on a tile containing a friendly unit.") # Invalid selection warning pop-up message

                                elif selectedTile != "Empty": # Checks is destination tile contains enemy unit

                                    '''if confirmationPrompt:

                                        Root.open\_window(self,"Move Confirmation",type="moveConfirm")'''

                                    BoardFunctionality.highlight\_selection(tileArray,tileStateArray,y,x) # Highlights the selected tile

                                    arrayDict[tileStateArray][y][x] = BoardFunctionality.resolve\_combat(tileContent,selectedTile) # Resolves combat between held unit and unit on tile

                                    tileContent = "" # Clears storage variable to allow new source tile

                                    unitAbility == "None" # Clears unit ability to return normal movement

                                    BoardFunctionality.end\_turn(activePlayer) # End the active player's turn

                                else: # If tile is empty

                                    BoardFunctionality.highlight\_selection(tileArray,tileStateArray,y,x)

                                    arrayDict[tileStateArray][y][x] = tileContent # Assigns destination tile to original value of source tile

                                    tileContent = "" # Clears storage variable to allow new source tile

                                    unitAbility == "None" # Clears unit ability to return normal movement

                                    BoardFunctionality.end\_turn(activePlayer) # End the active player's turn

                            else: # Unit is not making a legal move

                                messagebox.showinfo(title="Invalid Selection",message="You must move to an adjecent orthagonal tile, or according to the selected unit's special ability.") # Invalid selection warning pop-up message

                        else:

                            if selectedTile[0] == activePlayer: # Checks that destination tile does not contain friendly unit

                                messagebox.showinfo(title="Invalid Selection",message="You cannot end your movement on a tile containing a friendly unit.") # Invalid selection warning pop-up message

                            else: # If tile is empty

                                BoardFunctionality.highlight\_selection(tileArray,tileStateArray,y,x) # Highlights the selected tile

                                arrayDict[tileStateArray][y][x] = tileContent # Assigns destination tile to original value of source tile

                                tileContent = "" # Clears storage variable to allow new source tile

                                BoardFunctionality.end\_turn(activePlayer) # End the active player's turn

                    else: # Runs if selected tile is first tile clicked (ie, selected tile is the source tile)

                        if selectedTile != "Empty": # Won't allow an empty tile to be assigned as source tile

                            if selectedTile[0] == activePlayer: # Selected unit is controlled by active player

                                if (activePlayer in deploymentComplete) and selectedTile[1] != 'T' and selectedTile[1] != 'F': # Checks if deployment is finished for active player and that piece is mobile

                                    BoardFunctionality.highlight\_selection(tileArray,tileStateArray,y,x) # Highlights the selected tile

                                    tileContent = selectedTile # Set storage variable to source tile's value

                                    oldX,oldY = x,y # Stores coordinates of previous tile

                                    arrayDict[tileStateArray][y][x] = "Empty" # Set source tile value to empty to show piece has left tile

                                    abilityButton.configure(state="normal",bg='yellow') # Re-enables ability button for new held unit

                                elif (activePlayer not in deploymentComplete): # Checks that player is still deploying

                                    BoardFunctionality.highlight\_selection(tileArray,tileStateArray,y,x) # Highlights the selected tile

                                    tileContent = selectedTile # Set storage variable to source tile's value

                                    if tileStateArray[-1:] != "M": # Checks that player is placing new unit

                                        arrayDict['quantityStateArrayP%1c'%activePlayer][y] -= 1 # Reduces the quantity of selected unit left to deploy

                                        arrayDict['quantityArrayP%1c'%activePlayer][y].configure(text=str(arrayDict['quantityStateArrayP%1c'%activePlayer][y])) # Updates quantity counter

                                        if arrayDict['quantityStateArrayP%1c'%activePlayer][y] == 0: # Quantity of unit type remaining to place is 0

                                            arrayDict[tileStateArray][y][x] = "Disabled" # Set source tile value to disabled to show all pieces of that type are placed

                                    else: # Player is moving an already deployed unit

                                        arrayDict[tileStateArray][y][x] = "Empty" # Set source tile value to empty to show piece has left tile

                                else: # Deployment complete and unit is immobile

                                    messagebox.showinfo(title="Invalid Selection",message="That is not a valid piece of the active player.") # Invalid selection warning pop-up message

                            else: # Unit not controlled by active player

                                messagebox.showinfo(title="Invalid Selection",message="That is not a valid piece of the active player.") # Invalid selection warning pop-up message

                        else: # Tile is empty

                            messagebox.showinfo(title="Invalid Selection",message="That is not a valid piece of the active player.") # Invalid selection warning pop-up message

                else: # Tile is impassible

                    messagebox.showinfo(title="Invalid Selection",message="That tile is impassible terrain.")

                heldUnitTracker.configure(text=("Held Unit: "+tileContent)) # Updates value of unit tracker

            # Highlights selected cell in gold on click

            def highlight\_selection(tileArray,tileStateArray,y,x):

                global selectedButton, lastBG

                button = arrayDict[tileArray][y][x] # Assigns clicked button to temp var

                if selectedButton is not None: # If not first tile selected of game

                    selectedButton.configure(bg=lastBG) # Restore original background color of previously selected tile

                lastBG = button.cget("bg") # Store original background color

                selectedButton = button # Overrides old button with new button

                button.configure(bg="gold") # Updates background of tile to stay highlighted

                if arrayDict[tileStateArray][y][x] == "Disabled": # Checks if all units of type on playerSheet are placed/held

                    button.configure(state="disabled") # Disables unit type button

        # ??MERGE GENERATE + POPULATE??

        # Generates an empty board array based on the dimensions provided

        # arrayDict['tileArrayM'][y][x] format -> stored as yx  -> increments across

        def generate(arrayHeight, arrayWidth, useCase):

            global arrayDict

            # Expandable else if tree that determines where arrays are being created

            if useCase == "board":

                # Creates clickable tiles for the user

                arrayDict['tileArrayM'] = [[tk.Button(self,relief="groove",activebackground="gold") for x in range(arrayWidth)] for y in range(arrayHeight)]

                # Stores data on cells in background

                arrayDict['tileStateArrayM'] = [["Empty" for x in range(arrayWidth)] for y in range(arrayHeight)]

            elif useCase == "playerSheet":

                # Creates clickable tiles for the user

                arrayDict['tileArrayP%1c'%activePlayer] = [[tk.Button(self,relief="groove",activebackground="gold") for x in range(arrayWidth)] for y in range(arrayHeight)]

                # Stores data on cells in background

                arrayDict['tileStateArrayP%1c'%activePlayer] = [[(activePlayer+str(UNIT\_STRENGTH[y])) for x in range(arrayWidth)] for y in range(arrayHeight)]

                # Creates additional array to display unit quantity to players (unique to player sheets)

                if selectedGamemode.get() == "Classic": # Checks current gamemode to chose correct unit quantities

                    arrayDict['quantityStateArrayP%1c'%activePlayer] = [CLASSIC\_ROSTER[UNIT\_STRENGTH[y]][1] for y in range(arrayHeight)] # Background data

                else: # Fantasy gamemode variant

                    arrayDict['quantityStateArrayP%1c'%activePlayer] = [FANTASY\_ROSTER[UNIT\_STRENGTH[y]][1] for y in range(arrayHeight)] # Background data

                arrayDict['quantityArrayP%1c'%activePlayer] = [tk.Label(self,text=arrayDict['quantityStateArrayP%1c'%activePlayer][y]) for y in range(arrayHeight)] # Quantity displays

                for y in range(arrayHeight): # For each quantity display

                    arrayDict['quantityArrayP%1c'%activePlayer][y].grid(row=y,column=1,sticky=(tk.N,tk.S,tk.E,tk.W)) # Positions quantity displays on playerSheet

        # Fills generated arrays with properly formatted buttons

        def format\_initial\_array(tileArray, tileStateArray, leftOffset=0, offsetSize=0): # leftOffset - Number of columns to offset for GUI on left of board grid

            # Section for left-side GUI widgets

            if leftOffset > 0:

                for i in range(leftOffset):

                    self.columnconfigure(i, weight= 2, minsize=offsetSize)

            # Creates array grid and places button within each cell

            for y in range(len(arrayDict[tileArray])):

                self.rowconfigure(y, weight= 1, minsize=50)

                for x in range(len(arrayDict[tileArray][y])):

                    self.columnconfigure(x+leftOffset, weight= 1, minsize=50)

                    # Call sub-dictionaries (children of arrayDict) of button variable names and configure appropriately, each initial assigned as str(y)+str(x)'[row-number][col-number]'

                    arrayDict[tileArray][y][x].configure(text=arrayDict[tileStateArray][y][x],

                                        command=lambda tempTA=tileArray,

                                        tempTSA=tileStateArray,tempY=y,tempX=x:

                                        [BoardFunctionality.move\_selection(tempTA,tempTSA,tempY,tempX)])

                    if tileArray[-1:] == "M": # Checks if main board is being generated

                        self.rowconfigure(y, weight= 1, minsize=75)

                        self.columnconfigure(x+leftOffset, weight= 1, minsize=75)

                        # Sets up impassible tiles in middle of board and player deployment zones based on gamemode

                        if selectedGamemode.get() == "Classic": # Positions middle of board and deployment zones based on gamemode

                            i,j = 4,5

                            Root.set\_geometry(self,750+((leftOffset+1)\*offsetSize),750) # Sets size of board window based on gamemode

                        else: # Fantasy gamemode variant

                            i,j = 3,4

                            Root.set\_geometry(self,750+((leftOffset+1)\*offsetSize),600) # Sets size of board window based on gamemode

                        if (y==i or y==j) and (x==2 or x==3 or x==6 or x==7): # Indicates impassible terrain

                            arrayDict[tileStateArray][y][x] = "Impassible" # Changes tile background data

                            arrayDict[tileArray][y][x].configure(text="Impassible",bg="#13f3ee") # Formats tiles to cyan backing

                        elif y < i: # Indicates Blue player deployment zone

                            arrayDict[tileArray][y][x].configure(bg="#a7cffa") # Formats tiles to light blue backing

                        elif y >= (len(arrayDict[tileArray])-i): # Indicates Red player deployment zone

                            arrayDict[tileArray][y][x].configure(bg="#ea9f9f") # Formats tiles to light red backing

                        arrayDict[tileArray][y][x].configure(state="disabled") # Defaults all buttons to disabled

                        # Section for right-side GUI widgets on main board

                        global rulesButton, abilityButton

                        self.columnconfigure(leftOffset+len(arrayDict[tileArray][0]), weight= 2, minsize=offsetSize) # Configures the widgets of the right side GUI

                        # Example of a GUI element that could be placed here

                        RIGHTELEMENT1 = tk.Text(self,wrap='word')

                        RIGHTELEMENT1.insert(1.0,"Here's where a GUI element will go.")

                        RIGHTELEMENT1.grid(column=leftOffset+len(arrayDict[tileArray][0]),row=0,rowspan=1)

                        RIGHTELEMENT1.configure(bg=self.cget('bg'),relief='flat',state='disabled')

                        # Opens rules booklet

                        rulesButton = tk.Button(self,text="Rules of Play",bg='green',command=lambda:[Root.open\_window(self,"Rules of Play","ruleSheet")])

                        rulesButton.grid(column=leftOffset+len(arrayDict[tileArray][0]),row=len(arrayDict[tileArray])-3,rowspan=1) # Positions rules button on board

                        # Activates unit special ability, if any

                        abilityButton = tk.Button(self,text="Unit Ability",bg='yellow',state="disabled",command=lambda:[BoardFunctionality.activate\_unit\_ability()])

                        abilityButton.grid(column=leftOffset+len(arrayDict[tileArray][0]),row=len(arrayDict[tileArray])-2,rowspan=1) # Positions ability button on board

                    arrayDict[tileArray][y][x].grid(column=x+leftOffset,row=y,sticky=(tk.N,tk.S,tk.E,tk.W)) # Positions buttons within the window's grid

        # Allows user to close listed windows manually, otherwise prevents deletion of critical windows/components

        if type != "exit" and type != "invalid" and type != "options" and type != "pauseMenu":

            self.protocol("WM\_DELETE\_WINDOW", root.close\_confirm)

        # Large and expandable else if tree determines what type of window is being built to format accordingly

        if type == "mainMenu": # UNFINISHED - Main menu and application splash screen for startup

            Root.set\_geometry(self,300,500) # Set window dimensions and screen position

            tk.Button(self,

                       text="Play Game",

                       command=lambda: [root.open\_window("Game Setup", "setup"), self.withdraw()]).pack(side=tk.TOP, pady=5) # Create new game and hide main menu

            tk.Button(self,

                       text="Continue",

                       command=lambda: [changeContinueCheck(), root.open\_window("Game Load", "setup"), self.withdraw()]).pack(pady=5) # Load saved game and hide main menu

            tk.Button(self,

                       text="Options",

                       command=lambda: [root.open\_window("Options", "options"), self.withdraw()]).pack(pady=5) # Open options menu and hide main menu

            tk.Button(self,

                       text="Exit",

                       command=root.close\_confirm).pack(pady=5) # Close application

            tk.Label(self,image=ImageTk.PhotoImage(Image.open("1.png"))).pack()

            tk.Label(self,image=ImageTk.PhotoImage(Image.open("2.png"))).pack()

            def changeContinueCheck(): # If saving is being loaded

                global continuationCheck

                continuationCheck = True

        elif type == "options":  # UNFINISHED - Options menu

            Root.set\_geometry(self,200,300) # Set window dimensions and screen position

            tk.Button(self,

                       text="Save options",

                       command=lambda: [mainMenu.deiconify(), self.destroy()]).pack(side=tk.LEFT)  # UNFINISHED - Update application options, save info to .txt

            tk.Button(self,

                       text="Cancel",

                       command=lambda: [mainMenu.deiconify(), self.destroy()]).pack(side=tk.RIGHT) # Return to main menu screen

        elif type == "pauseMenu": #UNFINISHED - Pause menu for game in progress

            Root.set\_geometry(self,200,300) # Set window dimensions and screen position

            tk.Button(self,

                       text="Return",

                       command=lambda: [self.destroy()]).pack(side=tk.TOP, pady=5) # Close pause menu and return to game

            # UNFINISHED SAVE AND LOAD FUNCTION

            tk.Button(self,

                       text="Save",

                       command=lambda: [self.destroy()]).pack(pady=5) # Save game state information to unique .txt file

            tk.Button(self,

                       text="Load",

                       command=lambda: [changeContinueCheck(), root.open\_window("Game Load", "setup"), self.destroy()]).pack(pady=5) # Load game state information from unique .txt file

            tk.Label(self,text="Options").pack(pady=5) # UNFINISHED - RadioButtons for sound and movement confirmation go here

            # UNFINISHED RADIO BUTTONS TO ENABLE SOUND/MUSIC/CONFIRMATION PROMPT

            tk.Button(self,

                       text="Main Menu",

                       command=lambda: [mainMenu.deiconify(), self.destroy()]).pack(side=tk.BOTTOM, pady=5) # Return to main menu

        elif type == "setup": # UNFINISHED - When creating new game, allows game settings to be chosen

            Root.set\_geometry(self,250,100) # Set window dimensions and screen position

            # This is kinda a stupid way to do things, but it works in theory

            # Save data needs to be saved in a way that allows program to differeniate sections of info

            '''if continuationCheck == True:

                savedPlayerCount = open("savedata.txt", "r")

                numPlayersEntry.insert(savedPlayerCount)

            else:

                numPlayersEntry.insert(0, "Enter the number of players, "+str(MIN\_PLAYERS)+"-"+str(MAX\_PLAYERS))

                tk.Button(self,

                        text="Continue",

                        command=lambda: addPlayers()).pack(side=tk.LEFT)  # UNFINISHED

                tk.Button(self,

                        text="Back",

                        command=lambda: [mainMenu.deiconify(), self.destroy()]).pack(side=tk.RIGHT) # Hide menu and destroy setup page'''

            global selectedGamemode

            selectedGamemode = tk.StringVar()

            selectedGamemode.set("Classic") # Set default gamemode

            tk.OptionMenu(self,selectedGamemode,\*GAMEMODES).pack(side=tk.TOP) # Create dropdown menu for gamemode

            tk.Button(self,

                    text="Continue",

                    command=lambda: end\_setup()).pack(side=tk.LEFT) # Ends setup and creates new game from selected settings

            tk.Button(self,

                    text="Back",

                    command=lambda: [mainMenu.deiconify(), self.destroy()]).pack(side=tk.RIGHT) # Unhide main menu and destroy setup page

            # Signals setup is complete and generates board based on chosen settings

            def end\_setup():

                root.open\_window("Board", "board") # Creates main board

                create\_player\_windows() # Creates the two playerSheets

                BoardFunctionality.update\_deployment\_restrictions("tileArrayM") # Enables buttons in Red player deployment zone

                self.destroy() # Destroys the setup window

            def create\_player\_windows(): # Creates the initial windows for players' personal sheets

                global activePlayer

                root.open\_window("Player 1", "playerSheet") # Creates playerSheet for Red player

                activePlayer = 'B'

                root.open\_window("Player 2", "playerSheet") # Creates playerSheet for Blue player

                activePlayer = 'R'

        elif type == "playerSheet": # Formats the players' personal sheets

            # Creates button array for units

            generate(12,1,"playerSheet")

            # Formats player sheets based on identity

            sheetHeader = ttk.Label(self) # Creates header for sheet

            if activePlayer == 'R': # Format Red playerSheet

                sheetHeader.configure(text="Player Number 1") # Denotes Red player as Player 1

                self.configure(background="red") # Sets sheet background to red

                Root.set\_geometry(self,200,750,(root.winfo\_screenwidth()/2)-(1050/2)-213) # Sets dimensions and position of window

                format\_initial\_array('tileArrayPR', 'tileStateArrayPR') # Formats Red playerSheet buttons correctly

            elif activePlayer == 'B': # Format Blue playerSheet

                sheetHeader.configure(text="Player Number 2") # Denotes Blue player as Player 2

                self.configure(background="blue") # Sets sheet background to blue

                Root.set\_geometry(self,200,750,(root.winfo\_screenwidth()/2)+(1050/2)+13) # Sets dimensions and position of window

                format\_initial\_array('tileArrayPB', 'tileStateArrayPB') # Formats Blue playerSheet buttons correctly

            # Creates an obvious error if player count is misinterpretted

            else:

                sheetHeader.configure(text="PLAYER SHEET ERROR")

                self.configure(background="yellow")

            sheetHeader.grid(row=0,column=2) # Positions header correctly in frame

            # Confirms that the players units are deployed as they wish, updates board display, ends turn, and creates blocker window

            tk.Button(self,

                       text="Confirm Deployment",

                       command=lambda:[deploymentComplete.append(activePlayer),

                                       BoardFunctionality.end\_turn(activePlayer),

                                       Root.open\_window(root,"Change Player", "blocker"),

                                       BoardFunctionality.change\_visibility('tileArrayM','tileStateArrayM')]).grid(row=11,column=2)

        elif type == "board": # Creates the main board window, where most of the interaction takes place

            global arrayDict, heldUnitTracker, turnIndicator

            # Generates correct board layout based on chosen gamemode

            if selectedGamemode.get() == "Classic":

                generate(10,10,"board")

            elif selectedGamemode.get() == "Fantasy":

                generate(8,10,"board")

            # Creates button to access the pause menu in the upper-lefthand corner of the main board

            pauseButton = tk.Button(self,text="Menu",command=lambda: [root.open\_window("Pause Menu", "pauseMenu")])

            pauseButton.grid(column=0,row=0,sticky=(tk.N,tk.W)) # Positions button correctly in frame

            # Creates indicator that shows the current active player (ie the player whose turn it is)

            tk.Label(self,text="Active Player:").grid(column=0,row=1)

            turnIndicator = tk.Label(self, bg="red") # Sets the initial value of the indicator to red (red is player 1)

            turnIndicator.grid(column=0,row=2,sticky=(tk.E,tk.W), padx=5) # Positions indicator correctly in frame

            # Creates indicator that shows identity of the held unit, if any (ie the unit being moved by the active player)

            heldUnitTracker = tk.Label(self,text=("Held Unit: "+tileContent))

            heldUnitTracker.grid(column=0,row=3) # Positions indicator correctly in frame

            format\_initial\_array("tileArrayM","tileStateArrayM",1,150) # Formats the main board correctly

        elif type == "ruleSheet": # Window that contains all\* rules information (\*most)

            Root.set\_geometry(self,800,750) # Set window dimensions and screen position

            # Block of text, roughly formatted

            tk.Label(self,text="PAGE UNDER CONSTRUCTION",wraplength=750).pack()

            tk.Label(self,text="--GENERAL RULES--",wraplength=750).pack()

            tk.Label(self,text="1. Capture your opponent's flag to win.",wraplength=750).pack()

            tk.Label(self,text="--------------------------------------------------------------------------------------------------------------------------------",wraplength=750).pack()

            tk.Label(self,text="--CLASSIC RULES--",wraplength=750).pack()

            tk.Label(self,text="Flag - Immovable; capturing the opponent's Flag wins the game.",wraplength=750).pack()

            tk.Label(self,text="Spy - Weakest piece, captured by any other attacking piece, but an attacking Spy can capture the Marshal.",wraplength=750).pack()

            tk.Label(self,text="Bomb - Immovable; any piece attacking a Bomb is removed from the game, unless the attacking piece was a Miner.",wraplength=750).pack()

            tk.Label(self,text="Scout - Can move any distance in a horizontal or vertical straight line without leaping over pieces or lakes; allows movement and attack in same turn.",wraplength=750).pack()

            tk.Label(self,text="Miner - Can defuse (i.e. capture) Bombs",wraplength=750).pack()

            tk.Label(self,text="Marshal - Most powerful piece, but vulnerable to capture by an attacking Spy",wraplength=750).pack()

            tk.Label(self,text="--------------------------------------------------------------------------------------------------------------------------------",wraplength=750).pack()

            tk.Label(self,text="--FANTASY RULES--",wraplength=750).pack()

            tk.Label(self,text="Flag - Immovable; capturing the opponent's Flag wins the game.",wraplength=750).pack()

            tk.Label(self,text="Slayer - Weakest piece, captured by any other attacking piece, but an attacking Slayer can capture the Dragon.",wraplength=750).pack()

            tk.Label(self,text="Trap - Immovable; any piece attacking a Trap is removed from the game, unless the attacking piece was a Dwarf.",wraplength=750).pack()

            tk.Label(self,text="Scout - Can move any distance in a horizontal or vertical straight line without leaping over pieces or lakes; allows movement and attack in same turn.",wraplength=750).pack()

            tk.Label(self,text="Dwarf - Can defuse (i.e. capture) Traps",wraplength=750).pack()

            tk.Label(self,text="Elf - May reveal, then attack an enemy piece within 2 squares (squares can be counted in any direction, including diagonal, and can cross forbidden zones); any attacked piece of rank 3 or lower is captured (including flags), otherwise no effect",wraplength=750).pack()

            tk.Label(self,text="Elemental - May reveal, then move to an adjecent square (not diagonal), then attack ALL adjecent pieces (including diagonal, and friendly pieces); any attacked pieces of rank 5 or lower are captured (including flags); if any attacked pieces are of rank 5 or higher (including traps) then the attacking piece is captured",wraplength=750).pack()

            tk.Label(self,text="Sorceress - May reveal, then force an enemy piece within 2 squares to reveal (squares can be counted in any direction, including diagonal, and can cross forbidden zones); if the revealed enemy piece is rank 5 or lower, it comes under the sorceress' control (including slayers, but not traps/flags)",wraplength=750).pack()

            tk.Label(self,text="Beast Rider- May reveal, then move 2 squares (squares can be counted in any direction except diagonal, and CANNOT cross forbidden zones); can move into a space containing an enemy piece, immediately ending its movement and resolving combat",wraplength=750).pack()

            tk.Label(self,text="Knight - May reveal, then move 2 squares (squares can be counted in any direction except diagonal, and CANNOT cross forbidden zones); can move into a space containing an enemy piece, immediately ending its movement and resolving combat",wraplength=750).pack()

            tk.Label(self,text="Mage - May reveal, then force an enemy piece within 2 squares to reveal (squares can be counted in any direction, including diagonal, and can cross forbidden zones)",wraplength=750).pack()

            tk.Label(self,text="Dragon - Most powerful piece, but vulnerable to capture by an attacking Slayer",wraplength=750).pack()

        elif type == "moveConfirm": # UNUSED/UNFINISHED -

            Root.set\_geometry(self,250,100) # Set window dimensions and screen position

            tk.Label(self,text="Confirm that this is your intended move?").pack(side=tk.TOP)

            tk.Button(self,

                    text="Confirm Move",

                    command=lambda: []).pack(side=tk.LEFT) # Player confirms move is correct, play proceeds as normal

            tk.Button(self,

                    text="Cancel Move",

                    command=lambda: [self.destroy()]).pack(side=tk.RIGHT) # Move is cancelled, held unit is returned, source tile info is wiped

            tk.Label(self,text="HINT: This setting can be disabled from the pause menu.",font=('Segoe UI',5)).pack(side=tk.BOTTOM) # HINT: It cannot :)

        elif type == "blocker": # Covers board while players exchange control

            self.attributes('-fullscreen',True) # Set window to lock in fullscreen

            tk.Label(self,text="Pass control to other player.",font=('Segoe UI',20)).pack()

            tk.Button(self,

                    text="Start Turn",font=('Segoe UI',15),

                    command=self.destroy).pack() # Destroys blocker

        elif type == "gameOver": # Displays that a player has won the game

            Root.set\_geometry(self,500,100) # Set window dimensions and screen position

            '''for child in root.winfo\_children():

                child.configure(state="disabled")

            self.configure(state="normal")'''

            if winningPlayer == 'R': # Checks which player has won

                tk.Label(self,text="Red Player has won the game by capturing their opponent's flag. Congratulations!").pack(side=tk.TOP)

            else: # Blue player wins

                tk.Label(self,text="Blue Player has won the game by capturing their opponent's flag. Congratulations!").pack(side=tk.TOP)

            tk.Button(self,

                       text="Exit application",

                       command=root.close\_all).pack(side=tk.LEFT) # Closes application

            tk.Button(self,

                       text="Observe the board",

                       command=self.destroy).pack(side=tk.RIGHT) # Returns to board

        elif type == "exit": # Exit confirmation window

            Root.set\_geometry(self,300,100) # Set window dimensions and screen position

            # If close is confirmed, calls close\_all function

            # Save functionality -> UNDER DEVELOPMENT

            tk.Label(self,text="Are you sure you want to close?").pack()

            tk.Button(self,

                       text="Save and close all windows",

                       command=root.close\_all).pack(side=tk.LEFT)

            # If close is not confirmed, closes confirmation window

            tk.Button(self,

                       text="Back",

                       command=self.destroy).pack(side=tk.RIGHT)

# Main foundational super-class from which all windows are created and initial system actions are taken on program start

class Root(tk.Tk):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.withdraw() # Hide this window on creation

    def close\_all(self):  # Closes all windows, including hidden

        for child in self.winfo\_children():

            child.destroy()

        self.destroy()

    # Opens window of the type specified by argument with entered title

    def open\_window(self, title, type):

        window = Window(self, title, type)

        window.resizable(False,False) # Disables ability to resize windows to maintain cohesive appearance

        if type == "exit" or type == "options" or type == "pauseMenu" or type == "blocker" or type == "moveConfirm":  # If critical window, forces user interaction

            window.grab\_set()

        return window

    def close\_confirm(self):  # Creates exit confirmation window

        self.open\_window("Confirm exit?", "exit")

    def set\_geometry(self,width,height,x=-1,y=-1): # Set position and dimensions of window

        w = width # Window width in pixels

        h = height # Window height in pixels

        # Get screen width and height

        ws = root.winfo\_screenwidth() # width of the screen

        hs = root.winfo\_screenheight() # height of the screen

        # Calculate x and y coordinates for window

        if x == -1: # If no specific location given, center to screen horizontally

            x = (ws/2) - (w/2)

        if y == -1: # If no specific location given, center to screen vertically

            y = (hs/2) - (h/2) - 50

        # Set the dimensions and location of window

        self.geometry('%dx%d+%d+%d' % (w, h, x, y))

    '''def close\_confirm(self):  # Creates confirmation window

        #confirm = messagebox.askokcancel(title="Confirm exit?",message="Save and close all windows?")

        if messagebox.askokcancel(title="Confirm exit?",message="Save and close all windows?"):

            for child in root.winfo\_children(): # Closes all windows, including hidden

                child.destroy()

            root.destroy()'''

    # If game in progress is continued, reads info from external txt file in which save game data is stored

    def load\_save(saveName):

        saveFile = open(saveName, "r") # Opens appropriate save data txt file

        for line in saveFile:

            if "playerCount" in line: # Reads [example] from save data txt file and assigns to global variable

                global playerCount

                playerCount = line[-1:]

                continue

# Creates and hides the initial Tk() entity

if \_\_name\_\_ == "\_\_main\_\_":

    root = Root()

    mainMenu = root.open\_window("Main Menu", "mainMenu") # Create main menu window

    root.mainloop()