Milestone 1: Create Chess Board

Objective: Create a chess board that can have a varying number of squares per column/row. The board should have alternating clickable white and black squares

Success Criteria: This Milestone covers the following success criteria:
1.1, 1.2, 1.3

Goal 1: Create a Chess Square for the Board

The initial goal of this milestone is to create a square. Because many chess squares need to be made, I decided to use a class for the chess square, so that I would not have to repeat similar code many times.

```
class ChessSquare:
    def __init__(self, x, y, width, height):
        self.x = x
        self.y = y
        self.width = width
        self.height = height
        self.position = (x,y)
        self.occupied_by = True
        self.absolute_x = x*width
        self.absolute_y = y*height
        self.absolute_coord = (self.absolute_x, self.absolute_y)
        self.change_color = False
        self.optional_space_color = (0,0,255)
        self.name = (chr(x+64) + str(y))
        self.optional_space = False
        self.rect = pygame.draw.rect(screen, (255,255,255), pygame.Rect(self.absolute_x, self.absolute_y, self.width, self.height))
```

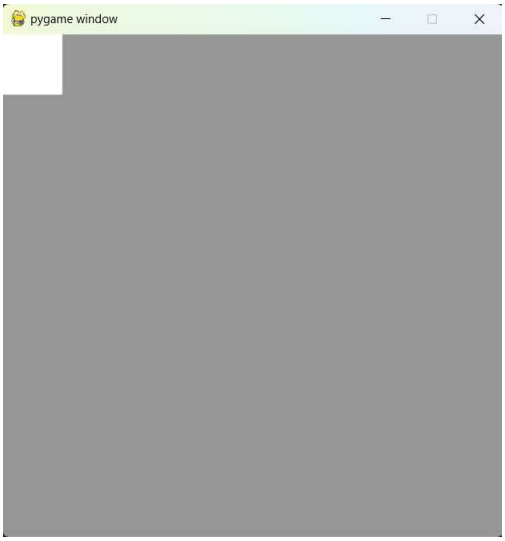
Note: These are the initial attributes of the class, but throughout my project it became necessary to add more attributes.

When I first created the chess square, I tried various values for the height and width of the square, and decided, after some experimentation, that 60 pixels resulted in a good sized display for an 8 by 8 chess board. With this setting, the square was large enough to be easily seen, and it allowed for many more squares to be displayed on the board.

```
ChessSquare(0, 0, 60, 60)
```

The first two numbers define the coordinates of the top left corner of the square, and the last two numbers represent the dimensions of the square (measured in pixels).

The output is as follows.



Goal 2: Create Squares of Alternating Colours

The second goal is to create multiple squares in alternating colours. To do this, I first created the "self.color" attribute, which uses the x and y coordinates of the square to determine if the square should be black or white. The coordinates of the white squares are: (0,0), (0,2), (0,4), (0,6), (1,1), (1,3), etc. The sum of the coordinates for white squares is always a multiple of two. When creating a square, I used the modulus 2 operator to check this. If the sum is a multiple of two, then the square is white, otherwise it must be black.

```
self.color = "white" if (self.x + self.y)%2 == 0 else "black"
```

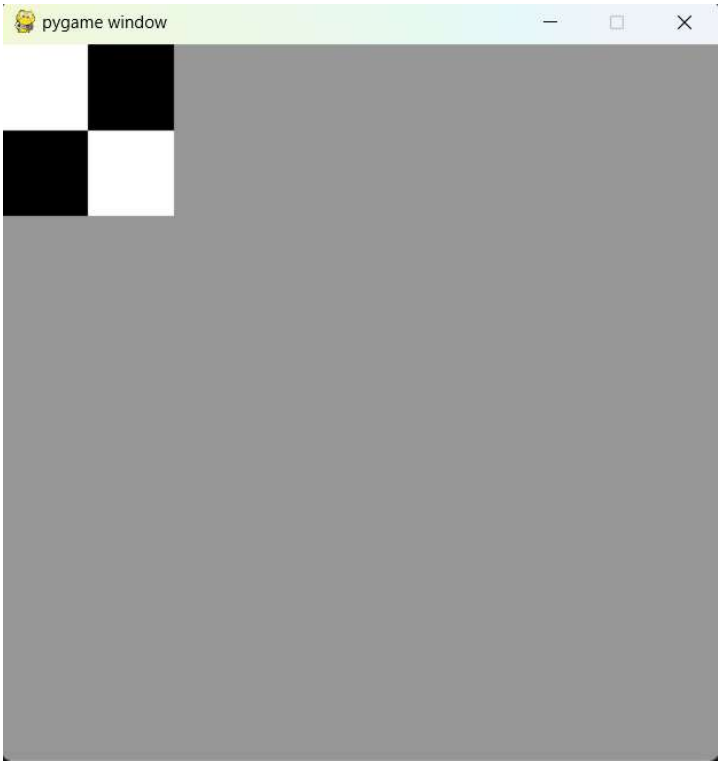
Pygame uses RGB values to define colours. The RGB value for white is (255, 255, 255), and for black it is (0, 0, 0).

```
self.draw_color = (255, 255, 255) if self.color == "white" else (0, 0, 0)
```

Within the line that defines the colour of the square, I changed the value (255, 255, 255) to self.draw_color, so that squares could be created either as black or white.

```
self.rect = (pygame.draw.rect(screen, self.draw_color,
pygame.Rect(self.absolute_x, self.absolute_y, self.width, self.height)))
```

I then created a few instances of the ChessSquare class, to ensure that this was successful.



Goal 3: Create an Entire Chess Board

Once I had verified that multiple squares could be created correctly, I needed a method to create a large number of squares more efficiently. Originally I created a class “ChessBoard” that inherits the attributes of “ChessSquare”. The new class had a method that contained two loops (each with a range of 8) that created the chess board. Each time a chess square was created, I used the vars() function in an attempt to make it possible to access the instances of the class later in the code.

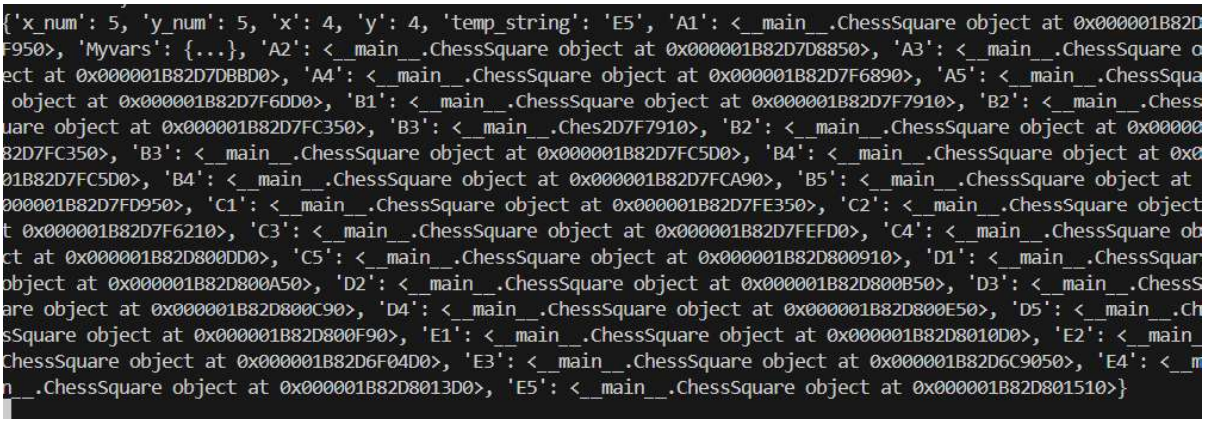
The original ChessBoard function is shown below.

```
class ChessBoard(ChessSquare):
    def __init__(self, x, y, width, height, position, absolute_x, absolute_y, absolute_coord,
                 change_color, color, draw_color, optional_space_color, optional_space, name, x_num, y_num):
        super(ChessSquare, self, x, y, width, height, position, absolute_x, absolute_y, absolute_coord,
              change_color, color, draw_color, optional_space_color, optional_space, name).__init__()
        self.x_num = x_num
        self.y_num = y_num
    def create_board(x_num, y_num):
        for x in range(x_num):
            for y in range(y_num):
                temp_string = chr(x+65) + str(y+1)
                Myvars = vars()
                Myvars.__setitem__(temp_string, ChessSquare(x, y, 60, 60))
                print(Myvars)
mouse_coord = pygame.mouse.get_pos()
```

An instance of the ChessBoard class is shown below.

```
board_1 = ChessBoard.create_board(8, 8)
```

When using the vars function, I tried to store the instances of the squares in a list, as this would allow me to access the instances of the class along with their attributes later when needed. However, the contents of the list were unmanageable, and it proved very difficult to access the attributes of the class. The contents of the list were as follows.



The reason that the vars() function was unmanageable was because when trying to retrieve the instance of the class, it would instead return the location in memory of the instance. This meant that retrieving the attributes from each instance was not possible.

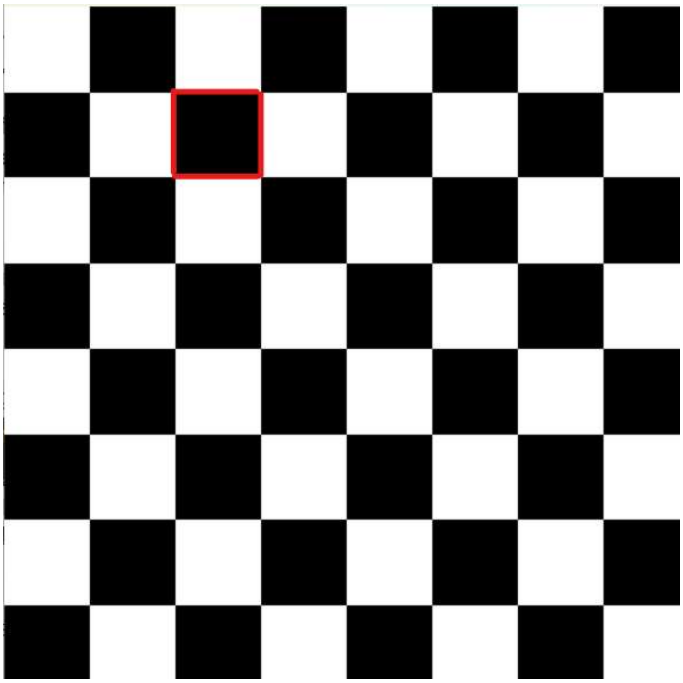
As the vars() function was difficult to work with, I decided on a new approach. I decided to create a single loop with the size of the board as the parameter, and change the ChessBoard function so that it could determine the required coordinates of the square based on the iteration in the loop.

I created a variable called Size_of_Board, that defines the number of squares in the board. Because the board is always a square, it only needs one variable to define its size.

The beginning of the loop is shown below.

```
for i in range(Size_of_Board):
    Board_Location = i
```

To determine the coordinates of a particular square, I used the following method. The top left square has a board location of 0. Moving from left to right, it increments by 1, and continues to increment as it moves to the next row. The square highlighted in red therefore has a board location of 10. To determine the x coordinate of the square, I used the modulus 8 function. 10 modulus 8 results in 2, which is the x coordinate of the square. In this example, 8 is used, as it is the number of squares per row and column on the board shown below. To find the y coordinate, I used 10 / 8, which is division without a remainder. 10 / 8 returns 1.



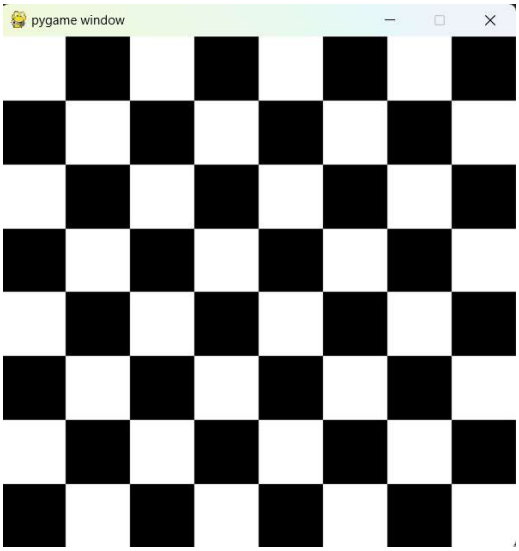
The logic for determining the coordinates of the square is shown below.
When determining the number to modulus and divide by, I used the square root of the Size_of_Board variable.

```
ChessSquare(Board_Location % int(math.sqrt(Size_of_Board)),
            int(Board_Location/int(math.sqrt(Size_of_Board))))
```

I also learned that an instance of a class can be stored within a list. This allowed me to store the instances of the ChessSquare class within a list called Square_Name_List. I did this because it would enable me to easily access the attributes of the different instances of the class throughout my code.

```
Square_Name_List[Board_Location] = ChessSquare(
```

Below is a standard 8 by 8 chess board that I was able to create using this code.

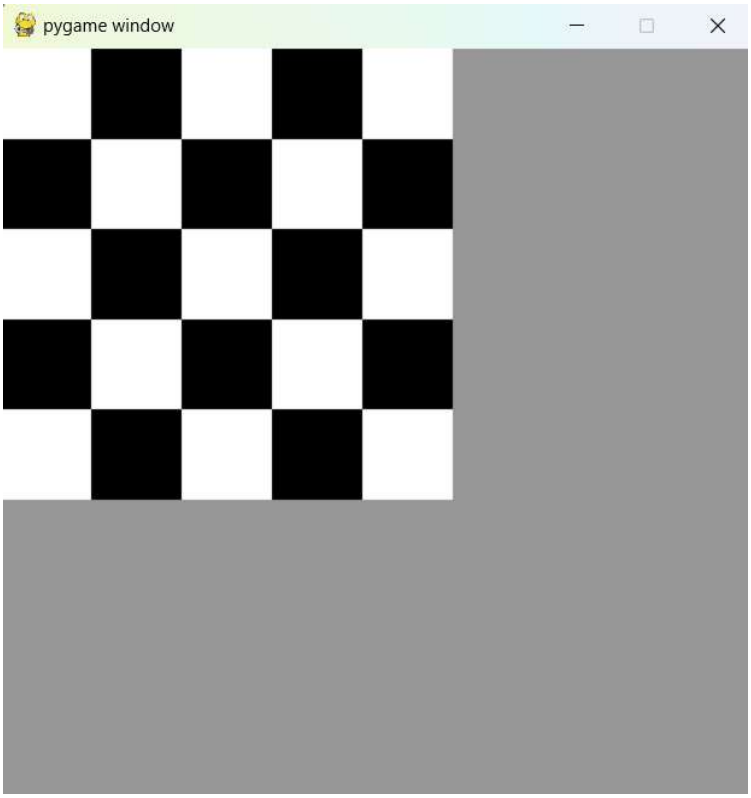


Goal 4: Allow the Chess Board to Change Sizes

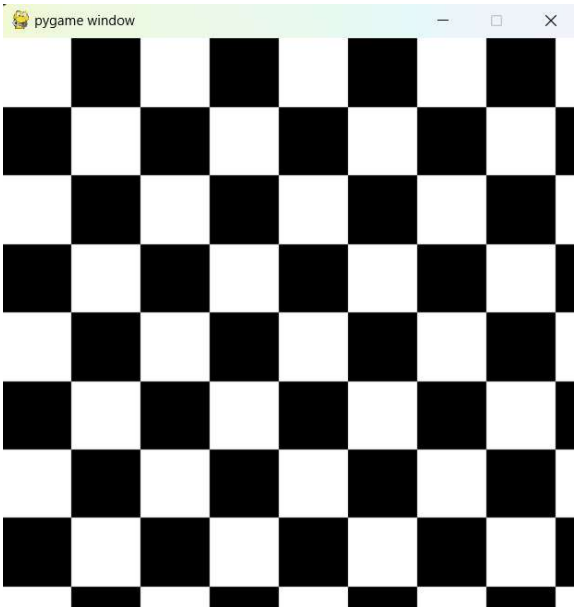
After verifying that a chess board with normal dimensions could be created, I needed to test chess boards with different dimensions. I did this by altering the Size_of_Board variable.

Board display Error

When choosing a smaller number for Size_of_Board, the board squares were displayed correctly, however there was a large grey border.



However, when choosing a larger value for Size_of_Board, the full board was not displayed on the screen.



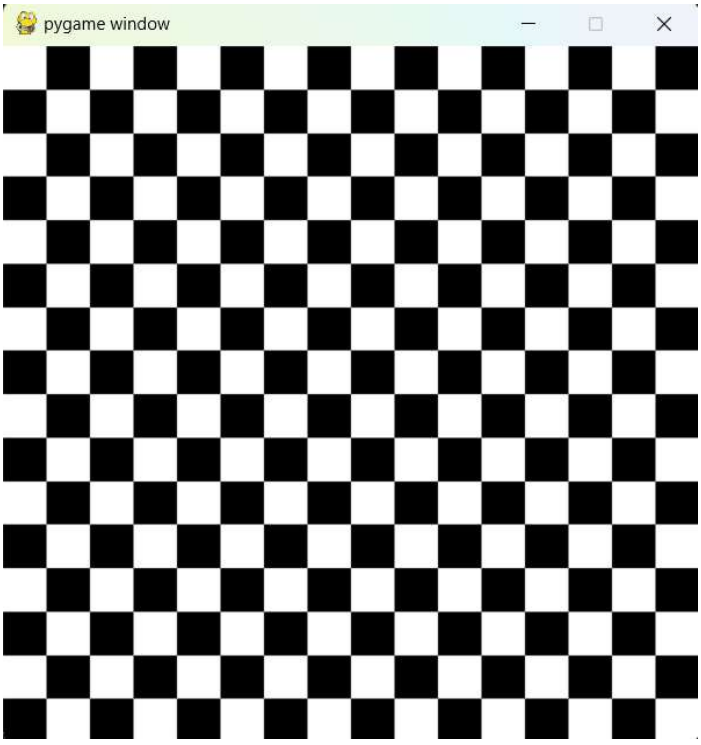
Fixing the Error

The problem was that I originally created the chess squares with preset widths and heights of 60 pixels. The size of the screen doesn't change with the size of the board, so the width and height of each square needs to be calculated based on the number of squares per row and column, in relation to the size of the screen. I am doing this because it allows the board to fill the entire screen regardless of how many squares there are per row and column. I used the code highlighted in red to define the width and height of the square, to ensure that the entire board could be shown on the screen.

```
ChessSquare(Board_Location % int(math.sqrt(Size_of_Board)),
            int(Board_Location/int(math.sqrt(Size_of_Board))),
            screen.get_width()/int(math.sqrt(Size_of_Board)),
            screen.get_height()/int(math.sqrt(Size_of_Board)), 0)
```

Using the pre-built “get_width” and “get_height” functions, I was able to retrieve the dimensions of the screen, and by dividing those values by the number of squares per row and column (i.e. the square root of Size_of_Board), I was able to calculate the required dimensions of the squares.

Below is an example of a board with 256 squares.



Module Testing

Test Number	Success Criteria	Input	Output	Expected Output	Outcome
1	1.1	As of milestone 1, there is no user input for the size of board, however at milestone 6, the user will be able to input values for the size of board.	Shown on Page 96	8 by 8 board of squares.	No changes required.
2	1.2	Because the square colours are defined as white or black depending on their location on the board, there is no user input for this success criteria.	Shown on Page 92	Squares alternate between being black and white.	No immediate changes, but the black squares were made to be grey to make black pieces visible on them during milestone 2.
3	1.3	Because the placement of images on the screen is an internal function, there is no user input for this success criteria.	There is no output as this success criteria only enables pieces to be placed.	Pieces are able to be placed on the squares.	No changes required.

Reflection:

I am pleased with the outcome of this milestone as it has been fully completed. I was able to complete every success criteria and I was particularly pleased with how I was able to display the variable size of the board whilst fully covering the screen.

Milestone 2: Piece Representation

Success Criteria: This Milestone covers the following success criteria:
2.1, 2.2, 2.3

Goal 1: Obtain Piece Images

Initially, I required images of the standard pieces in chess that will represent the pieces in my code. To find these, I downloaded the following images from the internet into a folder on my computer.

Goal 2: Allow the Code Access to the Images

Once I had acquired the images, I needed to be able to access the images from my computer and put them onto the screen. Using the location of the images on my computer, I attempted to access the images.

```
Piece_Name_Background = pygame.image.load("C:\Users\pango\OneDrive\Pictures\ " + Piece_Name + ".png")
```

File Path Error

Unfortunately, when trying to access these images, there was a syntax error. After some research, I discovered that the backslash symbol "\" (which is part of the path to locate the files) is the escape character in Python, which means that the function of the character following the backslash is disregarded. This invalidated my line of code, as python was interpreting the directory locations as commands, and was generating an error as a result.

```
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in position 2-3: truncated
\UXXXXXXXX escape (file:///c%3A/Users/pango/OneDrive/Desktop/Python/second.py, line 63) compile
```

Fixing the Error

To fix this issue, I used a double backslash (to "escape" the directory path backslash character), so that Python can interpret the path of the image as required.

```
Piece_Name_Background = pygame.image.load("C:\\Users\\pango\\OneDrive\\Pictures\\" + Piece_Name + ".png")
Piece_Name_Background.set_colorkey((0, 0, 0))
```

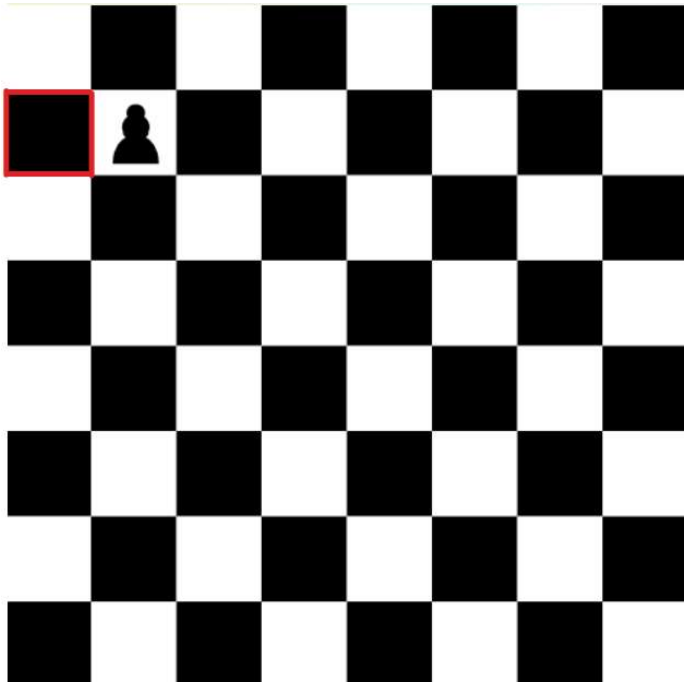
Goal 3: Sizing and Placement of Images

The piece images needed to have the same dimensions as the squares, because the images need to fill the entire square and not overlap into adjacent squares. The images need to be centred on the squares for aesthetic purposes. To make their dimensions the same, I used the width and height dimensions of the square that the piece was on as the parameters for the piece's dimensions, as this ensures the pieces would always be the same size as the square they are located on.

```
Piece_Name_Background = pygame.transform.scale(Piece_Name_Background, (Square_Name_List[Board_Location].width,
                                                                       Square_Name_List[Board_Location].height))
screen.blit(Piece_Name_Background, (Square_Name_List[Board_Location].absolute_coord))
```

The pygame.transform.scale sets the size of the image of the piece. The pygame blit() method was used to place the image on the defined coordinates.

Pawn Display Error



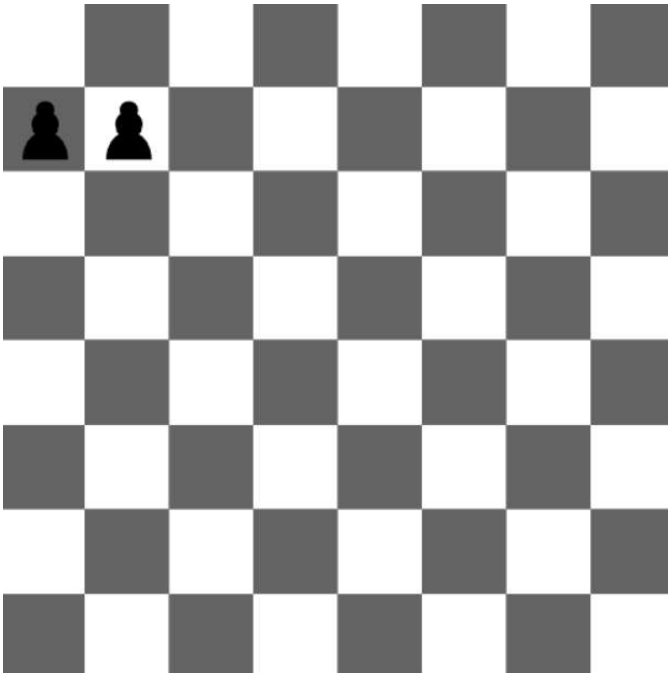
Above is an image of two black pawns on the board. The pawn on the white square displays correctly, however because the colour of the black square is identical to the colour of the pawn, the pawn on the highlighted square is not visible.

Fixing the error

To fix this, I altered the colour of the black squares to be grey rather than black so that the black pawns were still visible.

The value (0, 0, 0) is pure black, and I changed it to (100, 100, 100), which is grey, which allows for the black pawns to be easily seen on the “black” squares.

```
self.draw_color = (255, 255, 255) if self.color == "white" else (100, 100, 100)
```



After being able to display pawns successfully, I needed to be able to display the images for all 32 pieces. Instead of doing this individually, I changed the ChessBoard class so that it could also put a piece on the square when creating the square. I also changed the name of the ChessBoard class to Create_Chess_Piece.

```
def Create_Chess_Piece(Piece_Name, Board_Location, Normal_Add, Blue_Add):
    #Pieces.extend([Square_Name_List[i], Piece_Name])
    Square_Name_List[Board_Location] = ChessSquare(Board_Location % int(math.sqrt(Size_of_Board)),
                                                   int(Board_Location/int(math.sqrt(Size_of_Board))),
                                                   screen.get_width()/int(math.sqrt(Size_of_Board)),
                                                   screen.get_height()/int(math.sqrt(Size_of_Board)), 0)

    pygame.display.set_caption(Piece_Name)
    Piece_Name_Background = pygame.image.load("C:\\Users\\pango\\OneDrive\\Pictures\\" + Piece_Name + ".png")
    Piece_Name_Background.set_colorkey((0, 0, 0))
    Piece_Name_Background = pygame.transform.scale(Piece_Name_Background, (Square_Name_List[Board_Location].width,
                                                                           Square_Name_List[Board_Location].height))
    screen.blit(Piece_Name_Background, (Square_Name_List[Board_Location].absolute_coord))

    Piece_Locations.extend([Square_Name_List[Board_Location].absolute_x, Square_Name_List[Board_Location].absolute_y,
                            Square_Name_List[Board_Location].absolute_x + screen.get_width()/int(math.sqrt(Size_of_Board)),
                            Square_Name_List[Board_Location].absolute_y + screen.get_width()/int(math.sqrt(Size_of_Board))])
    Piece_Board_Locations.append(Board_Location)
    pygame.display.update()
```

I also created two lists. The first one is called Piece_Board_Locations, which contains the relative numerical position of each square (for an 8 by 8 board the numbers would be 0 to 63). I did this so that each square could be easily identified. The second list is called Piece_Locations, representing the coordinates of the piece (which are the same as the coordinates of the square that the piece is on). This was done so that the coordinates of each piece could be easily retrieved.

Because the initial relative positions of the pieces will remain the same regardless of the board size (e.g. the rooks always start in the corners of the board), I was able to use the following logic to define where the pieces will be placed on the board. The logic is explained in the inline comments below (in green).

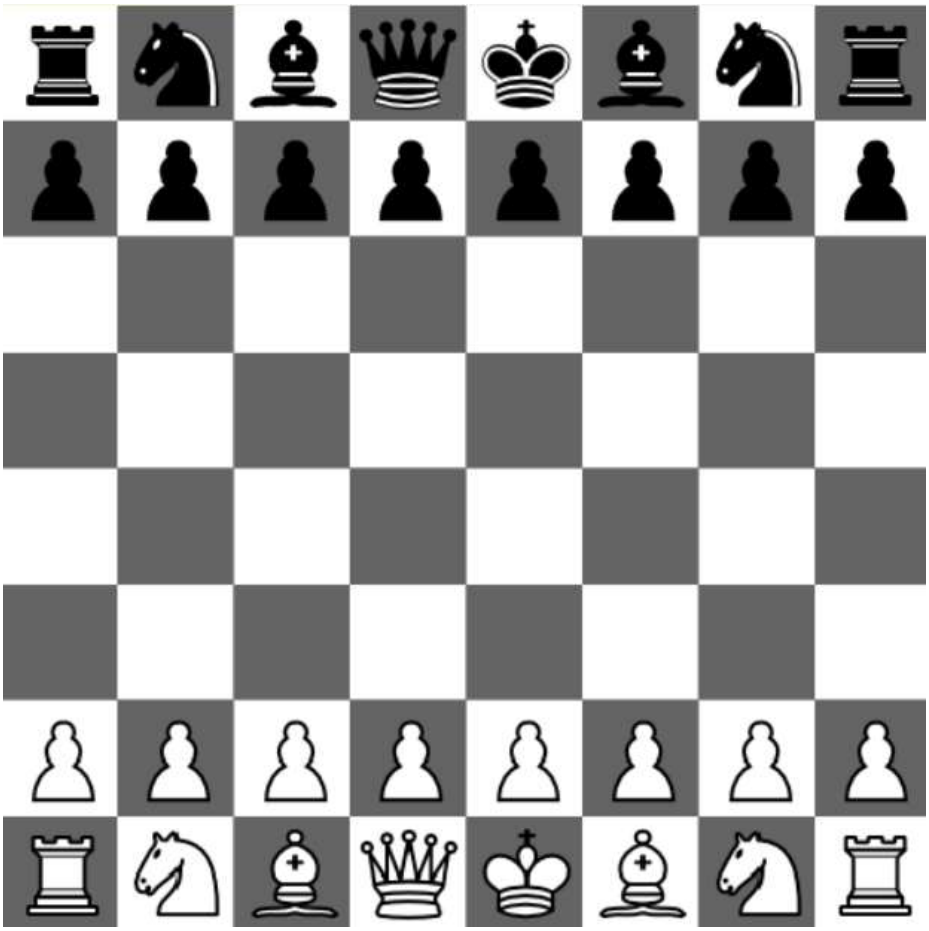
```
for i in range(Size_of_Board):
    Board_Location = i
    Square_Location = i

    # if the square is on the second line from the top of the board, place a black pawn on the square
    if i >= int(math.sqrt(Size_of_Board)) and i < 2*(int(math.sqrt(Size_of_Board))):
        Create_Chess_Piece("Black_Pawn", Board_Location, True, True, True, Board_Location, 0, 0)

    # if the square is on the second line from the bottom of the board, place a white pawn on the square
    elif (i >= (int(math.sqrt(Size_of_Board))-2)*int(math.sqrt(Size_of_Board)) and
          i < (int(math.sqrt(Size_of_Board))-1)*int(math.sqrt(Size_of_Board))):
        Create_Chess_Piece("White_Pawn", Board_Location, True, True, True, Board_Location, 0, 0)

    # if the square is the top right or top left square, place a black rook on the square
    elif i == 0 or i == int(math.sqrt(Size_of_Board))-1:
        Create_Chess_Piece("Black_Rook", Board_Location, True, True, True, Board_Location, 0, 0)

    # if the square is the bottom right or bottom left square, place a white rook on the square
    elif i == Size_of_Board -1 or i == (int(math.sqrt(Size_of_Board))-1) * int(math.sqrt(Size_of_Board)):
        Create_Chess_Piece("White_Rook", Board_Location, True, True, True, Board_Location, 0, 0)
```

```
# if the square is just to the right of centre on the top row, place a black king on the square
elif i == round((int(math.sqrt(Size_of_Board)))/2):
    Create_Chess_Piece("Black_King", Board_Location, True, True, True, Board_Location, 0, 0)

# if the square is just to the right of centre on the bottom row, place a white king on the square
elif i == Size_of_Board - round((int(math.sqrt(Size_of_Board)))/2):
    Create_Chess_Piece("White_King", Board_Location, True, True, True, Board_Location, 0, 0)

# if the square is just to the left of centre on the top row, place a black queen on the square
elif i == round((int(math.sqrt(Size_of_Board)))/2)-1:
    Create_Chess_Piece("Black_Queen", Board_Location, True, True, True, Board_Location, 0, 0)

# if the square is just to the left of centre on the bottom row, place a white queen on the square
elif i == Size_of_Board - round((int(math.sqrt(Size_of_Board)))/2) - 1:
    Create_Chess_Piece("White_Queen", Board_Location, True, True, True, Board_Location, 0, 0)
```

The logic for creating the squares without pieces on them (i.e. the middle rows of the board) is shown below.

```
#if a square does not have a piece on it an empty ChessSquare is created and its coordinates are added to Square_Coord_Edges
if i >= 2*int(math.sqrt(Size_of_Board)) and i < (int(math.sqrt(Size_of_Board))-2) * int(math.sqrt(Size_of_Board)):
    Square_Name_List.append(ChessSquare(i % int(math.sqrt(Size_of_Board)), int(i/int(math.sqrt(Size_of_Board))),
                                        (screen.get_width()-100)/int(math.sqrt(Size_of_Board)),
                                        (screen.get_height()-100)/int(math.sqrt(Size_of_Board)), 0, 0, "Empty", 0))
    Square_Coord_Edges.extend([Square_Name_List[i].absolute_x, Square_Name_List[i].absolute_y,
                               Square_Name_List[i].absolute_x + (screen.get_width()-100)/int(math.sqrt(Size_of_Board)),
                               Square_Name_List[i].absolute_y + (screen.get_height()-100)/int(math.sqrt(Size_of_Board))])
```

Below is a completed 8 by 8 chess board with all of the pieces placed in the correct initial positions.



Board Size Issue

Because the board width and height can only be a multiple of two (i.e 4 by 4, 6 by 6, 8 by 8, 10 by 10 etc.), and the decision has been made to maintain left-right symmetry, it is only possible to place bishops and knights if Size_of_Board is greater than 36 (i.e. greater than 6 by 6). This is because the king, queen and and both rooks for each player take priority, and they take four squares. This means that when the board is a 6 by 6, there is room for either bishops or knights, but not both.

The following code shows that knights and bishops will both be placed only when Size_of_Board is greater than 36.

```
# if the dimensions of the board are greater than 6 by 6, then knights and bishops can be placed on the board
if Size_of_Board > 36:

    # if the square is second from left or right on the top row, place a black knight on the square
    if i == 1 or i == int(math.sqrt(Size_of_Board))-2:
        Create_Chess_Piece("Black_Knight", Board_Location, True, True, True, Board_Location, 0, 0)

    # if the square is second from left or right on the bottom row, place a white knight on the square
    elif i == Size_of_Board -2 or i == ((int(math.sqrt(Size_of_Board))-1) * int(math.sqrt(Size_of_Board)) + 1):
        Create_Chess_Piece("White_Knight", Board_Location, True, True, True, Board_Location, 0, 0)

    # if the square is third from left or right on the top row, place a black bishop on the square
    elif i == 2 or i == int(math.sqrt(Size_of_Board))-3:
        Create_Chess_Piece("Black_Bishop", Board_Location, True, True, True, Board_Location, 0, 0)

    # if the square is third from left or right on the bottom row, place a white bishop on the square
    elif i == Size_of_Board -3 or i == (int(math.sqrt(Size_of_Board))-1) * int(math.sqrt(Size_of_Board)) + 2:
        Create_Chess_Piece("White_Bishop", Board_Location, True, True, True, Board_Location, 0, 0)
```

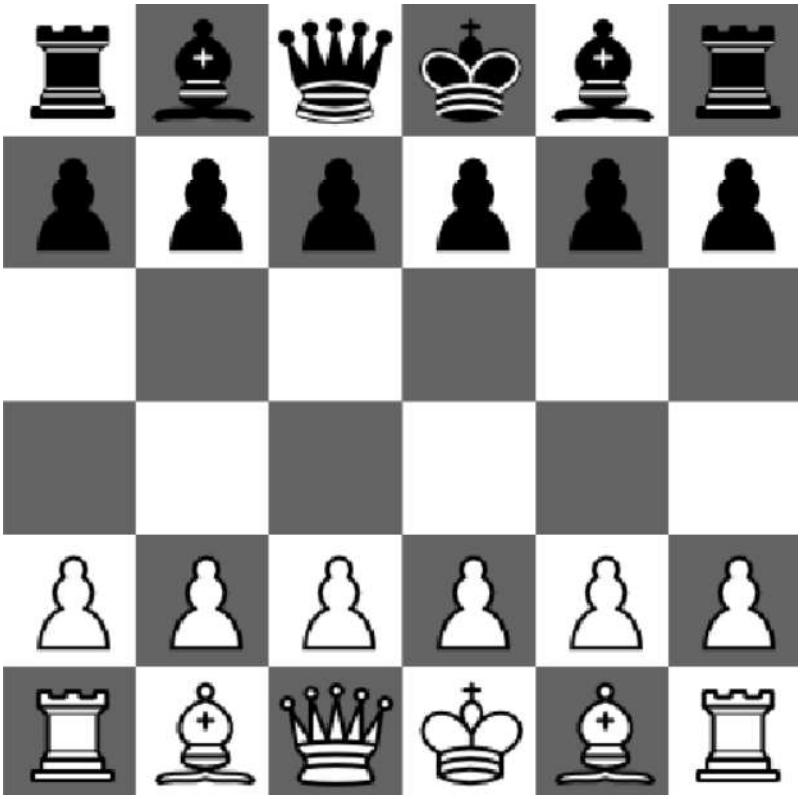
Knights or Bishops Decision

Because both knights and bishops could not be placed on a 6 by 6 board, a decision had to be made for whether bishops or knights would be placed.

I asked my stakeholders whether they would prefer for knights or bishops to be present on a 6 by 6 board, and three of them (Dean, Zayan, and Ben) stated that they would prefer for bishops to be present, and Louis stated that he would prefer knights. Because the majority of my stakeholders would prefer bishops rather than knights, I prioritised bishops.

The following code was used to create bishops on a 6 by 6 board, along with the result when Size_of_Board is set to 36.

```
if Size_of_Board == 36:
    if i == 1 or i == 4:
        Create_Chess_Piece("Black_Bishop", Board_Location, True, True, True, Board_Location, 0, 0)
    if i == 31 or i == 34:
        Create_Chess_Piece("White_Bishop", Board_Location, True, True, True, Board_Location, 0, 0)
```



Goal 4: Clickable Pieces

In order to make the pieces clickable, I first needed the position of the mouse to be recorded when I clicked on the screen, as this will allow me to determine in which square the user has clicked. I was able to use a prebuilt pygame function to do this.

```
mouse_coord = pygame.mouse.get_pos()
```

The next step was determining the range of coordinates within each chess square. The ChessSquare class already contained the attributes “absolute_x” and “absolute_y”, which are the coordinates on the screen of the top left corner of each square. I was able to add the width and and height of the chess square to these coordinates in order to calculate the coordinates of the bottom right corner of each square. I created a list called Square_Coord_Edges, that contained the x and y coordinates of the top left corner and bottom right corner of every square i.e. four entries per square.

```
Square_Coord_Edges = []
```

A section of the contents of Square_Coord_Edges is as follows, where the four highlighted values represent the coordinates of the top left square (the first two represent the x and y coordinate of its top left corner and the last two represent the coordinates of its bottom right corner).

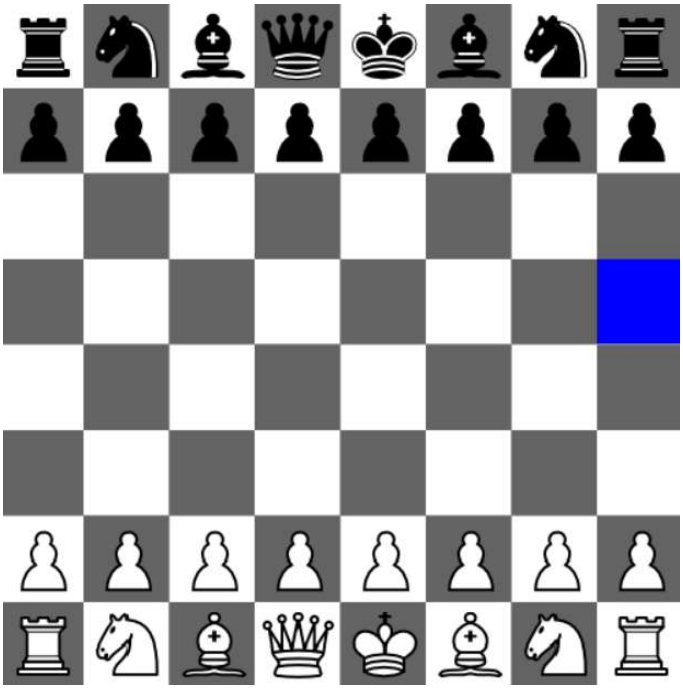
```
0.0, 0.0, 62.5, 62.5 62.5, 0.0, 125.0, 62.5, 125.0, 0.0, 187.5, 62.5, 187.5, 0.0, 250.0, 62.5, 250.0, 0.0,
, 62.5, 312.5, 0.0, 375.0, 62.5, 375.0, 0.0, 437.5, 62.5, 437.5, 0.0, 500.0, 62.5, 0.0, 62.5, 62.5, 125.0, 6
2.5, 125.0, 125.0, 125.0, 62.5, 187.5, 125.0, 187.5, 62.5, 250.0, 125.0, 250.0, 62.5, 312.5, 125.0, 312.5, 6
75.0, 125.0, 375.0, 62.5, 437.5, 125.0, 437.5, 62.5, 500.0, 125.0, 0.0, 125.0, 62.5, 187.5, 62.5, 125.0, 125
```

In order to determine whether a piece has been clicked, I added two attributes to the ChessSquare class called optional_space_color and change_colour. “change_color” is an attribute that is initialised when the instance of the class is created. This can alter the colour of a square from its default colour to blue. The colour is changed depending on whether change_color is even or odd. If change_color is even then the colour of the square is blue, and if it is odd the colour of the square is set to its default colour (black or white). This will be used to show when a square has been clicked.

```
def __init__(self, x, y, width, height, change_color):
```

```
self.change_color = change_color
self.optional_space_color = (0,0,255)
self.draw_color = (self.optional_space_color if self.change_color %2 != 0 else
(255, 255, 255) if self.color == "white" else (100, 100, 100))
```

An example of a highlighted square is as follows.



The top line of the following code is used to associate the variable “left” with a left mouse click. Whenever the left mouse button is clicked, the coordinates of the click on the screen are retrieved and stored in a list click_pos_list. This list contains two entries, representing the x and y coordinates of the clicked location. I am storing the clicked coordinates in a list because this makes it easier to access them and compare them with other sets of coordinates.

```
while running == True:
    for event in pygame.event.get():
        left = pygame.mouse.get_pressed()
        if left:
            click_pos = pygame.mouse.get_pos()
            click_pos_list = list(click_pos)
```

I used the following code to loop through the coordinates of each individual piece and store them. The coordinates of the clicked location are checked against each set of stored coordinates, until the clicked square is found. Because Square_Coord_Edges has four entries per square, the modulus 4 operation is used to loop through each square.

```
if i % 4 == 0:
    temp_coord_store = []
    second_temp_coord_store = []
    temp_coord_store.append(Square_Coord_Edges[i])
    temp_coord_store.append(Square_Coord_Edges[i+1])
    second_temp_coord_store.append(Square_Coord_Edges[i+2])
    second_temp_coord_store.append(Square_Coord_Edges[i+3])
```

If the values in click_pos_list are between the two sets of coordinates for a given square, then the value of change_color for that square is incremented by one, and the square is recreated with the newly changed colour. The screen is then refreshed (this updates the screen so that any changes made to it are visible to the user).

```
if (click_pos_list[0] >= temp_coord_store[0] and click_pos_list[1] >= temp_coord_store[1] and
click_pos_list[0] <= second_temp_coord_store[0] and click_pos_list[1] <= second_temp_coord_store[1]):
    Square_Name_List[int(i/4)].change_color += 1
    Square_Name_List[int(i/4)] = (ChessSquare(Square_Name_List[int(i/4)].x,
    Square_Name_List[int(i/4)].y, screen.get_width()/int(math.sqrt(Size_of_Board)),
    screen.get_height()/int(math.sqrt(Size_of_Board)), Square_Name_List[int(i/4)].change_color))
    pygame.display.update()
```

Once the clicked square is identified, I needed to check if there was a piece on that square. This means that I need to know the coordinates of the pieces.To do this, I defined a new list called “Piece_Locations”. Whenever a piece is created, the coordinates of its square are added to this list. This allows for all of the pieces’ coordinates to be easily checked.

```
Piece_Locations = []
```

Now that every piece has a stored set of coordinates, once it has been determined which square is clicked, the coordinates of all of the pieces can be looped through to see if there is a piece on the square.

Each piece’s png file is actually a square with a transparent background behind the piece. When placed on the chess board, the colour of the square behind the piece remains visible.

Even though the backgrounds of the pieces are transparent, I was originally unable to show the pieces and the blue changed background at the same time. Effectively, the square never changed colour when I clicked it. To work around this issue, I created new images of each piece with a blue background. The new png files are stored in the same file location, and the names are the same as the original files with the addition of "_Blue" added to the end of the file name. The new blue "clicked" squares appear on the board as follows.

When a piece is clicked, if it has its default (black or white) background, then it is swapped to the version of that piece with the blue background, and if it has a blue background, it is swapped for the piece with the normal background. This is done so that it is obvious which piece has been selected. The following code implements the colour change.

```
for j in range(len(Piece_Board_Locations)):
    if (click_pos_list[0] >= Piece_Locations[4*j] and click_pos_list[1] >= Piece_Locations[4*j+1] and
    click_pos_list[0] <= Piece_Locations[4*j+2] and click_pos_list[1] <= Piece_Locations[4*j+3]):
        if Square_Name_List[int(i/4)].change_color % 2 == 1:
            Create_Chess_Piece(Pieces_Blue[int(j)], int(i/4), False, True)
            Square_Name_List[int(i/4)].change_color += 1
            pygame.display.update()
        else:
            Create_Chess_Piece(Pieces[int(j)], int(i/4), True, False)
            pygame.display.update()
        break
```

Within the Create_Chess_Piece function, I added two new attributes, Normal_Add and Blue_Add. If Normal_Add is true, then an instance of the class ChessSquare is created using the image with a default background (black or white). If Blue_Add is true, then an instance of the class is created using the image with a blue background.

```
if Normal_Add == True:
    Pieces.append(Piece_Name)
    Pieces_Unique.append([Piece_Name, Board_Location])
if Blue_Add == True:
    if "_Blue" in Piece_Name:
        Piece_Name = Piece_Name.replace("_Blue", "")
    Pieces_Blue.append(Piece_Name + "_Blue")
    Pieces_Blue_Unique.append([Piece_Name + "_Blue", Board_Location])
```