Chess Project

For my project I created a game of Chess using Pygame within the programming language of Python. This was my first-time using Python to code with, so getting past that first barrier of entry was the first difficult task. However, after weeks and months of working on this project, I believe I understand Python well enough to submit a suitable senior project. For the project itself, I split the project into two files: one file that houses the variables and another file that houses the main game loop of chess, including the functions for drawing the board and pieces, how the pieces should move, and how to end the game. I will go into detail on the breakdown of each of these files, starting with the shorter Variables.py file.

Starting with the Variables.py file, I initiate the file with importing pygame so I can use all the features pygame has to offer. I start with initializing the variables for the game. This includes the height and width of the screen, creating two surfaces for the game to be ran on. The first “screen” is the initial display that the game will use and the second “surface” being a surface that allows for items to be drawn on top of the initial screen while remaining transparent. I then included 4 different sized fonts to be used in various places throughout the game. I then set my caption of the screen, included a timer in the game that uses a specified fps variable. Finally, I set the color of the background and created a mainMenu variable that was set to true, which allowed for my soon to be created main menu to be displayed when you start up the game. Once all of the initialization of the screen was set, I then started to incorporate items used for chess. This included a pieces [] that contained the names of all of the pieces. This was then broke down by color, first creating a whitePieces [] that contained all the names of the pieces that white would use, including 8 pawns, 2 rooks, 2 knights, 2 bishops, a queen and a king. One by one I created variables for each white piece by loading a png of the piece into the game, shrinking it to fit on the board within a square, and creating a smaller sized png that would be displayed in a column of captured pieces. I then repeated this for each piece of the whitePieces. After this, I created a variable whitePictures to have an array of the specific pieces for example: [whiteRook, whiteKnight, whiteBishop, whiteQueen, whiteKing, whitePawn]. I repeated for a variable that included whitePiecesCaptured. Finally, I created the whitePromotions variable to house what pieces a pawn will be able to promote to when the game is created. With all of this set up completed, I was able to create a whiteMoved [] whitePieces [] replacing each piece name to “False” indicated the piece has yet to move from its starting position. I then recreated the same set up for a whitePiecesLocation [] to instead house the coordinates of where these pieces would start; with (0,0) being the top leftmost square and (7,7) being the bottom rightmost square. Finally, I created a whiteCaptured [] that was empty, which would fill when a piece was captured. With all of this set up, I then repeated all of this the exact same but this time in terms of blackPieces.

With the pieces now set up, I then loaded images that would be used in the tutorial page. This includes arrows for turning the pages of the tutorial and images of the special moves and how to perform them. I then added sound effects for pieces moving, pieces being captured, a sound to notify when a king is in check, a victory sound that played when the game ended, and a sound for when a page was turned I the tutorial. With all of the initial image and sound set up out of the way, I was finally able to initialize the variables controlled in playing the game. This included a multitude of variables including: a turn variable set to 0, what would increment to 3 throughout the game; 0 being white’s turn to select a piece, 1 being white’s turn to select a location for the piece to move, 2 being black’s turn to select a piece, and 3 being black’s turn to select a location for the piece to move. This would then cycle back to 0 to allow white to move again. Next, a selection variable set to -1 was used to identify what piece a player selected. A possibleMoves[] was created to hold what moves are valid per piece when the piece was clicked (which caused issues later down the road of the game). A counter variable that would work with a timer variable used with the fps and the ticking of the “check box” used to notify the player their king is in check. A gameOver variable set to False that would update to True when a player was in apposition to win the game. enPassant variables for each color that controlled when a Pawn could En Passant. Promote variables for each color that controlled when a Pawn could promote, including a promoteIndex. An inCheck variable that was global initially set to False that would change to True when a king was in check. A castleMoves [] that housed the moves that could be made to castle when possible, and finally a variable to run each of the screen states (runMainMenu, runGame, runCPUGame, runTutorial) that would change between True and False depending on what button was clicked to show the correct corresponding information. Finally, the last part of the Variables.py file was the Button class that was sued to create the button on the main menu. When these buttons were clicked it would take you to a different screen where you could play a CPU game, a local multiplayer game, or to view the tutorial respective of which button the user clicked. With all of the initial set up now out of the way, I was finally able to create functions that would be used in playing the game.

Within the Chess.py file, all of the functions used to play the game for both multiplayer and against the CPU were created, as well as functions that draw the events happening within both the game and the tutorial screen. I started with importing pygame, random and the Variables.py file. The first function created was the **checkMoveOptions(pieces, locations, turns)** function that accepted the piece, location and turn of the player. It contained empty [] of a moveList, a totalMoveList and a castleMoves to all be sued throughout the game. Next, within this function I iterate through the movelist of each piece. Depending on the piece, the game decides what move function to call for the piece selected. For example, if a pawn was selected it would set the moveList = checkPawnMoves() function and the perform the pssoble moves for that pawn. This occurred for all piece types and appended all of these moveLists to a totalMoveList that was returned from the function. The next function was the **checkPossibleMoves()** function. This contained the inCheck, intersectingMoves, and checkmate variables. For white’s turn it would set the moveOptionsList to whiteMoveOptions, the king\_position to the location of the white king, the enemy\_moves to blackMoveOptions and the enemy\_position to the blackPieceLocations. This was repeated in terms of black’s turn. This function determines what moves are valid per piece at any time of the game. It would take the initial empty [] of intersectingMoves and determine if the move could be made. This allowed for any move to be made with a couple exceptions. When a king was in check, it would get the kings position and the piece attacking the kings position. It would then limit the movement of all pieces to only be pieces that would take the king out of check; whether this was in the form of moving the king himself using the **kingInCheckAfterMove(king\_position, move, enemy\_moves)** function, putting a piece between the attacker and the king using the **isBetween(pos1, pos2, pos3)** function or just capturing the attacker using normal play methods. If any of these moves were available, then the game would continue. However, an issue I came upon with the design of my game (which I stated above) was with the possibleMoves[]. Since the game was set to determine the moves of a piece depending on the piece clicked at the time, the only way a checkmate would work was when I clicked on a piece with an empty moveset. To remedy this I would need to change the entire mechanic of how the moves of each piece were determined but with the little time I had remaining to work on the project, this task was unobtainable. The game ends when I click on a piece with an empty moveset (which mimics checkmate), however I would have preferred to allow the game to end automatically when this occurred.

The next set of functions created we can classify as “Move Checking Functions.” These functions are the logic behind how each piece obtains the moves they can make. With each piece having unique movement, functions per piece were necessary to define the logic to move said piece. We start with the **checkPawnMoves(position, color)** function. It starts with an empty moveList [] and uses the color and position of the pawn to determine how far it can move. With a pawn, they can move 1 square forward except for when they are in their starting position, which at this point they can move two squares forward. For pawns to capture an enemy piece, they must look at their front diagonals. Using our position[] we take the x and y coordinate of the pawn at any given time and subtract one to both it’s x and y to travel up the board. These are repeated for black pawns but using the math to travel down the board. The **checkEnPassant(oldCoordinates, newCoordinates)** function allows for pawns to make the special En Passant move. This is when a pawn moves two spaces from its starting position and lands adjacent to a preexisting enemy pawn. When this occurs, the enemy pawn can diagonal capture the pawn that moves two spaces in its normal diagonal manner. We use the same math for capturing and pair it with the logic for en passant to allow for these moves to be made. The final pawn special move is a pawn promotion. When a pawn reaches the other end of the board, they can promote to any other type of piece except for the king. When they promote to a different piece, they acquire the moveset of that piece. Using the **checkPawnPromotion()** function, we use a pawnIndex [], a promotionIndex set to -1 and whitePromotion and lackPromotion Booleans. These check if any pawn of either color reaches their respective opposite end of the board. When this does occur we return whitePromotion, blackPromotion, and promotionIndex and use these in the **checkPawnPromotionSelection()** function. This new function takes the mousePosition of the user and determines where it clicks on the soon to be mentioned drawPawnPromotion() function, which draws the menu to select which piece the pawn wants to promote to. The next function is the **checkRookMoves(position, color)** function. In similar fashion to the checkPawnMoves(position, color) function, it starts with an empty moveList [] and uses the color and position of the rook to determine how far it can move. With a rook, they can move the entire board in all directions perpendicular to the piece as long as they don’t run into a friendly piece. The function uses a pathway Boolean and a chain variable set to 1. It will then check up, down, left and right respective of the rook and will append the straight movement to the moveList [] as long as it doesn’t intersect a piece in the friendList; which is the location of its friendly pieces. The movement can traverse the whole board in a straight line, so we also need to include an enemyList; which is the location of its opponent’s pieces. Once the rook intersects an enemy piece, it ends its path and gives it a path leading all the way to that enemy piece, including the ability to capture it. The next function is the **checkKnightMoves(position, color)** function. Similarly, it starts with an empty moveList [] and uses the color and position of the knight to determine how far it can move. With a knight however, it’s moveList is limited to an “L” shape. This means that a knight can move 2 squares in one direction, and 1 square in another direction in the shape of an “L”. To achieve this, we use a targetSquares [] that contains all the coordinates of moving 2 squares in one direction, and 1 square in another direction. (targetSquares = [(1, 2), (1, -2), (2, 1), (2, -1), (-1, 2), (-1, -2), (-2, 1), (-2, -1)]). We then again limit this to not allow the knight to move onto a square with a friendly piece, and to capture an enemy piece if it lands on a square containing an enemy. The next function is the **checkBishopMoves(position, color)**. This uses the exact same logic as the checkRookMoves(position, color) function, except a bishop moves in a diagonal across the entire board instead of perpendicular. Using the same pathway and chain variables as we used in the checkRookMoves(position, color) function, we use the exact same logic except we change the math for the piece to move in an and y coordinate diagonal of the bishop. Again, the piece stops when it reaches a friendly piece, and can capture an enemy piece that it may intersect. The next function for movement was the easiest of all, the **checkQueenMoves(position, color)** function. Because a queen moves in all directions across the entire board, we just give it both the checkRookMoves(position, color) and the checkBishopMoves(position, color) functions, which already contain the logic for perpendicular and diagonal movement respectively. The next function is the **checkKingMoves(position, color)**. We used similar logic as the checkKnightMoves(position, color) function because a king can move one square in any direction. We take a targetSquare[] that contains the x and y coordinates for 1 square on all sides of a king, (targetSquares = [(0, -1), (1, -1), (1, 0), (1, 1), (0, 1), (-1, 1), (-1, 0), (-1, -1)]) and similarly to the checkKnightMoves(position, color) function, we limit this to not allow the king to move onto a square with a friendly piece, and to capture an enemy piece if it lands on a square containing an enemy. Finally, the last “Move Check Function” is the **checkCastling()** function. Castling is a special move in where that if a rook and a king have not moved from their initial squares and all squares between the two are empty, the king and rook may come together and swap sides with each other. This function is more complicated than the others due to the fact that we need to use a castlingMoves [] that stores the coordinates as [((kingCoordinates), (rookCoordinates))] to determine if either piece has yet to move. We use the rookIndices [], rookPosition [], the kingIndex set equal to 0 and the kingPosition set to the coordinates (0,0) to set the variables. These variables then are used to determine what rook has yet to move and if the rook and king are in their starting positions. If these are true, then we can allow castling. If not, these moves are not drawn onto the board. With the “Move Check Functions” out of the way, we can now look at the “Draw Functions.”

The “Draw Functions” are all the functions used to draw all of the visuals seen within the game. The first function is the **drawMainMenu()** function. It uses the buttons from the button class in the Variables.py file to create 3 buttons: A “Play CPU Button”, a “Play Multiplayer Button” and a “Tutorial Button.” These buttons are then drawn on the screen under the title and have checks in place to see when they are clicked. With a command variable set to 0, depending on the button clicked the command variables changes to 1, 2 or 3. If the “Play CPU Button” is clicked command = 1, if the “Play Multiplayer Button” is clicked command = 2 and if the “Tutorial Button” is clicked command = 3. This command variable is then used to determine which screen to switch to next, the CPU game, the multiplayer game, or the tutorial respectively. The next function is the **drawBoard()** function. This function draws the chess board for both the CPU and multiplayer game. It uses the preexisting background of the screen and draws 32 evenly spaces white squares in a checkboard fashion to give the illusion of a checkboard. The function draws the box at the bottom that contains the game dialogue indicating what needs to occur based on the turn number; where if turn = 0 it outputs “White, Please Select a Piece”, if turn = 1 it outputs “White, Please Select a Location”, if turn =2 it outputs “Black, Please Select a Piece”, and if turn = 3 it outputs “Black, Please Select a Location.” This function also draws the column of letters and row of numbers indicating what square of the chessboard is which. The function also draws two areas on the far right; a forfeit button in the bottom right to allow for a player to click the button to forfeit the game and an area to display captured pieces. Finally, the function will output the text in the bottom rectangle to “Select Piece to Promote Pawn” when a pawn is able to be promoted. The next function is the **drawPieces()** function. This places the pngs of the pieces in their correct starting squares according to the coordinates already stated. It will then outline the piece you select in blue if the piece is white and red if the piece is black. The **drawPawnPromotion()** function previously stated will draw the menu for pawn promotion. When a pawn reaches the end of the board, it will display a menu of the 4 selectable pieces (Rook, Knight, Bishop, and Queen) on top of the captured pieces rectangle. It will then change the png of the pawn to whichever piece is selected, giving it that pieces new moveset. The next function is the **drawCastling(moves)** function. It will draw the possible castle move according to the checkCastling() function in the color respective of the player (blue for white and red for black). The **drawMoves(moves)** function will highlight the squares of possible moves in the color that accompanies the piece (blue for white and red for black). The **drawCapturedPieces()** function will draw a png of the piece that was landed on by an enemy player in the rectangle above the forfeit button. It separates the black and white pieces into columns and displays them according to the piece captured. The **drawCheck()** function uses the inCheck variable and the kingInCheck() function to see when the king is in check. When the king is in check, it draws a yellow square around the king and plays a sound effect alerting the player that they are in check. The **kingInCheck()** function is used in the checkPossibleMoves() function to determine if the moves are valid when the king is in check. The kingInCheck() function specifically checks when the king is in check and send that information to the checkPossibleMoves() function. The final ‘Draw Function” is the **drawGameOver()** function. This function takes the winner variable that is used in the game loop and draws a game over screen according to what the winner variable is set to. It displays the color of the winner and asks whether the player wants to play again by prompting them to hit the ENTER key to reset the board or the ESC key to close the program. With these functions out of the way, we can set the move options per piece using **blackMoveOptions = checkMoveOptions(blackPieces, blackPiecesLocation, 'Black')** and **whiteMoveOptions = checkMoveOptions(whitePieces, whitePiecesLocation, 'White')**. These two variables are what set are game loop and allow us to separate the moves per color by updating the color with what piece moved, what its new location is, and white color it belongs to. With these in place, we move on to the final two functions before the main game loop.

These final two functions work hand in hand to allow for the CPU to make its own moves without human input. The first function being the **makeSmartMove()** function. This function uses the main game loop that will be stated later and adapts it to allow for the AI to move. We only use this function when turn is equal to 2, meaning that the AI will be black. It takes the best\_move\_score variable and best\_moves [] and will make a move. The move is based off of a piece\_index to select a piece and the possible\_moves variable to see which moves that piece has. For any move that is a part of possible\_moves, it will set a score equal to the other CPU function, the **evaluateMove(piece, move)** function. This function initializes a piece\_value and a move\_score to 0. It then assigns each piece a piece\_value dependent on a created value of the piece. The “better” the piece movewise, the higher the piece\_value. The list of piece\_values is as follows: if piece == 'Pawn': piece\_value = 1. if piece == 'Knight': piece\_value = 3. if piece == Bishop: piece\_value = 3. if piece == Rook: piece\_value = 5. if piece == Queen: piece\_value = 9. if piece == King: piece\_value = 100. The higher the piece\_value the higher the defense priority is (it will defend the king before using him) and the more likely it is to capture the player’s piece of a higher piece\_value (goal is to take the player’s king). It then uses the same format to create a captured\_value set to the same values as the pieces and returns a move\_score that increments with the captured\_value – the piece\_value to determine the best possible move to be made. This returned move\_score is then used back in the makeSmartMove() function. With this calculated move\_score, it sets score equal to the move\_score. If score is greater than the best\_move\_score then best\_move\_score becomes score and best\_move is set to the piece and move determined by the evaluateMove() function. When the best\_moves variable is created, it will take the piece\_index and the new\_location variables to pick a random piece and random move of that piece using Pythons built in random library. The algorithm will most likely pick any random move of a piece unless it can capture one of the player’s pieces, which it will do so if possible. The best\_move that is randomly generated is then made, setting the current\_location\_index to the new location it chose, and switches the turn variable to 0 making it white’s turn. It then uses the same logic as the main game loop that will be explained later to remove any piece that is captured from the board, place the png in the captured column and update the blackMoveOptions to allow for the moves and pieces to be stored correctly. With all of the functions now explained, we can finally look through the different while loops that are used to create a playable game.

We first look at the while **runMainMenu** loop. This loop fills the screen with and if the mainMenu Boolean is true (which is on start up), will set command to the drawMainMenu() function. Using the same numbers for the commands as the drawMainMenu() function, as each button is clicked it will change the command variable and will set the runMainMenu Boolean to False, turning off that while loop and setting either the runCPUGame, runGame, or runTutorial Boolean to True, starting that while loop and turning on that screen. We will start with explaining the tutorial page.

When the Tutorial button is clicked, we start the **runTutorial** screen. In this while loop, we set the tutorialPage variable to 1. This indicates that when this while lop starts, we are met with page one of the tutorial. Page one of the tutorial explains all about how each piece can move. There are arrows within the tutorial pages that when clicked will either send you forward or backwards a page dependent on the direction of the arrow. On page one, if we click the right arrow we are met with page two. This page shows pngs of the 3 special moves. When you click on one of the pngs of the special move, it will enlarge the png and show the text explaining said move. The three special moves explained are: Pawn Promotion on page 3, En Passant on page 4 and Castling on page 5. Each of these special moves pages have a left arrow that take you back to page one of the tutorial so you can view the tutorial has a whole in whatever order you please. With these 5 pages of the tutorial explained, we can now move into the runGame while loop.

When the Play Multiplayer button is clicked, we start the **runGame** screen. In this while loop, we can play our multiplayer game. We fill the screen with our background, call the drawBoard(), drawPieces(), drawCapturedPieces() and drawCheck() functions. These are used to initialize the game screen. We can also call the checkPawnPromotion() and checkPawnPromotionSelection() functions to allow the game to recognize pawn promotion. The game is played by selected a piece. When a piece is selected selection (which is initialized to -1) is changed. When selection no longer equals -1, possibleMoves is set to the checkPossibleMoves() that determines what moves are valid. It will then perform the drawMoves(possibleMoves) function based on what he possibleMove variable is equal to, displaying what moves are valid. If the piece selected is a ing and one of the possibleMoves is a castling move dependent on the drawCastling(), it will draw that move to the screen to be a valid move. The game works on a 2 step turn per player. When turn is 0 or 1 it is white’s turn, when turn is 2 or 3 it is blacks turn. I will be explaining the game logic in terms of white’s turn as the logic for black’s turn is the exact same except anywhere that relates the turn to white is swapped for black. To start, white always begins a game. We start with clicking a piece. If we click a piece, the piece will be outlined in a colored square. After we click that piece, turn is equal to 1 telling you to select a location for that piece. If we click one of the highlighted squares that is a valid possibleMove the whitePiecsLocation will now become the click location of the selected piece. If the king is not in check while performing this action, the click is confirmed, the piece moves to that new location, a sound effect is played to audible alert that a move was made, both blackMoveOptions and whiteMoveOptions are updated with where their respective pieces are in this game state of the board and turn becomes 2, switching it to black’s turn. If the king is in check though, we check if the click is part of the moveset that would block check, by either placing yourself between the attacker and the king, capturing the attacker, or moving the king out of harms way. If one of these are true, again the click is confirmed, the piece moves to that new location, a sound effect is played to audible alert that a move was made, both blackMoveOptions and whiteMoveOptions are updated with where their respective pieces are in this game state of the board and turn becomes 2, switching it to black’s turn. However, if one of these are not the clicked-on space it will undo the move and reset the turn back to white’s initial turn, redoing all of the logic until a valid move is clicked. If any of these clicks are a valid move and an opponent occupies the square the white piece will capture the black piece, placing in the capture piece column and removing it from the game board. The click will also look if the clicked spot was an en passant move, which would perform the en passant function or if the move is a castling move, which will perform the castling move function. No matter the move the game will always confirm click if valid, the piece moves to that new location, a sound effect is played to audible alert that a move was made, both blackMoveOptions and whiteMoveOptions are updated with where their respective pieces are in this game state of the board and turn becomes 2, switching it to black’s turn. This logic is then repeated but for when turn is 2 and 3 for black. After any click from the black player, all of theabove will occur again except for when it sets turn to 2, it will now set turn to 0; which sets the game to white’s turn and starts the cycle over until checkmate occurs. Checkmate can only occur when a piece that has no valid moves in its moveList is clicked. This is not my referred method of calling checkmate but because of the way I coded the game, moves are only generated through clicks. For checkmate to automatically end the game I believe I would need to store all possible moves per piece in a dictionary and call the moves from there. Which if the dictionary was ever empty, it would call checkmate and end the game. This task is not feasible in my current version of the game because I would have to rework the entire move system of y game which I do not have ample time to do so. Nonetheless, when the game does end it sets gameOver to true, and winner is set to whichever color won, showing the winner their winning screen and playing a victory sound effect. It also shows the players that they can play again by pressing ENTER or close the game by pressing ESC. If they press ESC, the program ends. If they press ENTER, the game must be reinitialized to the beginning. The gameOver variable is set back to false, winner is set to an empty string, whitePieces and blackPieces are set back to an array of the pieces in order of their location on the board based on coordinates and their names. The accompanying captured [] are set back to empty, and the moved [] per color are set to false for each piece. Finally, turn is set back to 0 to allow white to start, selection is set to -1 to show nothing has been selected, possibleMoves is set back to an empty array and blackMoveOptions and whiteMoveOptions are both reinitialized with the checkMoveOptions() function resetting the selections made pre color throughout the last game. When all of this is completed, the game restarts and can be played again with no stored memory of the previous game. With the runGame while loop explained, we can now talk about the final game loop; the runCPUGame loop.

When the Play CPU button is clicked, we start the **runCPUGame** screen. In this while loop, we can play our game against the CPU. The exact same logic is used here as was used in the runGame while loop. However, the only change made is that when it becomes black’s turn, the game no longer relies on the use of a click variable form the player. Instead, when turn is equal to 2, it calls the makeSmartMove() function. This already explained function handles the logic for the AI to choose how it should move. The only logic added here is that the blackMoveOptions and whiteMoveOptions are set to include the new locations and moves of their respective pieces and sets turn back to 0 to allow the user (white) to make their move again.

With this explained, I have now gone into full detail of my chess project. While I created a working game of chess, in the future I would like to further perfect this project on my own time. In theory, I would rework the way moves are calculated, store them in a dictionary and allow for checkmate to be called automatically without the need of user input. I would also like to play around with the AI I created, possibly making it more difficult with other algorithms I came across in my research. Without a time constraint as a factor, I know I can make a better version of a good game I am presenting to you. All in all, I am pleased with the outcome of my project and look forward to what I can achieve with it on my own with all the knowledge of programming. I have acquired with my time at Youngstown State University.