I am making a chess game in python The game is played on an 8x8 board of alternating light and dark squares. There are six piece types, and each piece can move in a very specific way. The pieces may move as follows. Queens may move in one of the eight directions around her - North, North-East, East, South-East,South, South-West, West, or North-West. She may move as far as she can, but cannot move past a piece in her way. If a piece in her way is of the opposing color, she may capture that piece and take the spot. Captured pieces are removed from the board. Kings may move identically to the Queens, but only a maximum of one space at a time. Bishops may move diagonally, but with the same limits of the Queen; i.e. they can capture a piece of an opposing color but must stop there, and cannot move past a piece of their own color. Knights may move in an L-shape; either vertically one space and horizontally two, or vertically two spaces and horizontally one. They may not land on a piece of their same color, but can jump over them. They capture a piece of the opposing color if they land on that piece at the end of their movement. Rooks may move vertically or horizontally, as many spaces as they can, but are governed by the same limits as the Queen. Pawns may only advance toward the opposing teams direction. On their first move, they can move two spaces (but aren't required to). On subsequent moves they can only advance a single space. Pawns may only move vertically - unless there is a piece of the opposing color diagonally to them in their forward movement direction. In this case, they may capture the piece and occupy its square. If a Pawn reaches the opposite side of the board (i.e. the first row for White or the eighth row for Black), they are turned into a Queen and may move accordingly on their next turn. Notice that in other versions of chess the Pawn may become different types of pieces - or we may return captured pieces; we will limit ours to becoming a Queen. The GUI is given already (import pygame as pg

import pygame\_gui as gui

from game import \*

class GUI:

def \_\_init\_\_(self) -> None:

pg.init()

self.\_game = Game()

self.\_screen = pg.display.set\_mode((1440, 900))

pg.display.set\_caption("Laker Chess")

self.\_pieces = pg.image.load("./images/pieces.png")

self.\_ui\_manager = gui.UIManager((1440, 900))

self.\_side\_box = gui.elements.UITextBox('<b>Laker Chess</b><br /><br />White moves first.<br />', relative\_rect=pg.Rect((1000, 100), (400, 500)),

manager=self.\_ui\_manager)

self.\_undo\_button = gui.elements.UIButton(relative\_rect = pg.Rect((1000, 50), (100, 50)), text='Undo',

manager=self.\_ui\_manager)

self.\_restart\_button = gui.elements.UIButton(relative\_rect = pg.Rect((1200, 50), (100, 50)), text='Reset',

manager=self.\_ui\_manager)

self.\_piece\_selected = False

self.\_first\_selected = (0, 0)

self.\_second\_selected = (0, 0)

self.\_valid\_moves = []

def run\_game(self) -> None:

running = True

time\_delta = 0

clock = pg.time.Clock()

while running:

for event in pg.event.get():

if event.type == pg.QUIT:

running = False

if event.type == pg.MOUSEBUTTONDOWN:

x, y = pg.mouse.get\_pos()

y, x = self.\_\_get\_coords\_\_(y, x)

piece = self.\_game.get(y, x)

if not self.\_piece\_selected and piece:

if piece.color != self.\_game.current\_player:

continue

self.\_piece\_selected = True

self.\_first\_selected = y, x

self.\_valid\_moves = piece.valid\_moves(y, x)

self.\_piece\_selected = piece

elif self.\_piece\_selected and (y, x) in self.\_valid\_moves:

target = self.\_game.get(y, x)

moved = self.\_game.move(self.\_piece\_selected, self.\_first\_selected[0], self.\_first\_selected[1], y, x)

if moved:

self.\_side\_box.append\_html\_text(self.\_piece\_selected.color.name + ' moved '

+ str(type(self.\_piece\_selected).\_\_name\_\_))

if target:

self.\_side\_box.append\_html\_text(' and captures ' + str(type(target).\_\_name\_\_))

self.\_side\_box.append\_html\_text('<br />')

computer\_message = self.\_game.\_computer\_move()

if computer\_message:

self.\_side\_box.append\_html\_text(computer\_message)

else:

self.\_side\_box.append\_html\_text('Invalid move. Would leave '

+ str(self.\_piece\_selected.color.name) + ' in check.<br />')

if self.\_game.check(Color.WHITE):

self.\_side\_box.append\_html\_text("WHITE is in CHECK!<br />")

if self.\_game.check(Color.BLACK):

self.\_side\_box.append\_html\_text("BLACK is in CHECK!<br />")

if self.\_game.mate(Color.WHITE):

self.\_side\_box.append\_html\_text("WHITE is in CHECKMATE!<br />GAME OVER!")

if self.\_game.mate(Color.BLACK):

self.\_side\_box.append\_html\_text("BLACK is in CHECKMATE!<br />GAME OVER!")

self.\_piece\_selected = False

else:

self.\_piece\_selected = False

if event.type == gui.UI\_BUTTON\_PRESSED:

if event.ui\_element == self.\_restart\_button:

self.\_game.reset()

self.\_side\_box.set\_text("Restarting game...<br />")

if event.ui\_element == self.\_undo\_button:

if self.\_game.undo():

self.\_side\_box.append\_html\_text('Undoing move.<br />')

else:

self.\_side\_box.append\_html\_text('Nothing to undo.<br />')

self.\_ui\_manager.process\_events(event)

self.\_screen.fill((255, 255, 255))

self.\_\_draw\_board\_\_()

self.\_ui\_manager.draw\_ui(self.\_screen)

self.\_ui\_manager.update(time\_delta)

pg.display.flip()

time\_delta = clock.tick(30) / 1000.0

def \_\_get\_coords\_\_(self, y, x):

grid\_x = x // 105

grid\_y = y // 105

return grid\_y, grid\_x

def \_\_draw\_board\_\_(self) -> None:

count = 0

color = (255, 255, 255)

for y in range(0, 8):

for x in range(0, 8):

if count % 2 == 0:

color = (255, 255, 255)

else:

color = (127, 127, 127)

count = count + 1

pg.draw.rect(self.\_screen, color, pg.rect.Rect(x \* 105, y \* 105, 105, 105))

if self.\_piece\_selected and (y, x) == self.\_first\_selected:

pg.draw.rect(self.\_screen, (255, 0, 0), pg.rect.Rect(x \* 105, y \* 105, 105, 105), 2)

if self.\_valid\_moves and self.\_piece\_selected and (y, x) in self.\_valid\_moves:

pg.draw.rect(self.\_screen, (0, 0, 255), pg.rect.Rect(x \* 105, y \* 105, 105, 105), 2)

if self.\_game.get(y, x):

self.\_screen.blit(self.\_game.get(y, x).\_image, (x \* 105, y \* 105))

count = count + 1

pg.draw.line(self.\_screen, (0, 0, 0), (0, 840), (840, 840))

pg.draw.line(self.\_screen, (0, 0, 0), (840, 840), (840, 0))

def main():

g = GUI()

g.run\_game()

if \_\_name\_\_ == '\_\_main\_\_':

main()). This is what I need create a new project. Add a file called piece\_model.py. In the file create an enumeration called Color . Give it values for White and Black. These will make our code easier to read and debug. Create an abstract base class called Piece . I will be creating a folder called images and placing the chess images bB.png, bK.png, bN.png, bP.png, bQ.png, bR.png, wB.png, wK.png, wN.png, wP.png, wQ.png, wR.png. Make a static variable (not an instance variable) that holds the path to this image. Create another static variable called \_game so pieces can keep track of the current game. This will allow them to query the state of the game board while moving. Add a static method to the class:

def set\_game(game):

if not isinstance(game, Game):

raise ValueError("You must provide a valid Game instance.")

Piece.\_game = game

This method is static because it does not accept the self parameter. Recall that static variables belong to the class - not instances of the class. All pieces will thus share the same game instance. Create a constructor that takes as a parameter a Color . Set an instance variable for \_color to this value. Add a self.\_image . Whereas the static variable will be the file that holds all of the piece images, this instance variable will hold the individual image for the current piece. Set the image equal to

pygame.Surface((105, 105), pg.SRCALPHA)

Make a getter property for color . Do not create a setter property for it, as it should be read-only. Write a set\_image(self, x: int, y: int) -> None method. Add the following code: self.\_image.blit(Piece.SPRITESHEET, (0, 0), pygame.rect.Rect(x, y,

105, 105))

This code will take an x and y value and copy from our image file a 105x105 pixel chunk into the current piece's image.

Add a method with the signature

def \_diagonal\_moves(self, y: int, x: int, y\_d: int, x\_d: int, distance:

int) -> List[Tuple[int, int]]

This method signature looks like a lot, but it isn't that bad. Its job is to start at a particular

spot on the board, and find all valid moves in a given diagonal direction. This method will

take a board position ( y and x ), and a direction vector ( y\_d and x\_d ). For the

direction vector, vertically will be represented by -1 or 1 (up or down, respectively), and

horizontally as -1 or 1 (left or right, resepctively). So, if the function is called with the

values y\_d = 1 and x\_d = -1 , we will be checking down to the left. The method will

only check up to the value of distance ; so if we pass in 1 (for instance), we only want to

check up to 1 space away from the given y and x position. The method will add all valid

moves in the diagonal direction up to the given distance, to a list that will be returned from

the function.

As an example, if we pass in location (4, 3) to this method, with the direction -1, -1

(checking up and left), and there are no other pieces on the board between the given

location and the edge of the board, we should add (3, 2) and (2, 1) to the moves

list. If at any point we run into another piece, we must stop checking in that direction.

Therefore, if we are White and there is a white piece at (2, 1) , we would not be able to

add that location to our valid moves list. However, if the piece at (2, 1) is the opposite

of the current piece's color, we can capture it. In that case, we will add that location to the

valid moves list.

Note that this method only checks one direction at a time. We will write a helper method

that will call this for all four valid diagonal directions.

Add the method

def \_horizontal\_moves(self, y: int, x: int, y\_d: int, x\_d: int, distance:

int) -> List[Tuple[int, int]]

This method will work exactly as the diagonal version, but will only check for possible

moves horizontally in one direction.

Add the method

def \_vertical\_moves(self, y: int, x: int, y\_d: int, x\_d: int, distance:

int) -> List[Tuple[int, int]]

Again, it works exactly as the diagonal version, except this one checks a vertical direction.

Add the methods

def get\_diagonal\_moves(self, y: int, x: int, distance: int, moves) ->

List[Tuple[int, int]]

def get\_horizontal\_moves(self, y: int, x: int, distance: int) ->

List[Tuple[int, int]]

def get\_vertical\_moves(self, y: int, x: int, distance: int) ->

List[Tuple[int, int]]

These are convenience methods that simply call each of the previous methods for all

possible directions. The first should collect and return the moves from four calls to the

\_diagonal\_moves method, while the next two will call the appropriate functions to

collect both vertical (up and down) and horizontal (left and right) directions.

Add two abstract methods:

def valid\_moves(self, y: int, x: int) -> List[Tuple[int, int]] -

This function will be customized for each piece type. It's job will be to determine

all valid moves for a piece at the given location.

copy(self) - This function will copy a piece. We will use this to simulate

possible moves for the computer, by copying the state of the board (and thus the

pieces). Most pieces will use similar copying code, but not all! You may find that a

particular piece needs a piece of information that other pieces do not. Thus, they

would need a different copy method. In the next part I will be coding the pieces but can you code what I gave you so far? Please.

Now, we will begin creating the individual piece classes. These will all be subtypes of the Piece class we

created above - and thus able to share a lot of functionality. Each type must have its own valid\_moves

and copy methods, though. Be sure to call the super() constructor in each class. Then, there are still

small differences between them. Do the following add a King class. In the constructor call the set\_image method with appropriate values to copy the Black or White (depending on the value passed into the constructor) image to this piece's image variable. In this class's valid\_moves method, use the functions we created above to collect all the valid moves for the King one space in the horizontal, vertical, and diagonal directions and return them. In the copy method return a new King of the same color. Add a Queen class. The Queen code will be exactly the same as the King's, except the distance she may move will be 8 instead of 1. Add a Bishop class. The Bishop will only move diagonally, but is otherwise the same as the Queen. Add a Knight class. The Knight has eight possibile moves, and none of them are in a direction we have an existing method for. In the Knight's valid\_moves method, create a list of all eight possible moves. Remove any that contain a location of a piece of the same color as the current Knight. Add a Rook class. The Rook may only move horizontally or vertically, but up to 8 spaces. Add a Pawn class. The Pawn must also keep track of whether it has moved previously, since a Pawn's first move may be one or two spaces. In valid\_moves , add each location that the Pawn may be able to move to - including a diagonal space if a piece of the opposing color is in a space diagonally (at a distance of one space away) away from the Pawn. In the copy method don't forget to copy the new state you added for tracking whether the Pawn has moved previously.

In our game, the Computer will be the Black player and the Human will be White. You may randomly choose which player goes first, or set it to a default value. Complete these tasks Create a Game class. In the constructor, setup the board and determine the current player. Game will keep track of the current player Color , an 8x8 list of Pieces for the board (you can initialize each position to None ), and a stack to hold prior board states. We will copy the board after each move to allow us to undo moves, and to simulate moves for the AI, so that it may choose a "good" move. Add a reset method that resets the board, player, and prior states list to default states. Add a \_setup\_pieces method that will add pieces in their correct starting positions for both players. Write a get(self, y: int, x: int) -> Optional[Piece] method. This method will return the piece at the given position, or None if no Piece exists at that spot. Create a switch\_player method that switches the current player to the opposing player. Make an undo method. This method will pop the last board state off the stack, and set the current board to it. Return True if this can be done. If there is no prior state, return False . Implement a copy\_board method. This method is essential to the functioning of our game. In addition to allowing the human to undo a move, having a copy of the board allows the AI to explore possible moves without affecting the current board. Copying a board must be a deep copy - not a shallow copy. The copyfunctions you wrote for the Piece classes will perform a deep copy if written correctly. Add a move(self, piece: Piece, y: int, x: int, y2: int, x2: int) -> bool method. This method should first copy the board into the prior states stack. The method should then perform the move by setting the new location to the piece, and removing the piece from the old location. If the move was made by a Pawn , then the pawn should be updated to reflect that it is no longer the pawn's first move. You must then determine if the move resulted in the current player being placed in check, and undo the move if it does. No player should be allowed to perform a move that places themselves in check. For instance, if White moves a bishop that is blocking Black's queen from putting White's King in check, the move should be undone. If this happens, return False . If the piece is a Pawn and the new location is the opposite side of the board, the pawn should be removed and a Queen of the same color placed in its location. Your function will then switch the current player and return True to indicate a successful move occurred. Write a get\_piece\_locations(self, color: Color) -> List[Tuple[int, int]] method. This will return the (y, x) locations (as a tuple) for all the pieces on the board of the given color.

Implement a find\_king(self, color: Color) -> Tuple[int, int] method. This will

find the position of the King of the given color.

Create a check(self, color: Color) -> bool method. This method will get locations

of the opposing pieces (use the method above). It will then iterate over each of those

pieces, calling their valid\_moves method, and adding all valid moves for all pieces to a

list of possible moves. The method should then get the position of the king. If the position

of the king is in the list of possible moves, the king is in check. Return True in this case,

and False otherwise.

Checkmate results when the king is in check and no moves left on the board will allow the

king to escape check. To get out of check, the king can move, or another piece may move

to block the piece threatening the king. For this method, create a mate(self, color:

Color) -> bool . First, call the check method. If the king is not in check, the king

cannot be in checkmate. If the king is in check, you will then need to see if the king can

escape. You do this by getting a list of possible moves for the king, and a list of opponent

pieces. If there is a move the king can perform that is not in the list of the opponent's

possible moves, the king is not in checkmate. However, just because the king can't move to

escape check doesn't immediately mean the king is in checkmate. If another piece can

move such that it blocks the king being in check, then the king is safe. To determine this,

get a list of all possible moves for the current player. Loop through the list trying each

move, and then calling the check method to see if that move resulted in check. If a move

is found that results in the king not being in check, then the king is not in checkmate.

Remember to undo each move after you test it.

Implement a \_computer\_move method that selects a random (but valid) move for the

computer player. In the next phase, you will create a more robust method for picking

computer moves.