Create Performance Task: Written Responses

Assessment Overview and Performance Task Directions for Students

Video Submit one video in .mp4, .wmv, .avi, or .mov format that demonstrates the running of at least one significant feature of your program. Your video must not exceed 1 minute in length and must not exceed 30MB in size

Prompt 2a.

Provide a written response or audio narration in your video that:
- identifies the programming language;
- identifies the purpose of your program; and
- explains what the video illustrates.

(Must not exceed 150 words)

The programming language I used is python and I used Pycharm as my IDE. The purpose of my program is to allow two users to play chess on the same computer. My video demonstrates the chess game being played. My version of chess has a slight twist; the game is won by taking the king, not putting it in checkmate. In order to move a piece you click on the piece and then click on where you want to move it. If that move is valid it will move the piece, otherwise it will not. There is also an option in the bottom of the screen which asks whether the users want the board to be flipped after every move or not.

Prompt 2b.

Describe the incremental and iterative development process of your program, focusing on two distinct points in that process. Describe the difficulties and / or opportunities you encountered and how they were resolved or incorporated. In your description clearly indicate whether the development described was collaborative or independent. At least one of these points must refer to independent program development. *(Must not exceed 200 words)*

All of my development was independent. I first set up the gameboard which was an 8 by 8 grid of buttons which I then placed images of chess game pieces on in the right places. Next, I programmed rules for each piece so that no illegal moves could be made. I finally added functions/methods for special events such as castling, promotion, and flipping the board. The only difference to real chess is that the game is won when the king is taken. I came across many problems which I found through testing and then figured out how to fix. My biggest difficulty was assigning a command to each button that could pass the row and column of the button. Since the buttons were created in a for loop, the final row and column variable matched the last button. Through Google I found the lambda function which fixed all my problems. Another difficulty I encountered was converting a string to an image of the same name. After trying to research ways to change a string into a variable name I realized that I could just use a dictionary to map the string to the variable name.

Prompt 2c.

Capture and paste a program code segment that implements an algorithm (marked with an **oval** in **section 3**) and that is fundamental for your program to achieve its intended purpose. This code segment must be an algorithm you developed individually on your own, must include two or more algorithms, and must integrate mathematical and/or logical concepts. Describe how each algorithm within your selected algorithm functions independently, as well as in combination with others, to form a new algorithm that helps to achieve the intended purpose of the program. *(Must not exceed 200 words)*

Code Segment

```
def finishTurn(self):
  if self.tag2[0] == "blackKing":
    gameOver("White")
    self.switchTag()
  elif self.tag2[0] == "whiteKing":
    gameOver("Black")
    self.switchTag()
  elif self.tag1[0] == "whitePawn" and self.tag2[1][0] == 0:
    endRow(self.turn)
  elif self.tag1[0] == "blackPawn" and self.tag2[1][0] == 7:
    endRow(self.turn)
  else:
    if self.isFlip.get() == "1":
      self.flipBoard()
    else:
      if self.turn == "white":
        self.turn = "black"
        self.isTurn.set("Black's Move")
      elif self.turn == "black":
        self.turn = "white"
        self.isTurn.set("White's Move")
    self.switchTag()
    self.firstClick = False

def switchTag(self):
```

```python
    self.firstButton["image"] = ""
    self.secondButton["image"] = self.tagToIm[self.tag1[0]]
    for item in self.board:
        for items in item:
            for dict in items:
                if self.secondButton == dict:
                    items[self.secondButton][0] = self.tag1[0]
                elif self.firstButton == dict:
                    items[self.firstButton][0] = ""

def flipBoard(self):
    if self.turn == "white":
        self.row = 0
        for row in self.board:
            self.column = 0
            for buttons in row:
                for button, ids in buttons.items():
                    button.bind("<Button-1>", lambda event, x=self.row, y=self.column: self.clicked(event, x, y))
                    button.grid(row=7 - self.row, column=7 - self.column, sticky="NSEW", padx=0, pady=0)
                self.column += 1
            self.row += 1
        self.flip = False
    else:
        self.row = 0
        for row in self.board:
            self.column = 0
            for buttons in row:
                for button, ids in buttons.items():
                    button.bind("<Button-1>", lambda event, x=self.row, y=self.column: self.clicked(event, x, y))
                    button.grid(row=self.row, column=self.column, sticky="NSEW", padx=0, pady=0)
                self.column += 1
            self.row += 1
        self.flip = False
```

Written Response

An algorithm that I individually developed is the finishTurn() method. This method is imperative to the program because it finishes up a user's turn. The finishTurn() algorithm first uses logic (if/elif) to check whether the king has been taken, and if it has it will end the game and call the switchTag() method. It then checks whether a pawn has reached the last row. If none of the above conditions are met the finishTurn() algorithm would use logic to check whether the isFlip checkbox is checked. If it is, the flipBoard() method will be called. Finally, the algorithm switches the turn and calls the switchTag() method. The finishTurn() algorithm implements the flipBoard() and switchTag() algorithms. The switchTag() algorithm functions independently to “move” the piece by switching the images of the two squares. It loops through each button and uses logic to find the first and second button (where the piece moved to and was from) and then switched their tags. The flipBoard() algorithm functions independently to flip the board. It uses logic (if/else) to check whose turn it is and appropriately replace the pieces on the board to make it seem as if the board is flipped.

Prompt 2d.

Capture and paste a program code segment that contains an abstraction you developed individually on your own (marked with a **rectangle** in **section 3**). This abstraction must integrate mathematical and logical concepts. Explain how your abstraction helped manage the complexity of your program. *(Must not exceed 200 words)*

Code Segment

```
def diagMove(self, operator1, operator2, touch=False, isRook=-1):
    operators = {"<": operator.lt, ">": operator.gt, "==": operator.eq}
    if isRook == 1:
        self.increment1 = 0
        if operator2 == "<":
            self.increment2 = 1
        elif operator2 == ">":
            self.increment2 = -1
        secondOperator = operators[operator2](self.column2, self.column1)
    else:
        secondOperator = operators[operator2](self.column2 - self.column1, 0)
    if isRook == 0:
        self.increment2 = 0
        if operator1 == "<":
            self.increment1 = 1
        elif operator1 == ">":
            self.increment1 = -1
        firstOperator = operators[operator1](self.row2, self.row1)
    else:
        firstOperator = operators[operator1](self.row2 - self.row1, 0)
    if touch:
        if operator1 == "<" and isRook == -1:
            self.increment1 = 1
        elif operator1 == ">" and isRook == -1:
            self.increment1 = -1
        if operator2 == "<" and isRook == -1:
            self.increment2 = 1
        elif operator2 == ">" and isRook == -1:
            self.increment2 = -1
    else:
        self.increment1 = 0
        self.increment2 = 0
    if firstOperator and secondOperator:
        while not self.doneLoop:
            if self.tag2[1][0] + self.increment1 == self.row1 and self.tag2[1][1] + self.increment2 ==
self.column1:
                break
            if isRook == -1:
                if operator1 == "<":
                    self.row1 -= 1
                elif operator1 == ">":
                    self.row1 += 1
                if operator2 == "<":
                    self.column1 -= 1
                elif operator2 == ">":
                    self.column1 += 1
            elif isRook == 0:
                if operator1 == "<":
                    self.row1 -= 1
                elif operator1 == ">":
                    self.row1 += 1
            elif isRook == 1:
                if operator2 == "<":
                    self.column1 -= 1
                elif operator2 == ">":
                    self.column1 += 1
            for item in self.board:
                for items in item:
                    for dict, key in items.items():
                        if key[1][0] == self.row1 and key[1][1] == self.column1:
                            if key[0] != "":
                                self.isClear = False
                                self.doneLoop = True
                            break
        if self.isClear:
            self.finishTurn()
```

Written Response

An abstraction that I individually developed is diagMove(). This function is an abstraction because it simplifies the process of checking if a rook, queen, or bishop move is valid. Every time those pieces are moved, the program must validate the move to make sure that it is legal. This abstraction helps manage the complexity of the program because it moves the code away from the main body and I did not have to write the same code many times. Also, I called the abstraction 64 times so it drastically reduced the number of lines in my code, therefore making it easier to navigate around the program. The abstraction uses logic through many if/elif/else statements. All the logic statements are used in order to adjust the test for a valid rook, bishop, queen move depending on the parameters. For example, to test if a square is empty to validate a move, the column and/or row must change using many if/elif statements which test different conditions such as if isRook=-1 the program will use more if statements to find the operators that were passed as parameters to the method and adjust the row and column accordingly.