Tic-Tac-Toe is a game of X’s and O’s where two people play in a box of 9 spaces and someone tries to get three of their letters in a row to win.

To begin, we are going to define players. We are going to make use of classes so that we can define methods in them.

class Player:

    def \_\_init\_\_(self, letter):

        #letter is X or O

        self.letter = letter

    #we want all players to get their next move given a game.

    def get\_move(self, game):

        pass

When we initialize the class, we do so with letter so any class that extends this Player class inherits the letter argument. For the get\_move method, we are just going to pass. <code>pass</code> is kust saying skip and allows the code to run without errors since we are yet to do anything with it. We want to be able to have maybe two players against each other or a player against a computer. So we would define a random computer player class which would extend the player class.

class RandomComputerPlayer(Player):

    def \_\_init\_\_(self, letter):

        super().\_\_init\_\_(letter) #call this initialization in the super class.

    def get\_move(self, game):

        pass

We can also define a human player class but at this point, it is going to have the exact same look as the random computer player class above. Now that we have the player classes, we need to work on the game itself. In another file, we are going to define a Tic-Tac-Toe class. This class would have different methods that would define the game. First we initialize the class with a board argument.

class TicTacToe:

    def \_\_init\_\_(self):

        self.board = [" " for \_ in range(9)] #use a single list to replicate 3x3 board.

        self.current\_winner = None #keep track of winner

<code>self.board = [“ “ for \_ in range(9)]</code> means for each item in range(9), put an empty space so our board right now is a list of 9 empty spaces. Next we define a method to get available moves.

def available\_moves(self):

        #the line below is how it would look using list comprehension

        return [i for i, spot in enumerate(self.board) if spot == " "]

        #the code below does the exact same thing

        # moves = []

        # for (i, spot) in enumerate(self.board):

        #     #["x", "x", "o"] --> [(0, "x"), (1, "x"), (2, "o")]

        #     if spot == " ":

        #         moves.append(i)

        # return moves

Enumerate creates a tuple consisting of the index of the item and then the item itself. For example, [“x”, “x”, “o”] would return [(0, “x”), (1, “x”), (2, “o”)]. What the line of code above is doing is, for (i, spot) in enumerate(self.board), if the spot is empty, then we should append (add) that index (so we know the positions that are empty) to the moves list and return the list. This list would contain our available moves. Back to our player.py script, we can now work on our get\_move function. For the random computer player, we are going to change it to look like this.

class RandomComputerPlayer(Player):

    def \_\_init\_\_(self, letter):

        super().\_\_init\_\_(letter) #call this initialization in the super class.

    def get\_move(self, game):

        square = random.choice(game.available\_moves())

        return square

we would import random and then use <code>random.choice</code> to make a selection from the list and we would return the square that the computer chose. Keep in mind that the square is a number that correlates to the index position on the board. For the human player, we would need to do things a little bit differently.

class HumanPlayer(Player):

    def \_\_init\_\_(self, letter):

        super().\_\_init\_\_(letter)

    def get\_move(self, game):

        valid\_square = False

        val = None

        while not valid\_square:

            square = input(self.letter + "'s turn. Input move (0-8): ")

            #we are going to check that this is a correct value by trying to cast

            #it to an integer, and if it's not, then we say it's invalid

            #if that spot is not available on the board, we also say it is invalid

            try:

                val = int(square)

                if val not in game.available\_moves():

                    raise ValueError

                valid\_square = True #if these are successful

            except ValueError:

                print("Invalid square. Try again.")

        return val

What is going on here is the human player would be the one to input the square they want to move to on the board and then we need to check if the input is a valid input i.e. if the input is a number and not a letter and also if the input is actually in the available moves. We would make use of a try and except loop. If the try works, there is no need for the except block. It is for error checking. If the <code>val</code> can be converted to an integer, then we check if it is in the available moves list. If it is, we are good and we set <code>valid\_square</code> to true so the loop ends, else we raise an error. At the end we return val, val contains the square chosen by the user.