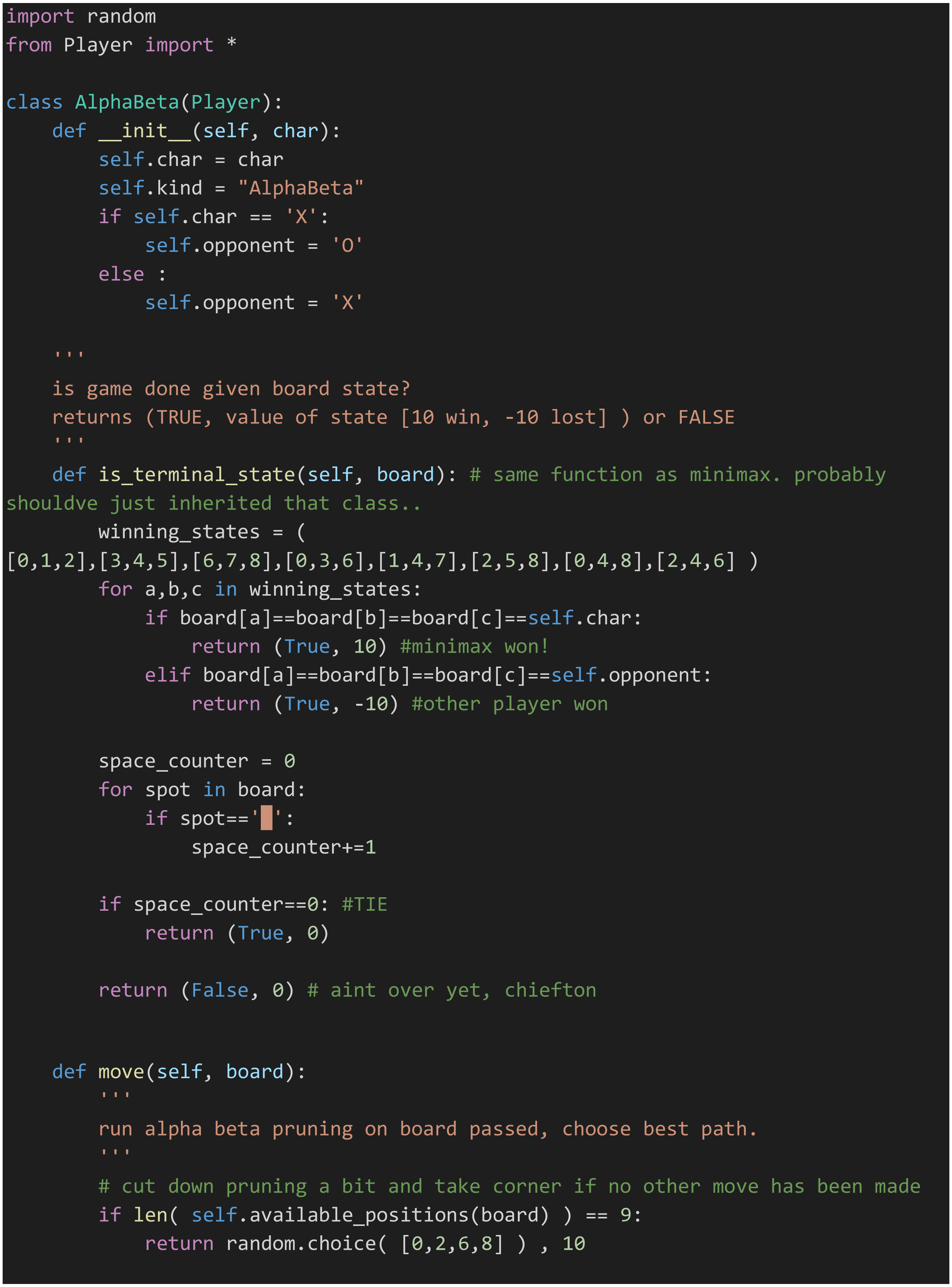
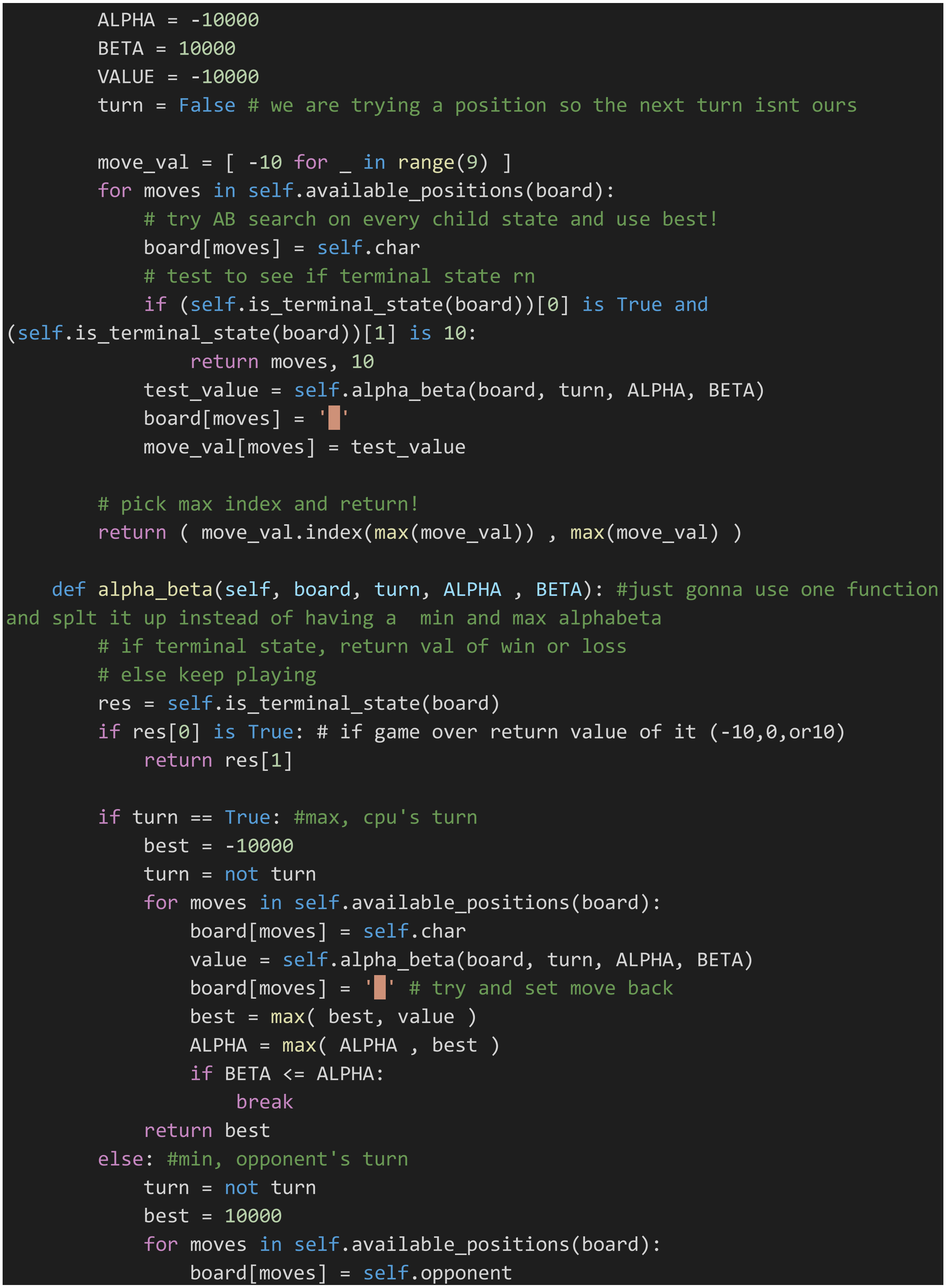


```
import random
from Player import *

class AlphaBeta(Player):
    def __init__(self, char):
        self.char = char
        self.kind = "AlphaBeta"
        if self.char == 'X':
            self.opponent = 'O'
        else :
            self.opponent = 'X'

    '''
    is game done given board state?
    returns (TRUE, value of state [10 win, -10 lost] ) or FALSE
    '''
    def is_terminal_state(self, board): # same function as minimax. probably
shouldve just inherited that class..
        winning_states = (
[0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8],[0,4,8],[2,4,6] )
        for a,b,c in winning_states:
            if board[a]==board[b]==board[c]==self.char:
                return (True, 10) #minimax won!
            elif board[a]==board[b]==board[c]==self.opponent:
                return (True, -10) #other player won

        space_counter = 0
        for spot in board:
            if spot==' ':
                space_counter+=1

        if space_counter==0: #TIE
            return (True, 0)

        return (False, 0) # aint over yet, chiefton


    def move(self, board):
        '''
        run alpha beta pruning on board passed, choose best path.
        '''
        # cut down pruning a bit and take corner if no other move has been made
        if len( self.available_positions(board) ) == 9:
            return random.choice( [0,2,6,8] ) , 10
```



```
        ALPHA = -10000
        BETA = 10000
        VALUE = -10000
        turn = False # we are trying a position so the next turn isnt ours

        move_val = [ -10 for _ in range(9) ]
        for moves in self.available_positions(board):
            # try AB search on every child state and use best!
            board[moves] = self.char
            # test to see if terminal state rn
            if (self.is_terminal_state(board))[0] is True and
(self.is_terminal_state(board))[1] is 10:
                return moves, 10
            test_value = self.alpha_beta(board, turn, ALPHA, BETA)
            board[moves] = ' '
            move_val[moves] = test_value

        # pick max index and return!
        return ( move_val.index(max(move_val)) , max(move_val) )

    def alpha_beta(self, board, turn, ALPHA , BETA): #just gonna use one function
and splt it up instead of having a  min and max alphabeta
        # if terminal state, return val of win or loss
        # else keep playing
        res = self.is_terminal_state(board)
        if res[0] is True: # if game over return value of it (-10,0,or10)
            return res[1]

        if turn == True: #max, cpu's turn
            best = -10000
            turn = not turn
            for moves in self.available_positions(board):
                board[moves] = self.char
                value = self.alpha_beta(board, turn, ALPHA, BETA)
                board[moves] = ' ' # try and set move back
                best = max( best, value )
                ALPHA = max( ALPHA , best )
                if BETA <= ALPHA:
                    break
            return best
        else: #min, opponent's turn
            turn = not turn
            best = 10000
            for moves in self.available_positions(board):
                board[moves] = self.opponent
                value = self.alpha_beta(board, turn, ALPHA, BETA)
                board[moves] = ' '
                best = min( best, value )
                BETA = min( BETA, best )
                if BETA <= ALPHA:
                    break
            return best
```

```
    def available_positions(self, board):
        return [i for i in range(0, 9) if board[i] == ' ']
```