```
#############################################################
#     Tic-Tac-Toe Server                                    #
#     a ttt server that multiple clients may connect to.   #
#                                                           #
#     Noah Jaffe                                            #
#     TCP Socket Programming                                #
#     CMSC 481                                              #
#     11/05/2018                                            #
#############################################################


###########################
# waits for clients to contact it - opens a listining TCP socket and
maintains an open socket until game ends
# when client opens a connection to the server listining port
#     the server begins the game making the first or 2nd move
depending on client
# server closes the listining socket and exits gracefully when ctrl+c
is pressed on command line
#
# NOTES:
# server stores the  current game state
# server must keep the board correctly, not overwriting moves and
knowing when a win has occurred
# It can play as stupidly as you like.
#
# SUBMIT
#     Working documented code.
#           For partial credit, you must be able to handle one client
at a time.
#           For full credit, you must handle multiple clients at a
time.
#     Protocol Specification documenting the messages that are sent
between the client
#           and the server which would allow someone to develop their
own client or
#           server to interact with yours.
###########################

from _thread import *   #_thread for python3, thread for python 2.7
from socket import *
from random import shuffle
import sys, select, struct

#CONSTANTS & GLOBALS
SERVER_MARK = 1
CLIENT_MARK = 2
UNUSED_MARK = 0
```

```
CATS_GAME = -1

ACTIVE_GAMES = [] #the global array to store the active games
UNIQUE_ID_COUNTER = 0   #the global uniqie ID index to ensure no
duplicate games

TTT_SERVER_PORT = 13037
TTT_PRTCL_REQUEST_FIRST_ARGS = "Please send an unsigned int
representing if the client wishes to make the first move.\n\t0 --
sever should go first\n\t1 -- client should go first"
TTT_PRTCL_GOT_FIRST_ARGS_ERR = "Failed to receive proper game
initiation arguments. Terminating connection.\nNext time " +
TTT_PRTCL_REQUEST_FIRST_ARGS
TTT_PRTCL_INSTRUCTIONS = "Welcome to Tic Tac Toe!\nEnter [0-8] for the
position of your move, or 9 to quit:\n0|1|2\n-----\n3|4|5\n-----\n6|7|
8\n"
TTT_PRTCL_INVALID_CLIENT_INPUT = "Invalid input, try again."
TTT_PRTCL_REQUEST_CLIENT_TURN = " | | \n-----\n | | \n-----\n | |
\nEnter [0-8] for the position of your move, or 9 to quit:\n"
TTT_PRTCL_CLIENT_ERR = "Sorry, that was invalid input. Please try
again."
TTT_PRTCL_TERMINATE = 0
TTT_PRTCL_EXPECTING_NO_RESPONSE = 1
TTT_PRTCL_EXPECTING_INT_RESPONSE = 2
TTT_PRTCL_EXPECTING_FIRST_ARGS_RESPONSE = 3
TTT_PRTCL_PACKED_UNSIGNED_INT_SIZE = 4 #4 is the size of a packed '!I'
value
class TTT_Game:

      def __init__(self, conn, addr, uid, turn=SERVER_MARK,
server_char='X', client_char='O'):
            '''
            INITALIZE TIC TAC TOE GAME. Sets the appropriate values
            and creates an empty game board.

            ARGUMENTS:
                  conn -- the socket connection
                  addr -- the connected address
                  uid -- the unique ID of this game
                  turn -- who goes first (default SERVER_MARK)
                  server_char -- the char of the server (default X)
                  client_char -- the char of the client (default O)
            '''

            self.conn = conn
            self.addr = addr
            self.uid = uid
            self.turn = turn
            self.board = []
            for i in range(9):
                  self.board.append(UNUSED_MARK)
            self.server_char = server_char
            self.client_char = client_char

      def reinit_vals(self, turn):
            '''
            Allows for reinitalization of certain values that was not
known
             at game object creation time. Sets to the correct players
turn
             and gives the player who has the first turn's mark to
'X'.

            ARGUMENTS:
                  turn -- valid values are CLIENT_MARK and SERVER_MARK
            '''
            self.turn = turn
            self.server_char = 'O' if turn == CLIENT_MARK else 'X'
            self.client_char = 'X' if turn == CLIENT_MARK else 'O'

      def print_game_info(self):
            '''
            prints the game info.
            '''
            msg = "Connection: \t{0}\nAddress: \t{1}\nGame ID:
\t{2}\nWhos"
            msg += " turn: \t{3}\nBoard: \n" +
self.get_board_as_string()
            turn = 'client' if self.turn == CLIENT_MARK else 'server'
            print(msg.format(self.conn, self.addr, self.uid, turn))

      def check_for_win(self,board=None):
            '''
            Checks if game has been won.
            RETURNS:
                  UNUSED_MARK -- if the game is not over
                  CLIENT_MARK -- if the game has been won by the
client
                  SERVER_MARK -- if the game has been won by the
server
                  CATS_GAME -- if there is no winner and board is full
            '''
            if board is None:
                  board = self.board

            #lazy/easy way of checking...
            #check horizontal win
            if board[0] != UNUSED_MARK and board[0] == board[1] and 
board[1] == board[2]:
                  return board[0]
            elif board[3] != UNUSED_MARK and board[3] == board[4] and 
board[4] == board[5]:
                  return board[3]
            elif board[6] != UNUSED_MARK and board[6] == board[7] and 
board[7] == board[8]:
                  return board[6]

            #check vertical win
            if board[0] != UNUSED_MARK and board[0] == board[3] and 
board[3] == board[6]:
                  return board[0]
            elif board[1] != UNUSED_MARK and board[1] == board[4] and 
board[4] == board[7]:
                  return board[1]
            elif board[2] != UNUSED_MARK and board[2] == board[5] and 
board[5] == board[8]:
                  return board[2]

            #check diagnal win
            if board[0] != UNUSED_MARK and board[0] == board[4] and 
board[4] == board[8]:
                  return board[0]
            elif board[2] != UNUSED_MARK and board[2] == board[4] and 
board[4] == board[6]:
                  return board[2]

            #check if board is not full
            if UNUSED_MARK in board:
                  return UNUSED_MARK
            #else, it must be a cat's game
            return CATS_GAME

            #print("ERROR @ ttts.py::TTT_Game::check_for_win(): 
REACHED END OF CHECK_FOR_WIN WITHOUT RETURNING A VALUE")

      def take_server_turn(self):
            '''
            The server takes a turn.
            version1: in order, no overwrite
            version2: smart version
            RETURNS:
                  True -- if we were able to make a move
                  False -- if we were unable to make a move
            if self.get_turn() == CLIENT_MARK:
                  print("ERROR @
ttts.py::TTT_Game::take_server_turn(): SERVER ATTEMPTED TO TAKE A TURN
DURING PLAYER TURN")
                  return False

            for i, v in enumerate(self.board):
                  if self.check_valid_move(i):
                        self.board[i] = SERVER_MARK
                        return True
            '''
            pos = self.get_server_move()
            if self.check_valid_move(pos):
                  self.board[pos] = SERVER_MARK
                  return True
            else:
                  for i, v in enumerate(self.board):
                        if self.check_valid_move(i):
                              self.board[i] = SERVER_MARK
                              return True
            return False

      def get_board_copy(self, board):
            '''
            ARGUMENTS:
                  board -- a board

            RETURNS:
                  <list[0:8] of int> -- a copy of the board passed in
            '''
            new_board = []
            for i in board:
                  new_board.append(i)
            return new_board

      def server_test_move_for_win(self, board, pos, mark):
            '''
            Checks if game would be won if the mark was placed at a
certian position.

            ARGUMENTS:
                  board -- a board
                  pos -- a position <0 to 8>
                  mark -- the player mark testing for

            RETURNS:
                  UNUSED_MARK -- if the game would not be won
                  CLIENT_MARK -- if the game would be won by the
```

```
client
                  SERVER_MARK -- if the game would be won by the
server
                  CATS_GAME -- if there would be no winner and board
is full
            '''
            test_board = self.get_board_copy(board)
            test_board[pos] = mark
            return self.check_for_win(test_board)

      def server_test_move_for_fork(self, board, pos, mark):
            '''
            Gets a copy of the board passed in, places the mark in
that position, then tests how many
            places could the next move be put to make a win.
            ARGUMENTS:
                  board -- a board
                  pos -- a position <0 to 8>
                  mark -- the player mark testing for

            RETURNS:
                  True -- if the position passed in has more than one
win condition
                  False -- if the position passed in has one or no win
conditions
            '''
            test_board = self.get_board_copy(board)
            test_board[pos] = mark
            win_potential = 0
            for i in range(0, 9):
                  if self.check_valid_move(i) and
self.server_test_move_for_win(board, i, mark) == mark:
                        win_potential += 1
            return win_potential > 1

      def get_server_move(self):
            '''
            ALGORITHM ADAPTED FROM:
https://mblogscode.wordpress.com/2016/06/03/python-naughts-crossestic-
tac-toe-coding-unbeatable-ai/
            RETURNS
                  <int> -- the position to place the server mark
            '''
            #check for server win
            for i in range(0,9):
                  if self.check_valid_move(i) and
self.server_test_move_for_win(self.board, i, SERVER_MARK) ==
SERVER_MARK:
```

```
                        return i

            #check for block to client win
            for i in range(0,9):
                  if self.check_valid_move(i) and
self.server_test_move_for_win(self.board, i, CLIENT_MARK) ==
CLIENT_MARK:
                        return i

            #check for server fork opportunity
            op_pos = list(range(0,9))
            shuffle(op_pos)
            for i in op_pos:
                  if self.check_valid_move(i) and
self.server_test_move_for_fork(self.board, i, SERVER_MARK) ==
SERVER_MARK:
                        return i

            #check for client fork opportunity
            for i in op_pos:
                  if self.check_valid_move(i) and
self.server_test_move_for_fork(self.board, i, CLIENT_MARK) ==
CLIENT_MARK:
                        return i


            #corner & center OP
            op_pos = list(range(0,9,2)) #random order of corners and
center [0, 2, 4, 6, 8]
            shuffle(op_pos)
            for i in op_pos:
                  if self.check_valid_move(i):
                        return i

            #sides if neccisary
            op_pos = list(range(1,9,2)) #random order of sides [1, 3,
5, 7]
            shuffle(op_pos)
            for i in op_pos:
                  if self.check_valid_move(i):
                        return i


      def make_client_move(self, pos):
            '''
            Attempts to apply the client's move.
            RETURNS:
                  True -- if valid move, and move applied
                  False -- if invalid move, and move not applied
            '''
            if self.check_valid_move(pos):
                  self.board[pos] = CLIENT_MARK
                  return True
            else:
                  return False

      def get_board_as_string(self):
            '''
            Returns a printable format of the game board.
            example:
             |X|O
            -----
            X| |O
            -----
            O| |X

            RETURNS:
                  <string> -- A string of the board that can be
printed
            '''

            board_str = []
            vert_seperator = '|'
            horz_seperator = '-----'
            positional_str = ["0|1|2", "3|4|5", "6|7|8"]
            positional_pos = 0
            blank = ' '

            for i in range(0,9):
                  #set board value
                  if self.board[i] == UNUSED_MARK:
                        board_str.append(blank)
                  elif self.board[i] == SERVER_MARK:
                        board_str.append(self.server_char)
                  elif self.board[i] == CLIENT_MARK:
                        board_str.append(self.client_char)

                  #add seperators
                  if i == 2 or i == 5:
                        #add on the positional number guide
                        board_str.append("\t\t" +
positional_str[positional_pos])
                        positional_pos += 1
                        #add the line seperators
                        board_str.append("\n" + horz_seperator + "\t\t"
+ horz_seperator + "\n")
```

```
                  elif ((i % 3 == 0) or (i % 3 == 1)):
                        board_str.append(vert_seperator)
            board_str.append("\t\t" + positional_str[positional_pos])
            #return the board as a string
            return ''.join(board_str)


      def check_valid_move(self,pos):
            '''
            RETURNS:
                  <bool> -- if the position on the board is not
occupied
            '''
            try:
                  return self.board[pos] == UNUSED_MARK
            except:
                  return False

      def get_turn(self):
            '''
            RETURNS:
                  <int> -- X_MARK value of whos turn it is
            '''
            return self.turn

      def change_turn(self):
            '''
            Sets the game's turn value to the other player.
            Sets to client if currently on the server,
            and to the server if currently on the client.
            '''
            self.turn = CLIENT_MARK if self.get_turn() == SERVER_MARK
else SERVER_MARK



def validate_TTT_PRTCL(protocol_id, recv):
      '''
      Validates client input for a TTT protocol

      ARGUMENTS:
            protocol_id -- a string with which protocol to check
            recv -- the user input

      RETURNS:
            True -- valid input
            False -- invalid input
      '''
      if protocol_id == "SRVR_RECV_REQUEST_SINGLE_DIGIT_INPUT":
            #INVALID IF: the value is not a single digit
            try:
                  return str(recv).isdigit() and len(str(recv)) == 1
            except:
                  return False

def send_server_response(conn, expecting_response_val, message):
      '''
      Sends three messages to the client.
      1. <int> a message length (packed)
      2. <string> the message (encoded)
      3. <int> an expected response value (packed)

      SENDING FROM SERVER TO CLIENT:
            pack message response length '!I'
            SEND PACKED: MESSAGE RESPONSE LENGTH
            SEND: MESSAGE
            pack expecting response val '!I'
            SEND PACKED: EXPECTING RESPONSE VAL

      ARGUMENTS:
            conn -- the connection
            expecting_response_val -- the response expected from the
client
            message -- a message to send to the client

      '''
      #pack message response length '!I'
      #SEND PACKED: MESSAGE RESPONSE LENGTH
      conn.sendall(struct.pack('!I', len(message)))
      #SEND: MESSAGE
      conn.sendall(message.encode())
      #pack expecting response val '!I'
      #SEND PACKED: EXPECTING RESPONSE VAL
      conn.sendall(struct.pack('!I', expecting_response_val))


def get_client_response(conn):
      '''
      Recv's one message from the client:
      1. <unsigned int> a response value (packed)

      RECIEVING FROM CLIENT TO SERVER:
            RECV PACKED: SINGLE DIGIT VAL
            unpack '!I'

      ARGUMENTS:
            conn -- the connection

      RETURNS:
            <unsigned int> -- a single digit value
            None -- if there was en error reading the response
      '''
      #recv message header
      digit_buff = recvall(conn, TTT_PRTCL_PACKED_UNSIGNED_INT_SIZE)
      #receive bytes of data for the packed unsigned int from client
      try:
            val,  = struct.unpack('!I', digit_buff)
            return val
      except:
            print ("ERROR @ ttts.py::get_client_response(): RECV
RESPONSE OF NONE")
      return None


def recvall(conn, size):
      '''
      Receives messages of a specific size (size) from the connection
(conn)

      ARGUMENTS:
            conn -- the connection
            size -- number of bytes to read

      RETURNS:
            <bytes> -- the (packed/encoded) message from the client
            None -- if connection failed before reading in the message
      '''
      #recv message header
      encoded_msg = b''
      while size:
            temp = conn.recv(size)
            if not temp:
                  return None
            encoded_msg += temp
            size -= len(temp)

      return encoded_msg

def remove_active_game(active_game):
      '''
      Removes a game from the ACTIVE_GAMES list and closes that
connection.

      ARGUMENTS:
            active_game -- the game to remove

      RETURNS:
            True -- successful connection closure and active game
removed
            False -- something went wrong during closure and removal
      '''
      try:
            active_game.conn.close()
            ACTIVE_GAMES.remove(active_game)
            return True
      except:
            return False

def game_thread(conn, addr, active_game):
      '''
      The game logic. Gets the command line args from client and
initalizes
      the game info. Sends instructions to client.
      Plays the game between the client and server. Sends a message
with the
      board and the winner. Closes the connection.

      ARGUMENTS:
            conn -- the server socket connection
            addr -- the address of the connection
            active_game -- the TTT_Game object with the game info
      '''
      print("Starting new game...")
      active_game.print_game_info()

      #SEND REQUEST WHO WILL BE GOING FIRST. CLIENT OR SERVER
      send_server_response(conn,
TTT_PRTCL_EXPECTING_FIRST_ARGS_RESPONSE, TTT_PRTCL_REQUEST_FIRST_ARGS)
      #GET RESPONSE
      cmd_line_args = get_client_response(conn)

      #validate the messages, if invalid terminate connection
      if not
validate_TTT_PRTCL("SRVR_RECV_REQUEST_SINGLE_DIGIT_INPUT",
cmd_line_args):
            print("ERROR @ ttts.py::game_thread(): INVALID COMMAND
LINE ARGS.")

            #SEND TERMINATION ERROR MESSAGE
            err_msg = TTT_PRTCL_CLIENT_ERR + "\n" +
TTT_PRTCL_REQUEST_FIRST_ARGS
            send_server_response(conn, TTT_PRTCL_TERMINATE, err_msg)
            return remove_active_game(active_game)

      #valid arguments...
      #if client goes first, update the game status thereby giving the
first player the 'X' mark
      if cmd_line_args:
            active_game.reinit_vals(CLIENT_MARK)

      #we have all the info we need, time to start the game!

      #sends instructions to client
      #SENDS MESSAGE INSTRUCTIONS
      send_server_response(conn, TTT_PRTCL_EXPECTING_NO_RESPONSE,
TTT_PRTCL_INSTRUCTIONS)

      print("********STARTING GAME ID: {0}
********".format(active_game.uid))
      active_game.print_game_info()      #TODO: DELETE AFTER DEBUG
      endgame_status = active_game.check_for_win()

      #start game process
      while endgame_status == UNUSED_MARK:
            #if client turn
            if active_game.get_turn() == CLIENT_MARK:
                  #send game board and instructions to client, along
with expecting int response
                  #SEND REQUEST FOR CLIENT MOVE
                  message = active_game.get_board_as_string() +
TTT_PRTCL_REQUEST_CLIENT_TURN[29:]
                  send_server_response(conn,
TTT_PRTCL_EXPECTING_INT_RESPONSE, message)

                  #get and validate client response
                  #get int response from client
                  #GET RESPONSE
                  client_move = get_client_response(conn)

                  #if client move is None, it most likely means user
process terminated,
                  #or if client move is 9, it means they requested to
exit,
                  #so terminate connetion
                  if client_move is None or client_move == 9:
                        print("ERROR @ ttts.py::game_thread(): GOT USER
RESPONSE OF " + str(client_move) + ". TERMINATING SERVER-CLIENT
SESSION." + "\nGAME ID: {0}".format(active_game.uid))
                        #close down game
                        return remove_active_game(active_game)
                  #else if we got an invalid response, log error and
continue
                  elif not
validate_TTT_PRTCL("SRVR_RECV_REQUEST_SINGLE_DIGIT_INPUT",
client_move):
                        print("ERROR @ ttts.py::game_thread(): GOT
INVALID INPUT FROM CLIENT." + "\nGAME ID:
{0}".format(active_game.uid))

                  #else we got a valid response
                  else:
                        #attempt to make the client move
                        if
active_game.make_client_move(int(client_move)):
                              #change turns if valid move
                              active_game.change_turn()
                        else:
                              #else if move falied. print err msg and
cry. dont change turns so that we ask them
                              #again on the next loop
                              print("ERROR @ ttts.py::game_thread():
FAILED TO MAKE CLIENT MOVE." + "\nGAME ID:
{0}".format(active_game.uid))
            #if server turn
            elif active_game.get_turn() == SERVER_MARK:
                  #request SERVER to take turn
                  if active_game.take_server_turn():
                        #attempt to make a move, if successful then
change turns, and send the updated game board to the client
                        active_game.change_turn()
                  else:
                        #else if move falied. print err msg and cry.
dont change turns so that we ask them again on next loop
                        print("ERROR @ ttts.py::game_thread(): FAILED
TO MAKE SERVER MOVE." + "\nGAME ID: {0}".format(active_game.uid))
                        active_game.print_game_info()

            #end of turn, check board to see if game has ended or not
            endgame_status = active_game.check_for_win()

            #end while loop


      print("********ENDING GAME ID: {0}
********".format(active_game.uid))
      active_game.print_game_info()      #TODO: DELETE AFTER DEBUG
      #end game logic
      endgame_message = active_game.get_board_as_string()
      if endgame_status == CLIENT_MARK:
            endgame_message = endgame_message + "\nCongratulations!
You[{client_char}] Won!\nWINNER: {client_char}\nCLIENT:
{client_char}\nSERVER: {server_char}".format(client_char =
active_game.client_char, server_char = active_game.server_char)
      elif endgame_status == SERVER_MARK:
            endgame_message = endgame_message + "\nSorry!
Server[{server_char}] Won!\nWINNER: {server_char}\nCLIENT:
{client_char}\nSERVER: {server_char}".format(client_char =
active_game.client_char, server_char = active_game.server_char)
      elif endgame_status == CATS_GAME:
            endgame_message = endgame_message + "\nSorry, Cat's game!
You[{client_char}] Tied!\nWINNER: None\nCLIENT: {client_char}\nSERVER:
{server_char}".format(client_char = active_game.client_char,
server_char = active_game.server_char)
      else:
            #print error message
            endgame_message = endgame_message + "\nENDGAME ERROR:
\nCLIENT: {client_char}\nSERVER: {server_char} ".format(client_char =
active_game.client_char, server_char = active_game.server_char)

      #SEND TERMINATION WITH END GAME MESSAGE
      send_server_response(conn, TTT_PRTCL_TERMINATE, endgame_message)

      return remove_active_game(active_game)

def main():
      global UNIQUE_ID_COUNTER
      #STREAM type of connection
      server_socket = socket(AF_INET, SOCK_STREAM)
      server_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)


      server_socket.bind(('',TTT_SERVER_PORT))
      server_socket.listen(5) #allow for 5 failures... not entirely
sure how backlog works

      #list of connections
      print ('The server is ready to receive connections')
      try:
            while 1:
                  #accept a connection
                  conn, addr = server_socket.accept()
                  print("Host:{0} \tPort:{1} connected. UNIQUE Game
ID: {2}".format(addr[0], addr[1], UNIQUE_ID_COUNTER))
                  #create a game state
                  ACTIVE_GAMES.append(TTT_Game(conn, addr,
UNIQUE_ID_COUNTER))
                  UNIQUE_ID_COUNTER += 1

                  #create new thread for this client
                  start_new_thread(game_thread,(conn,addr,
ACTIVE_GAMES[-1]))


      except KeyboardInterrupt:
            #dont crash program... allow for cleanup
            print("\nCLOSING DOWN TIC-TAC-TOE SERVER")
            for v in ACTIVE_GAMES:
                  v.conn.close()
            server_socket.close()
            sys.exit(0)

if __name__ == '__main__':
    main()
```