# Code

### censored-word.txt

### db.sqlite3

### manage.py

## accounts

### forms.py

### models.py

### serializers.py

### urls.py

### views.py

## leaderboard

### models.py

### urls.py

### views.py

## poker

### consumers.py

### models.py

### poker.py

import random

from .models import Players, Room

from channels.layers import get\_channel\_layer

from asgiref.sync import async\_to\_sync

from tables.models import Table

import time

from accounts.models import CustomUser

import sys

from datetime import datetime, timezone

class Player:

    def \_\_init\_\_(self, username, money):

        self.\_\_username = username

        self.\_\_money = money

        self.\_\_hand = []

        self.\_\_playerIn = True

        self.\_\_callAmount = self.\_\_putIn = 0

        self.\_\_handStrength = ''

        self.\_\_moneyWon = 0

    @property

    def username(self):

        return self.\_\_username

    @property

    def money(self):

        return self.\_\_money

    def increaseMoney(self, amount):

        self.\_\_money += amount

        self.\_\_moneyWon += amount

    @property

    def hand(self):

        return self.\_\_hand

    @hand.setter

    def hand(self, hand):

        self.\_\_hand = hand

    @property

    def playerIn(self):

        return self.\_\_playerIn

    @property

    def handStrength(self):

        return self.\_\_handStrength

    @handStrength.setter

    def handStrength(self, strength):

        self.\_\_handStrength = strength

    @property

    def moneyWon(self):

        return self.\_\_moneyWon

    @property

    def callAmount(self):

        return self.\_\_callAmount

    @callAmount.setter

    def callAmount(self, callAmount):

        if callAmount >= 0:

            self.\_\_callAmount = callAmount

        else:

            raise Exception('call amount less than 0')

    @property

    def putIn(self):

        return self.\_\_putIn

    def decreasePutIn(self, amount):

        self.\_\_putIn -= amount

    def fold(self):

        self.\_\_playerIn = False

    def newRound(self):

        self.\_\_playerIn = True

        self.\_\_callAmount = self.\_\_putIn = 0

        self.\_\_moneyWon = 0

        self.\_\_hand = []

        self.\_\_handStrength = ''

    def call(self, moneyToPutIn):

        if self.\_\_money > moneyToPutIn:

            self.\_\_money -= moneyToPutIn

        else: #all-in situation

            moneyToPutIn = self.\_\_money

            self.\_\_money = 0

        self.\_\_putIn += moneyToPutIn

        self.\_\_callAmount = 0

        return moneyToPutIn

class Cards:

    def \_\_init\_\_(self, players):

        TESTING = False #change for testing purposes

        self.\_\_players = players

        self.\_\_deck = []

        self.\_\_comCards = []

        self.makeDeck()

        if not TESTING:

            self.hands()

        else:

            self.makeHandsMan()

    def makeDeck(self):

        self.\_\_deck = [[k, j] for j in range(4) for k in range(2,15)]

    def hands(self):

        random.shuffle(self.\_\_deck)

        self.\_\_comCards = self.\_\_deck[:5][:]

        del self.\_\_deck[:5]

        for player in self.\_\_players:

            playerHand = self.\_\_deck[:2][:]

            del self.\_\_deck[:2]

            player.hand = playerHand

    @property

    def comCards(self):

        return self.\_\_comCards

    #converts cards into human readable form

    def convert(hand):

        numbers = [[11, 'J'], [12, 'Q'], [13, 'K'], [14, 'A'], [1, 'A']]

        suits = ['♥', '♦', '♠', '♣']

        convertHand = ''

        for a in range(len(hand)):

            add = False

            for item in numbers:

                if hand[a][0] == item[0]:

                    convertHand += item[1]

                    add = True

            if not add:

                convertHand += str(hand[a][0])

            for b in range(4):

                if hand[a][1] == b:

                    convertHand += (suits[b] + ' ')

        return convertHand

    def makeHandsMan(self):

        self.\_\_comCards = [[5, 3], [7, 1], [13, 2], [6, 2], [11, 2]]

        hands = [[

                [7, 3], [7, 0] #first player hand

            ], [

                [13, 3], [4, 2] #second player hand etc

            ], [

                [2, 1], [4, 3]

        ]]

        for player, hand in zip(self.\_\_players, hands):

            player.hand = hand

class Poker:

    def \_\_init\_\_(self, players, C):

        self.players = players

        self.C = C

        self.strengthList = ['High Card', 'Pair', 'Two Pair',\

        'Three of a kind', 'Straight', 'Flush', 'Full House', 'Four of a kind',\

        'Straight Flush', 'Royal Flush']

        self.win = []

        self.\_\_playerWin = []

        self.split = []

        self.handStrength()

        self.winQueue()

    @property

    def playerWin(self):

        return self.\_\_playerWin

    @playerWin.setter

    def playerWin(self, playerWin):

        self.\_\_playerWin = playerWin

    def handStrength(self):

        for player in self.players:

            self.orderHand = []

            self.strength = 0

            hand = player.hand + self.C.comCards

            hand.sort(reverse=True)

            self.checkRank(hand)

            self.flush(hand, 5)

            tempHand = self.addAceAsOne(hand)

            self.straight(tempHand)

            player.handStrength = self.strengthList[self.strength]

            self.win.append([self.strength, player, self.orderHand[:]])

        self.win.sort(key = lambda x: x[0], reverse=True)

        self.clash()

    def checkRank(self, hand):

        #determines whether cards are a pair or 3 of a kind,

        #alternatively a two pair or fullhouse

        def twoThree(pStrength2, pStrength3):

            if len(sameRank[0]) == 3:

                return pStrength3

            else:

                return pStrength2

        i = 0

        sameRank = []

        while i < 6:

            temp = [hand[i]]

            try:

                #adds cards of same rank to temp

                while hand[i][0] == hand[i+1][0]:

                    temp.append(hand[i+1])

                    i+=1

                else:

                    i+=1

            except IndexError:

                pass

            #if more than one card of same rank add to sameRank

            if len(temp) > 1:

                sameRank.append(temp[:])

        #sort by length of same ranked cards,

        #e.g. 4 of a kind > 3 of a kind > pair

        sameRank.sort(key = lambda x: len(x), reverse=True)

        sameRank = sameRank[:2]

        if len(sameRank) != 0:

            #four of a kind

            if len(sameRank[0]) == 4:

                sameRank = sameRank[0]

                self.strength = 7

            #two pair or full house

            elif len(sameRank) == 2:

                self.strength = twoThree(2, 6)

            #pair or three of a kind

            else:

                self.strength = twoThree(1, 3)

        #put all cards from sameRank in 1D array

        temp = []

        for cards in sameRank:

            for card in cards:

                temp.append(card)

        #add cards not included in sameRank

        for card in hand:

            if card not in temp:

                temp.append(card)

        #make orderHand

        self.orderHand = temp[:5]

    def flush(self, hand, pStrength):

        #iterates over all 4 suits

        for i in range(4):

            flush = []

            for card in hand:

                #appends card to flush array if same suit

                if card[1] == i:

                    flush.append(card)

            if len(flush) == 5 and self.strength < pStrength:

                self.strength = pStrength

                self.orderHand = flush[:5]

    def addAceAsOne(self, hand):

        #temporarily adds ace as 1

        for card in hand:

            if 14 in card:

                if [1, card[1]] not in hand:

                    hand.append([1, card[1]])

        return hand

    def straight(self, hand):

        straightHand = []

        for j in range(len(hand)):

            if len(hand) > j+1:

                #as hand sorted reversed compare less 1 to the card

                #rank below it

                if hand[j][0]-1 == hand[j+1][0]:

                    if len(straightHand) == 0:

                        straightHand.append(hand[j])

                    straightHand.append(hand[j+1])

                elif hand[j][0] == hand[j+1][0]:

                    #for straight flushes make a new straight check without

                    #duplicate card evey time same number is found

                    self.straight(hand[:j+1][:] + hand[j+2:][:])

                    self.straight(hand[:j][:] + hand[j+1:][:])

                else:

                    straightHand = []

                if len(straightHand) == 5:

                    #checks if straight is straight flush

                    self.flush(straightHand, 8)

                    if self.strength < 4:

                        self.strength = 4

                        self.orderHand = straightHand

        #if the straight flush is Ace to 10 then it is a royal flush

        if self.strength == 8 and self.orderHand[0][0] == 14:

            self.strength = 9

    #binary sort but adds the players to repeated array if values are the same

    def clash(self):

        repeated = []

        flip = True

        while flip:

            flip = False

            for a in range(len(self.win)):

                if len(self.win) > a+1:

                    if self.win[a][0] == self.win[a+1][0]:

                        flip = self.sorting(self.win[a][2], self.win[a+1][2])

                        if flip == 'split':

                            flip = False

                            repeated.append(self.win[a][1])

                            repeated.append(self.win[a+1][1])

                        elif flip:

                            temp = self.win[a]

                            self.win[a] = self.win[a+1][:]

                            self.win[a+1] = temp[:]

        self.splitWork(repeated)

    def sorting(self, hand1, hand2):

        a = 0

        #finds the first card where the values differ

        while hand1[a][0] == hand2[a][0] and a < 4:

            a+=1

        if hand1[a][0] > hand2[a][0]:

            return False

        elif hand1[a][0] < hand2[a][0]:

            return True

        else:

            return 'split'

    #adds players with the the same hand to split in an array

    def splitWork(self, repeated):

        for a in range(0, len(repeated), 2):

            if a - 1 >= 0:

                #the players are added in pairs, so if a player is the same as

                #a player in the previous iteration then all 3 players in the

                #current and previous iteration have the same strength hand.

                #So the other player in the current iteration is appended to the

                #previous iteration array

                if repeated[a] == repeated[a-1]:

                    #-1 is the index of the last item in the array

                    self.split[-1].append(repeated[a+1])

                else:

                    self.split.append([repeated[a], repeated[a+1]])

            else:

                self.split.append([repeated[a], repeated[a+1]])

    #adds each player to playerWin in an array

    def winQueue(self):

        for strength, player, hand in self.win:

            added = False

            for players in self.split:

                if player in players:

                    #if players in split it adds the split array instead

                    self.playerWin.append(players)

                    added = True

            if not added:

                self.playerWin.append([player])

        #remove duplicate split arrays

        self.playerWin = [tuple(x) for x in self.playerWin]

        self.playerWin = list(dict.fromkeys(self.playerWin))

        self.playerWin = [list(x) for x in self.playerWin]

        print('playerWin', self.playerWin)

class Game:

    def \_\_init\_\_(self, minimumBet, dealer, tableGroup, table, playersInGame):

        self.minimumBet = minimumBet

        self.dealer = self.turnIndex = self.better = dealer

        self.tableGroup = tableGroup

        self.table = table

        self.players = playersInGame

        self.winners = []

        self.noOfPlayers = len(self.players)

        self.comCount = 0

        self.pot = 0

        self.instantiateCardsPoker()

        self.play()

    def instantiateCardsPoker(self):

        self.C = Cards(self.players)

        self.P = Poker(self.players, self.C)

    def makeComCards(self):

        if self.comCount == 0:

            self.comCards = ''

            message = ''

        if self.comCount == 1:

            self.comCards = Cards.convert(self.C.comCards[:3])

            message = 'Flop: '

        elif self.comCount == 2:

            self.comCards = Cards.convert(self.C.comCards[:4])

            message = 'Turn: '

        elif self.comCount == 3:

            self.comCards = Cards.convert(self.C.comCards[:])

            message = 'River: '

        self.comCount+=1

        return message

    def sendComCards(self, message):

        message += self.comCards

        if message != '':

            self.sendMessage(message, self.tableGroup)

    def checkNotAllFolded(self):

        count = 0

        for player in self.players:

            if player.playerIn:

                count += 1

        if count > 1:

            return True

        else:

            return False

    def nextTurn(self):

        self.turnIndex = (self.turnIndex+1)%self.noOfPlayers

        self.turn = self.players[self.turnIndex]

    def getPlayer(self, player):

        try:

            userInstance = CustomUser.objects.get(username=player.username)

            player = Players.objects.get(user\_id=userInstance.id)

        except Players.DoesNotExist:

            self.getRoom()

            return (False, '')

        return (True, player)

    def getRoom(self):

        try:

            self.room = Room.objects.get(table=self.table)

        except Room.DoesNotExist:

            print('everyone left')

            self.table.lastUsed = datetime.now(timezone.utc)

            self.table.save()

            sys.exit()

    def blinds(self):

        sb = self.addRaiseAmount(self.minimumBet)

        self.nextTurn()

        bb = self.addRaiseAmount(self.minimumBet)

        self.nextTurn()

        message = self.turn.username + ' posted BB (' + str(bb + sb) + ')\n'

        message += self.turn.username + ' posted SB (' + str(sb) + ')'

        self.sendMessage(message, self.tableGroup)

    def sendCards(self):

        for player in self.players:

            if player.playerIn:

                hand = Cards.convert(player.hand)

                async\_to\_sync(get\_channel\_layer().group\_send)(

                    player.username,

                    {

                        'type': 'cards',

                        'hand': hand,

                        'comCards': self.comCards,

                        'dealer': self.players[self.dealer].username,

                        'moneyInTable': str(player.money)

                    }

                )

    def getChoice(self):

        putIn = str(self.turn.callAmount)

        async\_to\_sync(get\_channel\_layer().group\_send)(

            self.turn.username,

            {

                'type': 'playerTurn',

                'putIn': putIn,

            }

        )

        playerLeft = False

        self.getRoom()

        while self.room.action is None and not playerLeft:

            self.getRoom()

            #everyone leaves while its your turn

            if self.table.getNoOfPlayers() == 1:

                self.room.action = 'c'

                self.room.save()

                self.choice = 'c'

            elif self.room.action is not None:

                #the first character is the action the user wants to take

                #after that it is the optional raiseAmount

                self.choice = self.room.action[0]

                if self.choice == 'r':

                    try:

                        self.raiseAmount = self.room.action[1:]

                        if not int(self.raiseAmount) > 0:

                            raise ValueError()

                    except ValueError:

                        self.sendMessage('Raise amount must be a positive integer', self.turn.username)

                        self.makeTurn()

            if not self.getPlayer(self.turn)[0]:

                self.choice = 'f'

                playerLeft = True

                print(self.turn.username, 'left')

        self.room.action = None

        self.room.save()

    def makeTurn(self):

        playerExists, player = self.getPlayer(self.turn)

        if playerExists:

            player.turn = True

            player.save()

            self.getChoice()

        else:

            self.choice = 'f'

    def makeChoice(self):

        money = 0

        if self.choice == 'c':

            money = self.turn.call(self.turn.callAmount)

            self.pot += money

        elif self.choice == 'r':

            money = self.addRaiseAmount(int(self.raiseAmount))

        elif self.choice == 'f':

            self.turn.fold()

        self.makeMessage(money)

    def addRaiseAmount(self, raiseAmount):

        self.better = self.turnIndex

        callAmount = self.turn.call(self.turn.callAmount)

        raiseAmount = self.turn.call(raiseAmount)

        self.pot += (raiseAmount + callAmount)

        for player in self.players:

            if self.turn != player:

                player.callAmount += raiseAmount

        return raiseAmount

    def updateDBMoney(self):

        for user in self.players:

            playerExists, player = self.getPlayer(user)

            if playerExists:

                player.moneyInTable = user.money

                player.save()

    def makeMessage(self, money):

        if self.choice == 'f':

            message = self.turn.username + ' folded'

        elif self.choice == 'r':

            if self.turn.money == 0:

                message = self.turn.username + ' went all-in'

            else:

                message = self.turn.username + ' raised ' +  str(money)

        if self.choice == 'c':

            if money == 0:

                message = self.turn.username + ' checked'

            else:

                message = self.turn.username + ' called ' + str(money)

        self.sendMessage(message, self.tableGroup)

    def sendMessage(self, message, group):

        async\_to\_sync(get\_channel\_layer().group\_send)(

            group,

            {

                'type': 'pokerMessage',

                'message': message,

                'pot': str(self.pot),

            }

        )

    def checkMultiplePlayersIn(self):

        count = 0

        for player in self.players:

            if player.money > 0:

                count+=1

        if count > 1:

            if self.table.getNoOfPlayers() > 1:

                return True

        return False

    def makeWinnerMessage(self):

        self.message = '\n------------------------------------------'

        showHands = []

        startIndex = currentIndex = (self.dealer+1)%self.noOfPlayers

        winningIndex = 999

        firstRun = True

        #iterates through each player from dealers left as they show first

        while currentIndex != startIndex or firstRun:

            firstRun = False

            for a in range(len(self.P.playerWin)):

                if self.players[currentIndex] in self.P.playerWin[a]:

                    currentWin = a

            if self.players[currentIndex].playerIn and currentWin <= winningIndex:

                winningIndex = currentWin

                playerStats = {

                    'username': self.players[currentIndex].username,

                    'moneyWon': self.players[currentIndex].moneyWon

                }

                if self.checkNotAllFolded():

                    playerStats['hand'] = Cards.convert(self.players[currentIndex].hand)

                    playerStats ['strength'] = ': ' + self.players[currentIndex].handStrength + ' '

                else:

                    playerStats['hand'] = ''

                    playerStats ['strength'] =  ''

                showHands.append(playerStats)

            currentIndex = (currentIndex+1)%self.noOfPlayers

        for player in showHands:

            winnings = ''

            if player['moneyWon'] != 0:

                winnings = ' won ' + str(player['moneyWon'])

            self.message += '\n' + player['username'] + winnings + player['strength'] + player['hand']

        self.message += '\n------------------------------------------\n'

    def distributeMoney(self, players, winners, pot):

        sortedPlayers = sorted(players, key = lambda x: x.putIn)

        winners.sort(key = lambda x: x.putIn)

        if len(winners) != 0:

            money = sortedPlayers[0].putIn

            #money given out equal to the minimum players putIn

            #or pot if in the oddMoney recursion

            moneyMade = money \* len(sortedPlayers)

            if moneyMade > pot:

                moneyMade = pot

                money = pot // len(sortedPlayers)

            #if the money cannot be shared equally

            oddMoney = moneyMade % len(winners)

            if oddMoney != 0:

                print('odd money in pot:', str(oddMoney))

                #share the money between the all the winners except the last

                a = -1

                while players[a] not in winners:

                    a-=1

                print('removing winner:', players[a].username)

                tempWin = winners[:]

                tempWin.remove(players[a])

                pot += self.distributeMoney(players[:], tempWin, oddMoney)

            #decrease each players putIn by the min players putIn

            #increase each winners by the (min players putIn \* players)// no winners

            moneyWon = moneyMade // len(winners)

            for player in sortedPlayers:

                player.decreasePutIn(money)

                if player in winners:

                    player.increaseMoney(moneyWon)

            #decrease pot by money given out

            pot -= moneyMade

            #delete minimum putIn player

            players.remove(sortedPlayers[0])

            if winners[0] == sortedPlayers[0]:

                del winners[0]

            pot = self.distributeMoney(players, winners, pot)

        return pot

    def winner(self):

        a = 0

        while self.pot != 0:

            for player in self.P.playerWin[a]:

                if player.playerIn:

                    self.winners.append(player)

            print('winners', self.winners)

            self.pot = self.distributeMoney(self.players[:], self.winners[:], self.pot)

            print(self.pot)

            self.updateDBMoney()

            a+=1

        self.makeWinnerMessage()

        self.sendMessage(self.message, self.tableGroup)

    def play(self):

        print('in game')

        self.nextTurn()

        for a in range(4):

            #one to the dealers left

            self.better = (self.dealer+1)%self.noOfPlayers

            firstRun = True

            if a == 0:

                self.blinds()

            message = self.makeComCards()

            self.sendCards()

            if self.checkNotAllFolded():

                if self.checkMultiplePlayersIn():

                    self.sendComCards(message)

                    while (self.turnIndex != self.better or firstRun):

                        self.updateDBMoney()

                        self.sendCards()

                        firstRun = False

                        if self.turn.money != 0 and self.turn.playerIn:

                            self.makeTurn()

                            self.makeChoice()

                        self.nextTurn()

                elif a == 3:

                    self.sendComCards(message)

                    self.sendCards()

        self.winner()

def addPlayer(room, table, username):

    player = CustomUser.objects.get(username=username)

    playerInstance = Players.objects.create(user=player, room=room, moneyInTable=table.buyIn)

    player.money -= table.buyIn

    player.save()

def makePlayerOrder(playersInGame, players):

    for player in playersInGame:

        #sets all the Player objects back to their base values

        player.newRound()

        #check whether player in playersInGame is in players

        #if not, the player has left

        if not any(x for x in players if x.user.username == player.username):

            playersInGame.remove(player)

    for newPlayer in players:

        #check whether Player object is not already in playersInGame

        #if so, new player has joined the table

        if newPlayer.moneyInTable > 0 and not any(x for x in playersInGame if x.username == newPlayer.user.username):

            P = Player(newPlayer.user.username, newPlayer.moneyInTable)

            print(P.username, 'has joined')

            playersInGame.append(P)

    print('playersInGame:', playersInGame)

def startGame(table):

    TESTING = False

    playersInGame = []

    dealer = 0

    tableGroup = 'table\_' + str(table.pk)

    while True:

        #waits until their is more than one player in the table to start

        table.refresh\_from\_db()

        while table.getNoOfPlayers() == 1:

            table.refresh\_from\_db()

            time.sleep(0.2)

        #if single player leaves table before anyone joins

        if table.getNoOfPlayers() == 0:

            print('player left, not in game')

            table.lastUsed = datetime.now(timezone.utc)

            table.save()

            sys.exit()

        #gets players in table

#table group is the primary key of Room

        players = Players.objects.filter(room\_id=table.id)

        makePlayerOrder(playersInGame, players)

        if not TESTING:

            dealer = (dealer+1)%len(playersInGame)

        #starts game

        Game((table.buyIn)//100, dealer, tableGroup, table, playersInGame)

        time.sleep(0.4)

def main(pk, username):

    table = Table.objects.get(pk=pk)

    #check to see if table exists

    try:

        room = Room.objects.get(table=table)

        addPlayer(room, table, username)

    #if room dosen't exist create one

    except Room.DoesNotExist:

        room = Room.objects.create(table=table)

        addPlayer(room, table, username)

        startGame(table)

### urs.py

### views.py

### management/ commands

#### cleartables.py

## project

### routing.py

### settings.py

### urls.py

## rules

### urls.py

### views.py

## tables

### consumers.py

### forms.py