Team 8

Operations Manual

Conor Sheil

Table of Contents

Introduction

Our project utilizes various languages and technologies. The game itself can be accessed from the HTML page (http://blackjackorbust.000webhostapp.com/). The website acts as a hub for the game, containing all information needed to be able to play. The HTML pages are decorated with CSS stylesheets, which are used on every page on the website. The pages themselves are run off PHP files. The singleplayer version of the game can be accessed from the singleplayer page on the website.

The singleplayer page contains a canvas in which the game is played. The game is written in JavaScript, the user interacts with the canvas by using mouse-clicks and then number keys on the keyboard. This version of the game contains a vibrant background which was designed by a member of our team, along with the images of the cards. The player can compete against a programmed dealer. The options to raise your bet, and select the amount of decks being used are also available.

The website also offers a login system, this allows the game to keep track of specific players statistics from playing the single player game. These statistics are accessible from the ‘Account’ page on the site. This page allows for keeping track of games won/lost and virtual ‘money’ being won/lost, the current balance of the player is also displayed. It is impossible to access the game without being logged in. An account can be created from the register page.

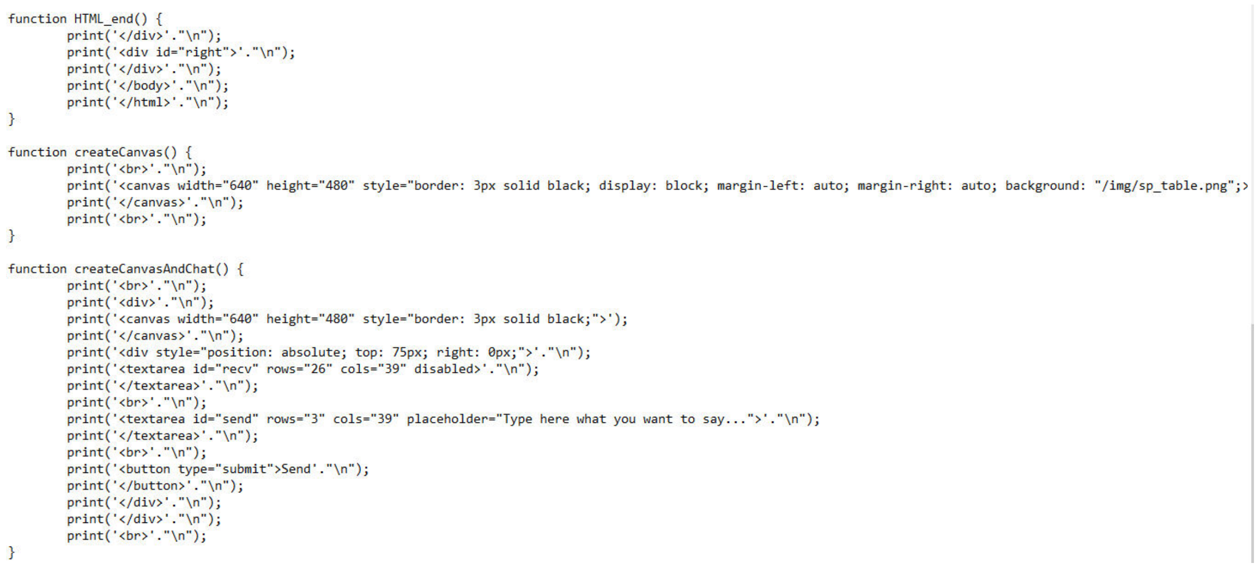
The website also offers a Tutorial for those unfamiliar with the rules of Blackjack, this allows players to know how to play before they begin. The tutorial was written by a member of our team and has its own dedicated HTML page. The final page is the multiplayer page, this page contains a download for the multiplayer version, this page also contains a functional chat window.

The chat window on the multiplayer page of the website was originally planned to work alongside a multiplayer JavaScript version of the game, however we decided to take a different approach due to time pressure and various other complications. The alternative to creating the multiplayer game in JavaScript was to make it in python, this was a language the team was more confident with. With the change to python we felt that we could have a functional multiplayer version of the game given the time pressure.

System Overview

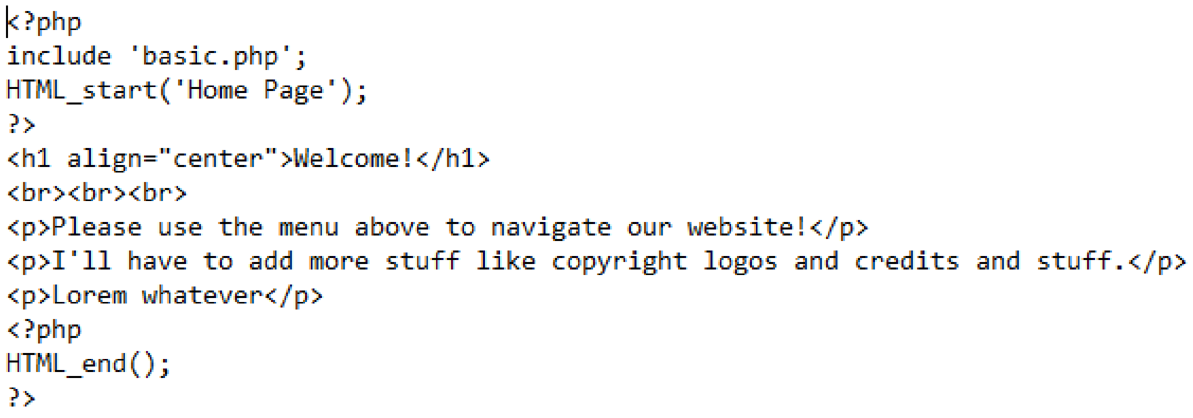
Webpage

The webpage is located on a free web host (0000webhost). The webpage consists of several PHP files which output HTML pages. The use of PHP allowed the webpage to remain modular. The functions used by the various PHP files are located in the PHP file ‘basic.php’. This approach allowed the other PHP files to share functions.



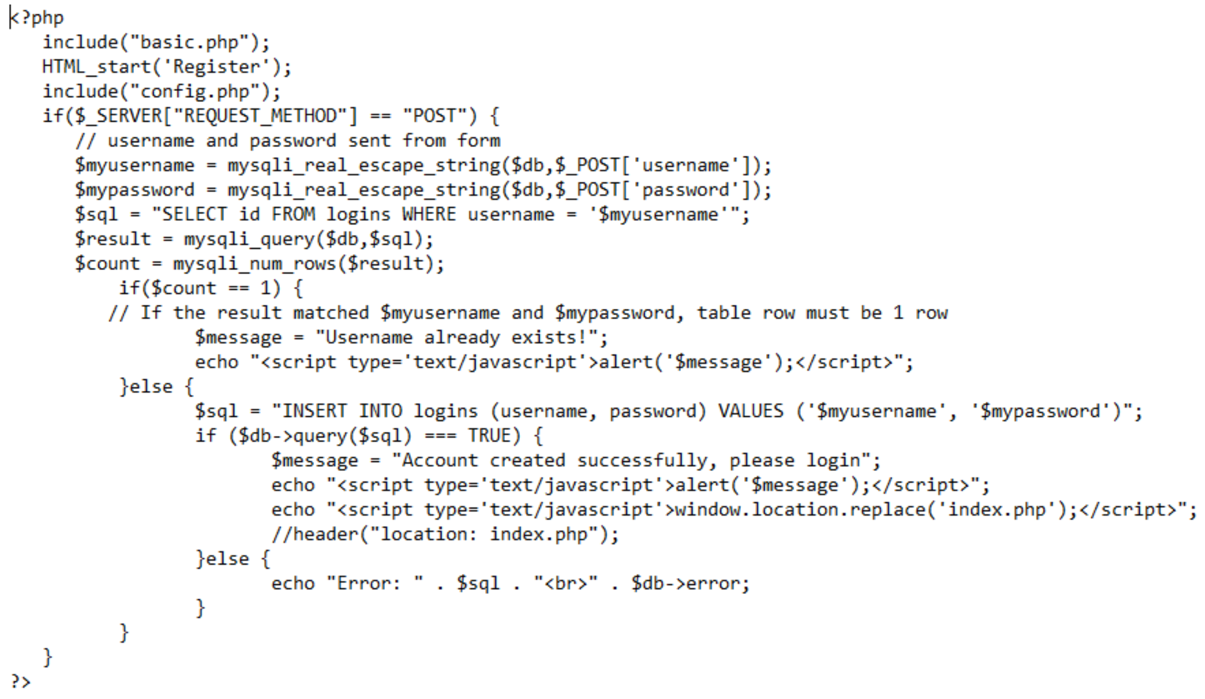
```
function HTML_end() {
        print('</div>'."\n");
        print('<div id="right">'."\n");
        print('</div>'."\n");
        print('</body>'."\n");
        print('</html>'."\n");
}

function createCanvas() {
        print('<br>'."\n");
        print('<canvas width="640" height="480" style="border: 3px solid black; display: block; margin-left: auto; margin-right: auto; background: "/img/sp_table.png";>
        print('</canvas>'."\n");
        print('<br>'."\n");
}

function createCanvasAndChat() {
        print('<br>'."\n");
        print('<div>'."\n");
        print('<canvas width="640" height="480" style="border: 3px solid black;">');
        print('</canvas>'."\n");
        print('<div style="position: absolute; top: 75px; right: 0px;">'."\n");
        print('<textarea id="recv" rows="26" cols="39" disabled>'."\n");
        print('</textarea>'."\n");
        print('<br>'."\n");
        print('<textarea id="send" rows="3" cols="39" placeholder="Type here what you want to say...">'."\n");
        print('</textarea>'."\n");
        print('<br>'."\n");
        print('<button type="submit">Send'."\n");
        print('</button>'."\n");
        print('</div>'."\n");
        print('</div>'."\n");
        print('<br>'."\n");
}
```

The above code shows an example of some of the functions from the ‘basic.php’ file. The other PHP files contain much less code due to the ‘basic.php’ containing most of the functions needed.



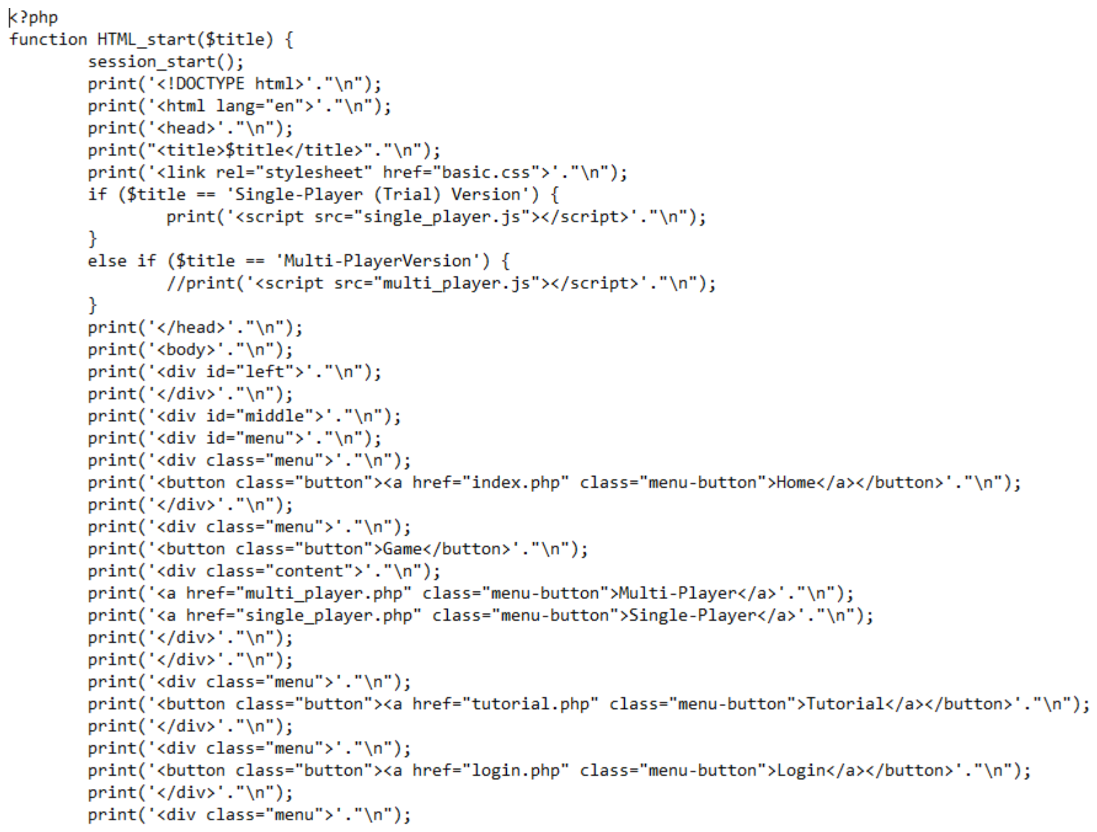
```
<?php
include 'basic.php';
HTML_start('Home Page');
?>
<h1 align="center">Welcome!</h1>
<br><br><br>
<p>Please use the menu above to navigate our website!</p>
<p>I'll have to add more stuff like copyright logos and credits and stuff.</p>
<p>Lorem whatever</p>
<?php
HTML_end();
?>
```

The above is ‘index.php’ in its entirety. The functions from ‘basic.php’ can be seen in use here. Some of the PHP files are much more complicated than this as they interact with the database to provide the login/registration system. To achieve this, a member of our team set up a MySQL database. The PHP files ‘register.php’, ‘login.php’, and ‘logout.php’ directly interact with the database. The register page presents the user with a form, which when filled in, inserts new user information into the database, creating a new account. The login page allows the user to enter a previously specified username and password, followed by a database query.

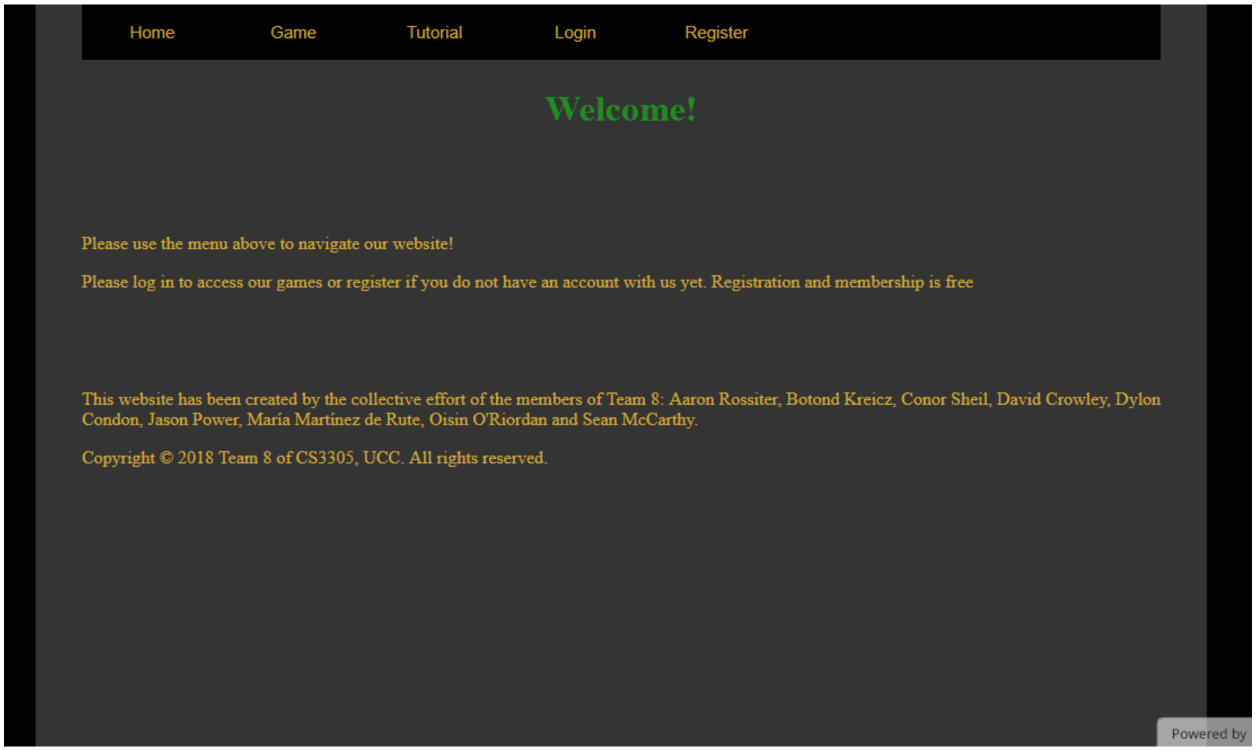


```
<?php
   include("basic.php");
   HTML_start('Register');
   include("config.php");
   if($_SERVER["REQUEST_METHOD"] == "POST") {
      // username and password sent from form
      $myusername = mysqli_real_escape_string($db,$_POST['username']);
      $mypassword = mysqli_real_escape_string($db,$_POST['password']);
      $sql = "SELECT id FROM logins WHERE username = '$myusername'";
      $result = mysqli_query($db,$sql);
      $count = mysqli_num_rows($result);
          if($count == 1) {
         // If the result matched $myusername and $mypassword, table row must be 1 row
                $message = "Username already exists!";
                echo "<script type='text/javascript'>alert('$message');</script>";
          }else {
                $sql = "INSERT INTO logins (username, password) VALUES ('$myusername', '$mypassword')";
                if ($db->query($sql) === TRUE) {
                       $message = "Account created successfully, please login";
                       echo "<script type='text/javascript'>alert('$message');</script>";
                       echo "<script type='text/javascript'>window.location.replace('index.php');</script>";
                       //header("location: index.php");
                }else {
                       echo "Error: " . $sql . "<br>" . $db->error;
                }
          }
   }
?>
```

The website has a consistent style due to the use of a CSS stylesheet. This stylesheet is known as ‘basic.css’. This stylesheet is used by every page on the site as it is linked to the ‘basic.php’ file.



```php
<?php
function HTML_start($title) {
        session_start();
        print('<!DOCTYPE html>'."\n");
        print('<html lang="en">'."\n");
        print('<head>'."\n");
        print("<title>$title</title>"."\n");
        print('<link rel="stylesheet" href="basic.css">'."\n");
        if ($title == 'Single-Player (Trial) Version') {
                print('<script src="single_player.js"></script>'."\n");
        }
        else if ($title == 'Multi-PlayerVersion') {
                //print('<script src="multi_player.js"></script>'."\n");
        }
        print('</head>'."\n");
        print('<body>'."\n");
        print('<div id="left">'."\n");
        print('</div>'."\n");
        print('<div id="middle">'."\n");
        print('<div id="menu">'."\n");
        print('<div class="menu">'."\n");
        print('<button class="button"><a href="index.php" class="menu-button">Home</a></button>'."\n");
        print('</div>'."\n");
        print('<div class="menu">'."\n");
        print('<button class="button">Game</button>'."\n");
        print('<div class="content">'."\n");
        print('<a href="multi_player.php" class="menu-button">Multi-Player</a>'."\n");
        print('<a href="single_player.php" class="menu-button">Single-Player</a>'."\n");
        print('</div>'."\n");
        print('</div>'."\n");
        print('<div class="menu">'."\n");
        print('<button class="button"><a href="tutorial.php" class="menu-button">Tutorial</a></button>'."\n");
        print('</div>'."\n");
        print('<div class="menu">'."\n");
        print('<button class="button"><a href="login.php" class="menu-button">Login</a></button>'."\n");
        print('</div>'."\n");
        print('<div class="menu">'."\n");
```



Welcome!

Please use the menu above to navigate our website!

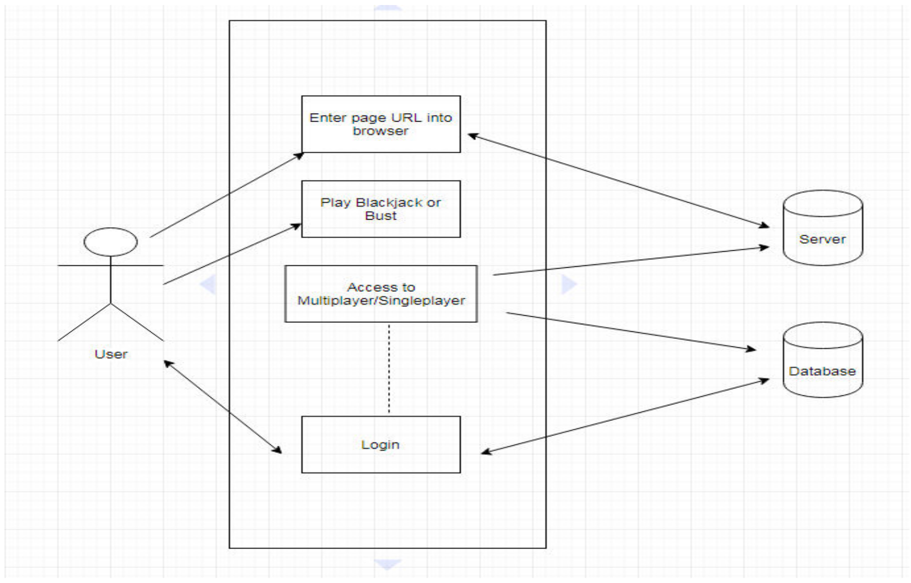
Please log in to access our games or register if you do not have an account with us yet. Registration and membership is free

This website has been created by the collective effort of the members of Team 8: Aaron Rossiter, Botond Kreicz, Conor Sheil, David Crowley, Dylon Condon, Jason Power, María Martínez de Rute, Oisin O'Riordan and Sean McCarthy.

Copyright © 2018 Team 8 of CS3305, UCC. All rights reserved.

The above is a picture of the homepage (index.php) of the site.

Due to the website containing both versions of the game, it is impossible to access our game without a browser. The user must use the website if he/she wishes to play the game. The user does this by following hyperlinks on the website, creating an account and selecting which version of the game to play.

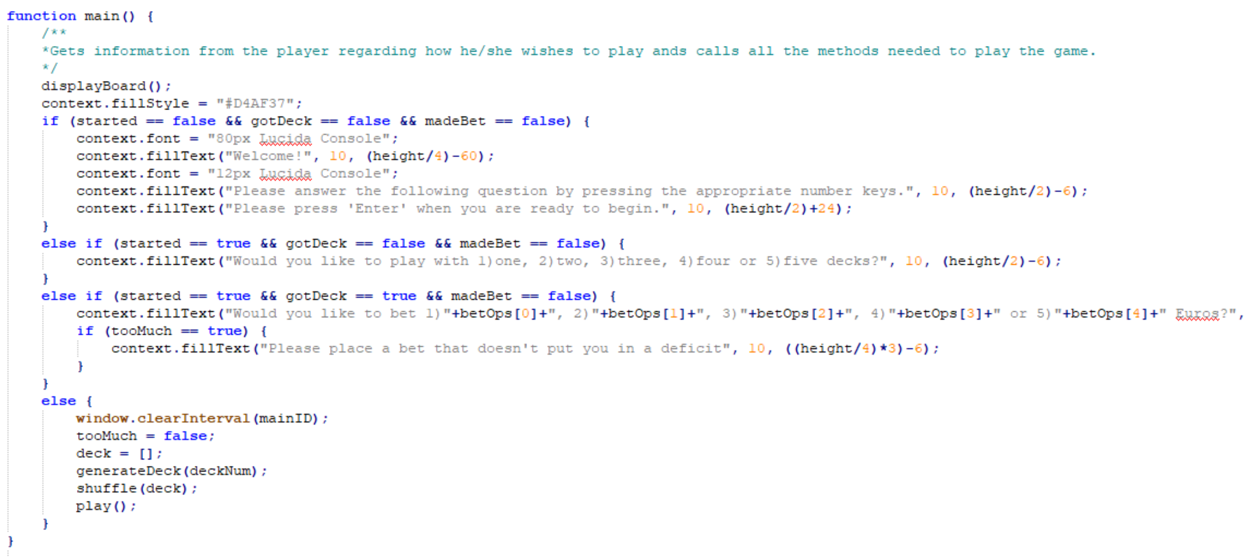

Enter page URL into browser
Play Blackjack or Bust
Access to Multiplayer/Singleplayer
Login
User
Server
Database

JavaScript

The JavaScript code interacts directly with the canvas on the Singleplayer page (‘singpleplayer.php’). At the beginning of the project the team decided that the game should be made in JavaScript. One milestone we hoped to reach was to have a working Singleplayer version of the game. The game is available to play in the user’s browser and features standard Blackjack functionality. The user plays against the dealer.

All logic and functions for the game are contained in a single file (‘singleplayer.js’). The game is played using a combination of both mouse-clicks and pressing of the number-keys. The artwork for the game was designed by members of the team. The game was originally text-based and so the artwork was implemented to provide a more pleasant user experience.

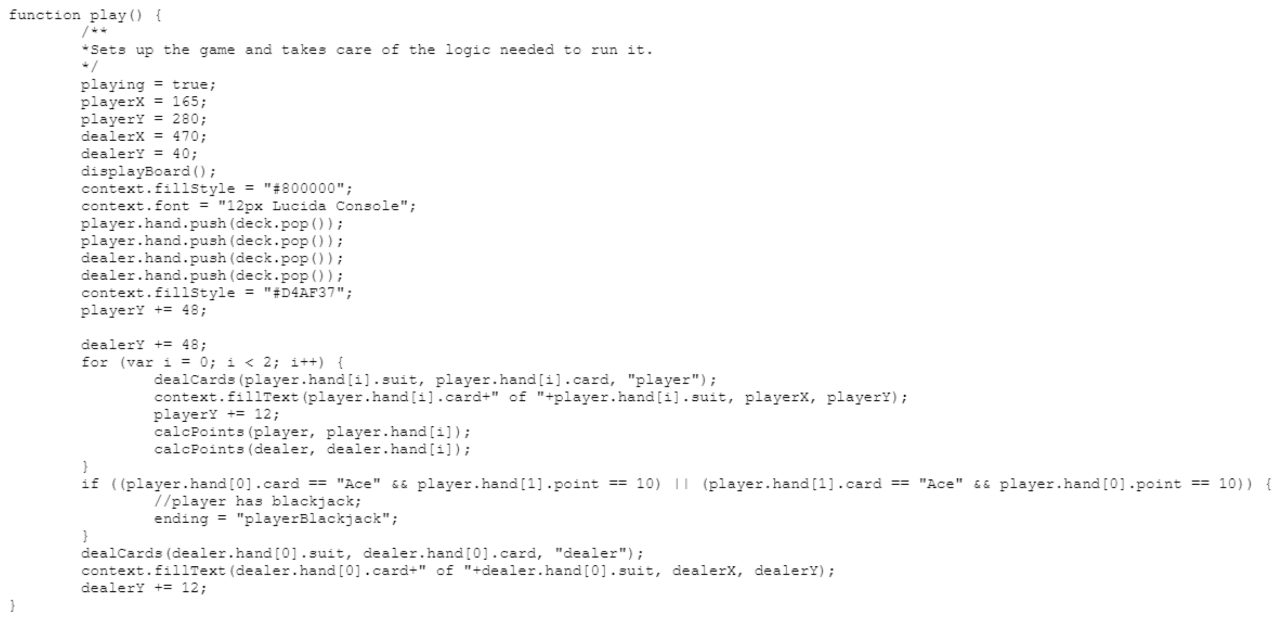
The main() function in ‘singleplayer.php’ is responsible for presenting the various play options to the user. The function outputs text to the canvas, suggesting which buttons the user should press based on their preference for number of decks used and the price of bet. The user’s input is recorded using event handlers for both key presses and clicks. Once the user has completed these actions the game begins and the player uses the mouse to perform further actions.



```
function main() {
    /**
    *Gets information from the player regarding how he/she wishes to play ands calls all the methods needed to play the game.
    */
    displayBoard();
    context.fillStyle = "#D4AF37";
    if (started == false && gotDeck == false && madeBet == false) {
        context.font = "80px Lucida Console";
        context.fillText("Welcome!", 10, (height/4)-60);
        context.font = "12px Lucida Console";
        context.fillText("Please answer the following question by pressing the appropriate number keys.", 10, (height/2)-6);
        context.fillText("Please press 'Enter' when you are ready to begin.", 10, (height/2)+24);
    }
    else if (started == true && gotDeck == false && madeBet == false) {
        context.fillText("Would you like to play with 1)one, 2)two, 3)three, 4)four or 5)five decks?", 10, (height/2)-6);
    }
    else if (started == true && gotDeck == true && madeBet == false) {
        context.fillText("Would you like to bet 1)"+betOps[0]+", 2)"+betOps[1]+", 3)"+betOps[2]+", 4)"+betOps[3]+" or 5)"+betOps[4]+" Euros?",
        if (tooMuch == true) {
            context.fillText("Please place a bet that doesn't put you in a deficit", 10, ((height/4)*3)-6);
        }
    }
    else {
        window.clearInterval(mainID);
        tooMuch = false;
        deck = [];
        generateDeck(deckNum);
        shuffle(deck);
        play();
    }
}
```

The above image shows the main() function.

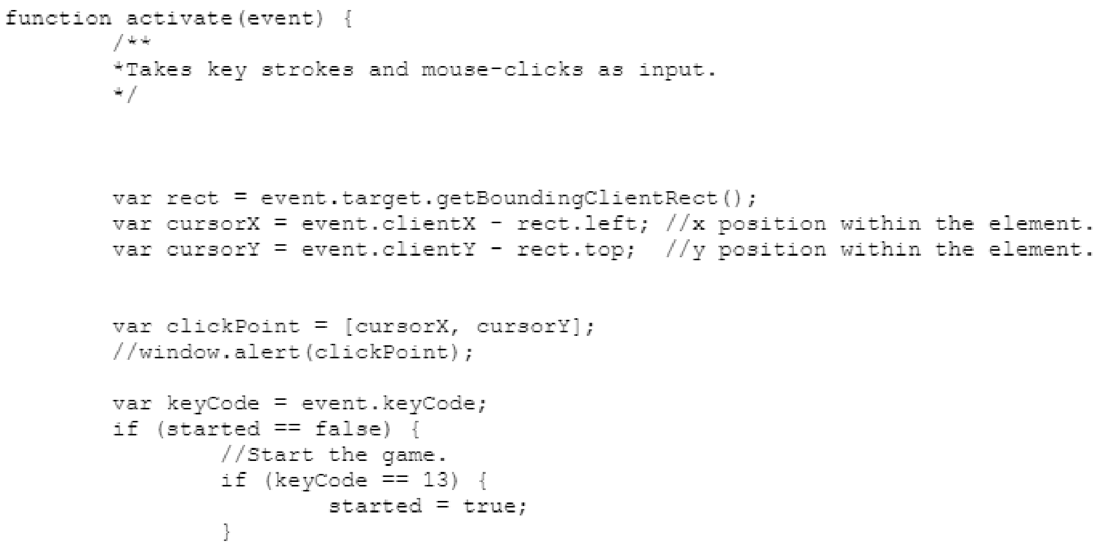
The file contains many helper functions which perform various different actions throughout the game. Such functions include shuffle() and generateDeck(). These functions can be found in larger functions such as play(). The play() function is executed after the user has finished selecting the options for playing the game. The function adds cards to the hands of the player and dealer, the hands are represented by arrays.



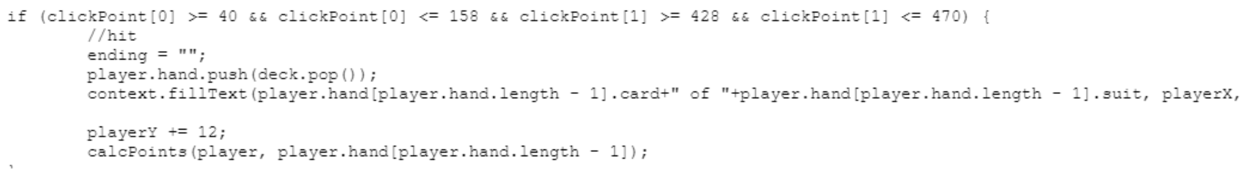
```
function play() {
        /**
        *Sets up the game and takes care of the logic needed to run it.
        */
        playing = true;
        playerX = 165;
        playerY = 280;
        dealerX = 470;
        dealerY = 40;
        displayBoard();
        context.fillStyle = "#800000";
        context.font = "12px Lucida Console";
        player.hand.push(deck.pop());
        player.hand.push(deck.pop());
        dealer.hand.push(deck.pop());
        dealer.hand.push(deck.pop());
        context.fillStyle = "#D4AF37";
        playerY += 48;

        dealerY += 48;
        for (var i = 0; i < 2; i++) {
                dealCards(player.hand[i].suit, player.hand[i].card, "player");
                context.fillText(player.hand[i].card+" of "+player.hand[i].suit, playerX, playerY);
                playerY += 12;
                calcPoints(player, player.hand[i]);
                calcPoints(dealer, dealer.hand[i]);
        }
        if ((player.hand[0].card == "Ace" && player.hand[1].point == 10) || (player.hand[1].card == "Ace" && player.hand[0].point == 10)) {
                //player has blackjack;
                ending = "playerBlackjack";
        }
        dealCards(dealer.hand[0].suit, dealer.hand[0].card, "dealer");
        context.fillText(dealer.hand[0].card+" of "+dealer.hand[0].suit, dealerX, dealerY);
        dealerY += 12;
}
```

The above image shows the play() function.

The user interaction is all processed through the activate(event) function. The activate() function takes an event as an input, whether it be a mouse-click or a keystroke, and processes the corresponding action based on the input. This function was heavily altered to cater for the detection of mouse-clicks relative to the canvas. This was achieved by finding where the user click on the canvas and comparing it to the areas in which an action occurs.



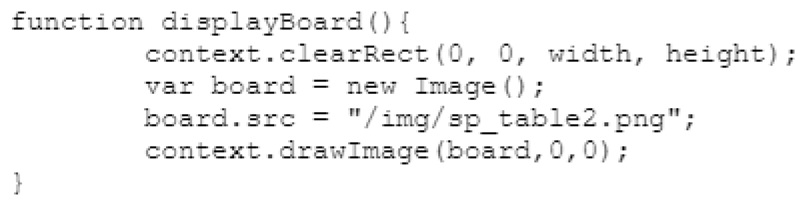
```
function activate(event) {
        /**
        *Takes key strokes and mouse-clicks as input.
        */



        var rect = event.target.getBoundingClientRect();
        var cursorX = event.clientX - rect.left; //x position within the element.
        var cursorY = event.clientY - rect.top;  //y position within the element.


        var clickPoint = [cursorX, cursorY];
        //window.alert(clickPoint);

        var keyCode = event.keyCode;
        if (started == false) {
                //Start the game.
                if (keyCode == 13) {
                        started = true;
                }
```



```
if (clickPoint[0] >= 40 && clickPoint[0] <= 158 && clickPoint[1] >= 428 && clickPoint[1] <= 470) {
        //hit
        ending = "";
        player.hand.push(deck.pop());
        context.fillText(player.hand[player.hand.length - 1].card+" of "+player.hand[player.hand.length - 1].suit, playerX,

        playerY += 12;
        calcPoints(player, player.hand[player.hand.length - 1]);
```

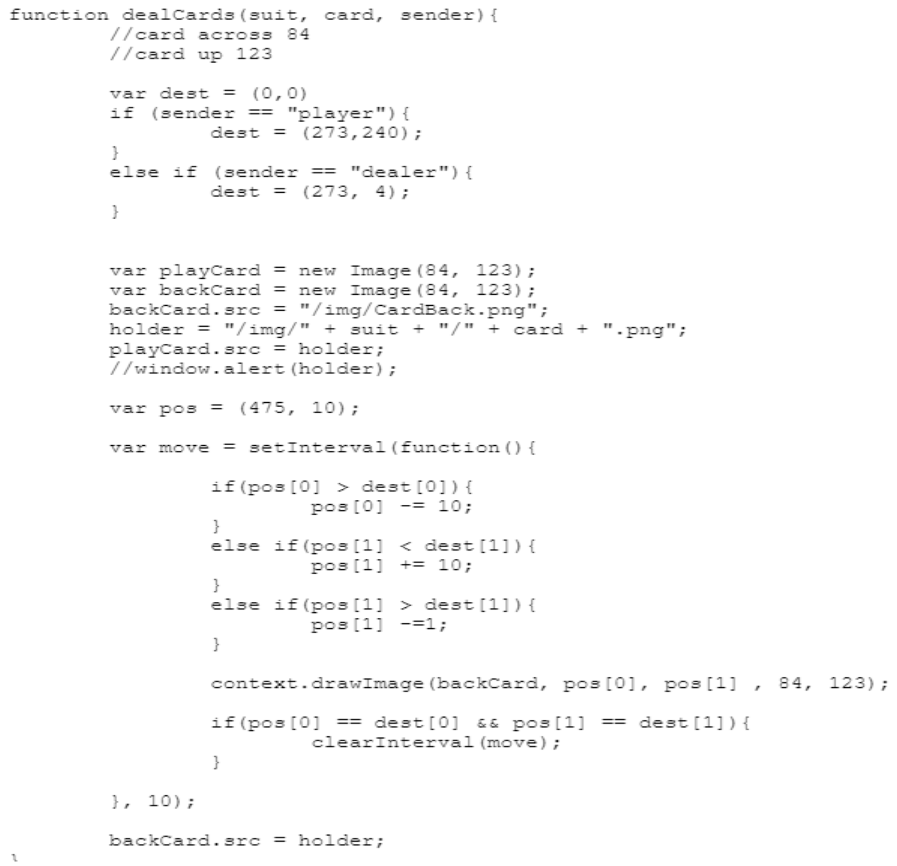
The above images show how the locations of mouse-clicks are recorded in relation to the canvas. How this information is used is shown in the second image (the example for ‘hit’)

The functions to display and manipulate images are located near the bottom of the file. These functions are used to display the game board and the cards used for playing. There was exceptional difficulty in getting the cards to appear as though they were moving. The approach taken involves the card being redrawn using the setInterval() function. The images used were designed by members of the team. The cards are retrieved by constructing the file path based off the hand dealt to the player in the play() function.



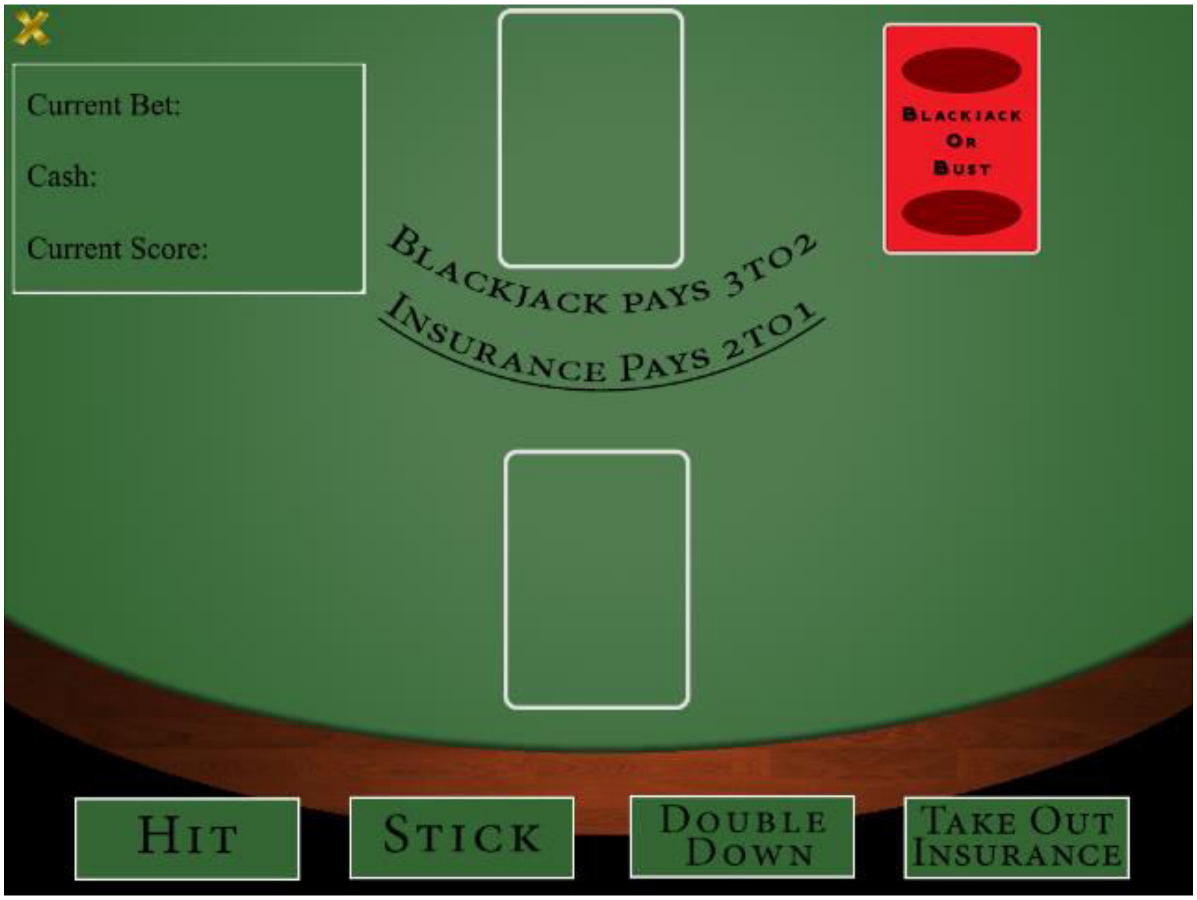
```
function displayBoard(){
        context.clearRect(0, 0, width, height);
        var board = new Image();
        board.src = "/img/sp_table2.png";
        context.drawImage(board,0,0);
}
```

The above image shows the function displayBoard() which is responsible for displaying the background of the game. The following image shows the dealCards() function., which handles the images for the cards.



```
function dealCards(suit, card, sender){
        //card across 84
        //card up 123

        var dest = (0,0)
        if (sender == "player"){
                dest = (273,240);
        }
        else if (sender == "dealer"){
                dest = (273, 4);
        }


        var playCard = new Image(84, 123);
        var backCard = new Image(84, 123);
        backCard.src = "/img/CardBack.png";
        holder = "/img/" + suit + "/" + card + ".png";
        playCard.src = holder;
        //window.alert(holder);

        var pos = (475, 10);

        var move = setInterval(function(){

                if(pos[0] > dest[0]){
                        pos[0] -= 10;
                }
                else if(pos[1] < dest[1]){
                        pos[1] += 10;
                }
                else if(pos[1] > dest[1]){
                        pos[1] -=1;
                }

                context.drawImage(backCard, pos[0], pos[1] , 84, 123);

                if(pos[0] == dest[0] && pos[1] == dest[1]){
                        clearInterval(move);
                }

        }, 10);

        backCard.src = holder;
}
```

The background image for the game has had many iterations, from a simple text based game to the image in use have now, which allows for the user to interact directly with the canvas. The current image also has a designated area for which the user's statistics are displayed.



The above image shows the game board used in the current state of the game. The functions in the file display the users statistics in the corresponding location. The buttons located at the bottom of the image are fully functional and allow the user to interact with the game.

Operation Guide for the Python code

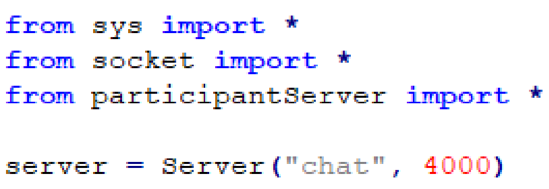
María Martínez de Rute

117106133

The Python code consists of six files: chatServer.py, gameServer.py, hostServer.py, client.py, participantServer.py and gameServer.py.

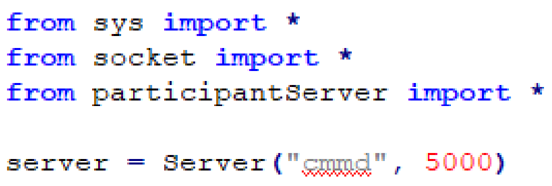
When **chatServer.py** and the **gameServer.py** files are executed, they call participantServer.py, open a connection and wait for data.

chat Server.py



```
from sys import *
from socket import *
from participantServer import *

server = Server("chat", 4000)
```

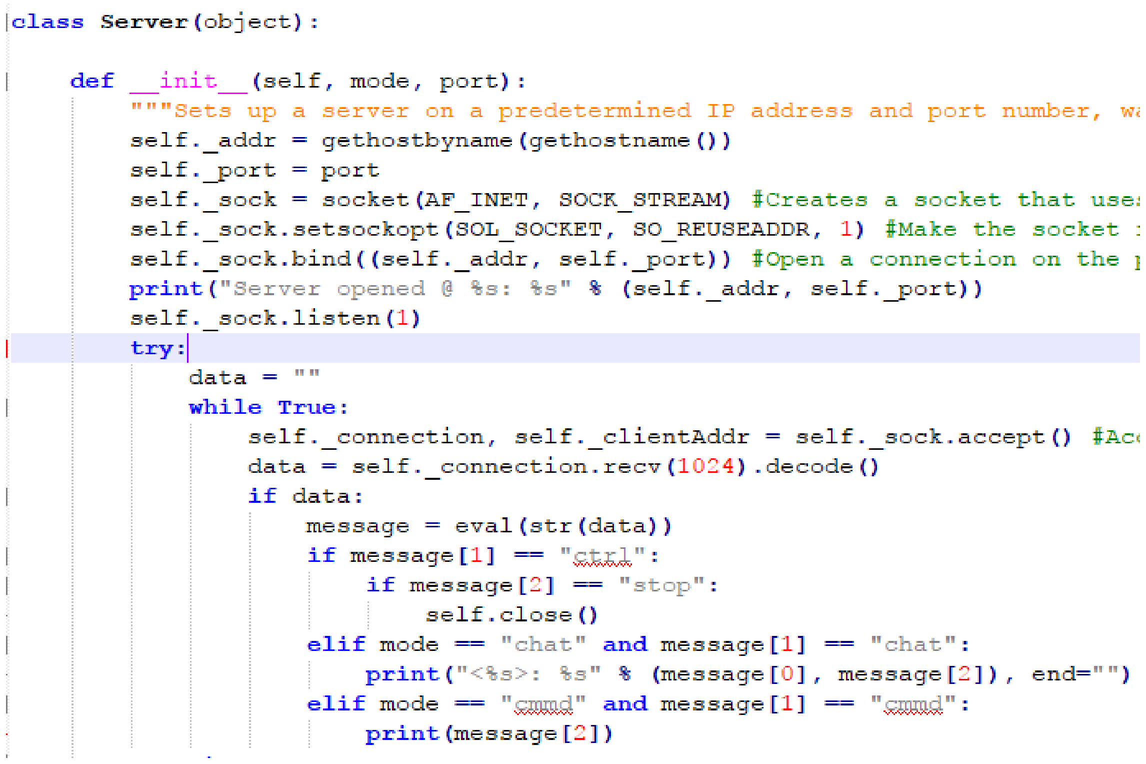
gameServer.py



```
from sys import *
from socket import *
from participantServer import *

server = Server("cmmd", 5000)
```

ChatServer.py uses mode="chat" , while gameServer.py uses mode="cmmd". When data is received it is printed out in different format. The chatServer.py receives and displays messages whereas the gameServer.py receives and displays game commands.

participantServer.py



```
class Server(object):

    def __init__(self, mode, port):
        """Sets up a server on a predetermined IP address and port number, wa
        self._addr = gethostbyname(gethostname())
        self._port = port
        self._sock = socket(AF_INET, SOCK_STREAM) #Creates a socket that use
        self._sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1) #Make the socket :
        self._sock.bind((self._addr, self._port)) #Open a connection on the p
        print("Server opened @ %s: %s" % (self._addr, self._port))
        self._sock.listen(1)
        try:
            data = ""
            while True:
                self._connection, self._clientAddr = self._sock.accept() #Ac
                data = self._connection.recv(1024).decode()
                if data:
                    message = eval(str(data))
                    if message[1] == "ctrl":
                        if message[2] == "stop":
                            self.close()
                    elif mode == "chat" and message[1] == "chat":
                        print("<%s>: %s" % (message[0], message[2]), end="")
                    elif mode == "cmmd" and message[1] == "cmmd":
                        print(message[2])
```

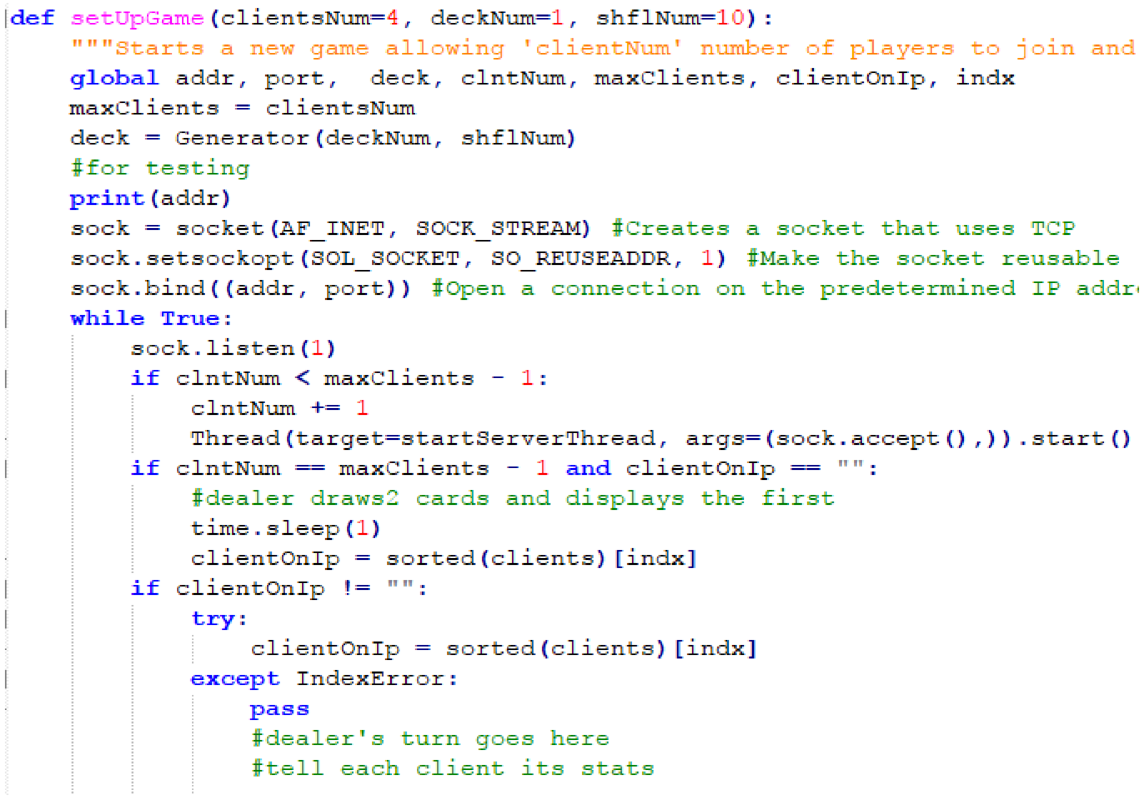
In the previous code inside the participantServer.py file, it can be seen how is implemented the Server class, when is called it has to receive two arguments, the mode and the port, as we can see in the chatServer.py and the gameServer.py, it sets up a server (with the socket module) and waits for incoming connection. When a message is received (if there is data), the message is displayed. It can be called by the chatServer with the chat mode, by the gameServer with the cmmd mode or by the hostServer with a message of ctrl to stop the game.

hostServer.py

When the hostServer.py is executed the function **setUpGame** is called. This function starts a new game allowing a defined number of players to play with a defined number of desks being shuffled a number defined of times.

The host can decide the three numbers when the code is executed. If the host does not choose a number they are by default: 4 players and one desk shuffled 10 times.

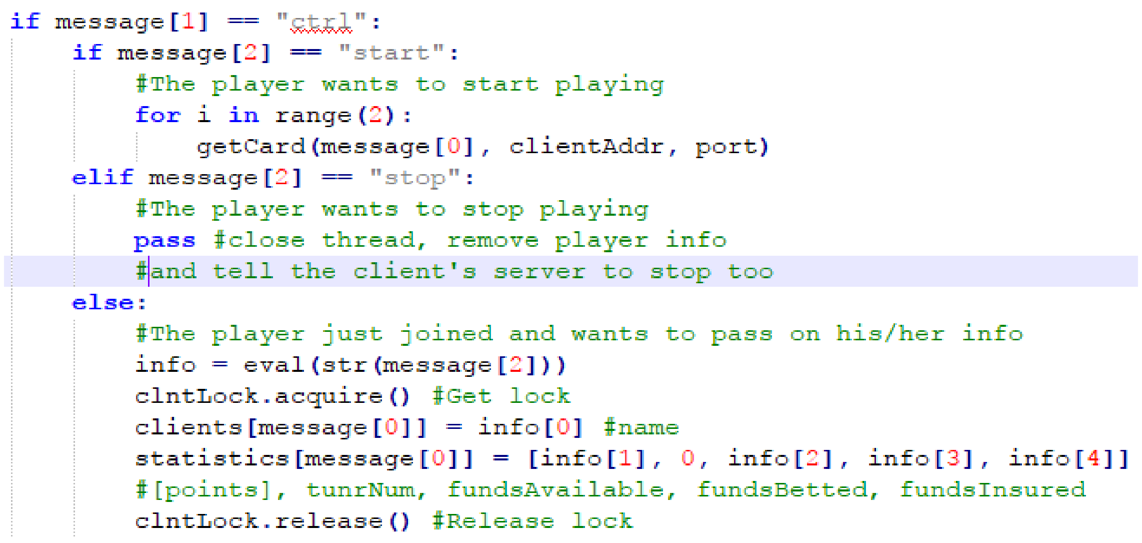
A new connection is open and it waits for connections from other players. This function limits the number of clients. The maximum number of players is 4. If a fifth player tries to join the game, he is rejected. If a new player wants to join and there are less than 4 players he can join and the function startServerThread is called.



```
def setUpGame(clientsNum=4, deckNum=1, shflNum=10):
    """Starts a new game allowing 'clientNum' number of players to join and
    global addr, port,  deck, clntNum, maxClients, clientOnIp, indx
    maxClients = clientsNum
    deck = Generator(deckNum, shflNum)
    #for testing
    print(addr)
    sock = socket(AF_INET, SOCK_STREAM) #Creates a socket that uses TCP
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1) #Make the socket reusable
    sock.bind((addr, port)) #Open a connection on the predetermined IP addr
    while True:
        sock.listen(1)
        if clntNum < maxClients - 1:
            clntNum += 1
            Thread(target=startServerThread, args=(sock.accept(),)).start()
        if clntNum == maxClients - 1 and clientOnIp == "":
            #dealer draws2 cards and displays the first
            time.sleep(1)
            clientOnIp = sorted(clients)[indx]
        if clientOnIp != "":
            try:
                clientOnIp = sorted(clients)[indx]
            except IndexError:
                pass
                #dealer's turn goes here
                #tell each client its stats
```

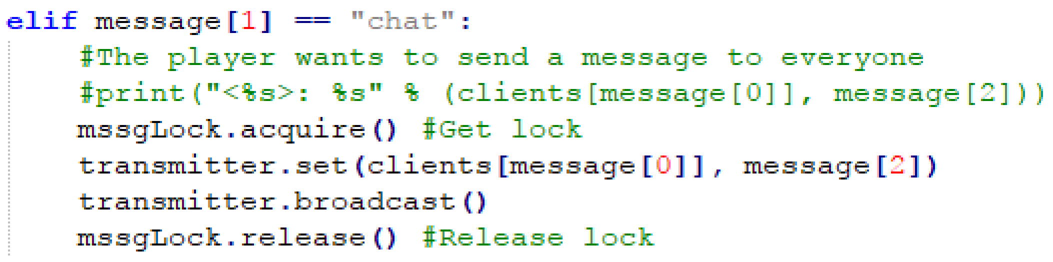
When a new player presses the button “Connect” in the GUI window the **startServerThread** is called. A server thread is created and connected to a player that receives and sends messages to him. It waits to receive messages from the player.

If a “ctrl” message is received there are three possibilities. The first one is when the second message is “start”. In that case this message is sent when the player press the button “Connect” and after that two cards are displayed. In the second one, the second message is “stop”, which means that the player does not want to continue the game and this message is sent when the player presses the button “disconnect”. The third one occurs when a player just join the game and his information is sent.



```
if message[1] == "ctrl":
    if message[2] == "start":
        #The player wants to start playing
        for i in range(2):
            getCard(message[0], clientAddr, port)
    elif message[2] == "stop":
        #The player wants to stop playing
        pass #close thread, remove player info
        #and tell the client's server to stop too
    else:
        #The player just joined and wants to pass on his/her info
        info = eval(str(message[2]))
        clntLock.acquire() #Get lock
        clients[message[0]] = info[0] #name
        statistics[message[0]] = [info[1], 0, info[2], info[3], info[4]]
        #[points], tunrNum, fundsAvailable, fundsBetted, fundsInsured
        clntLock.release() #Release lock
```

If a “chat” message is received this message is displayed to everyone’s screen (with the broadcast () function).

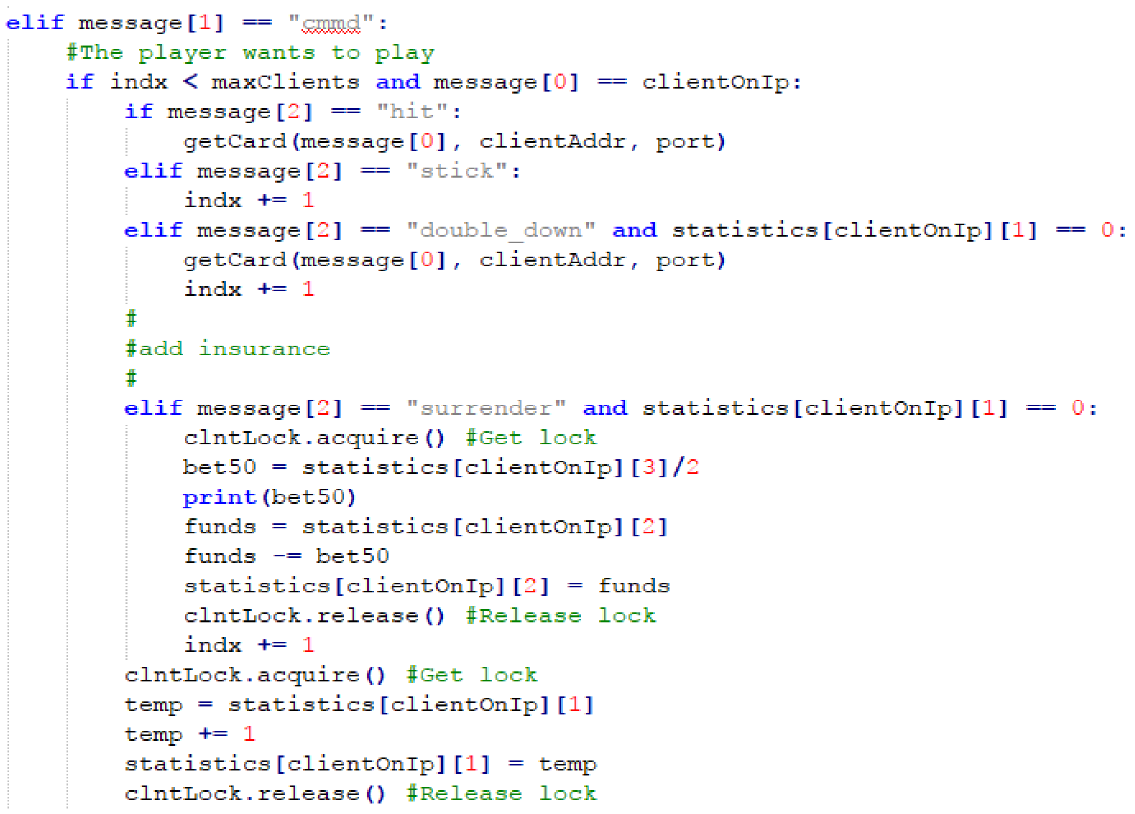


```
elif message[1] == "chat":
    #The player wants to send a message to everyone
    #print("<%s>: %s" % (clients[message[0]], message[2]))
    mssgLock.acquire() #Get lock
    transmitter.set(clients[message[0]], message[2])
    transmitter.broadcast()
    mssgLock.release() #Release lock
```

If a “cmmd” message is received, the player wants to play. If it is the turn of a player, he can choose between different options.

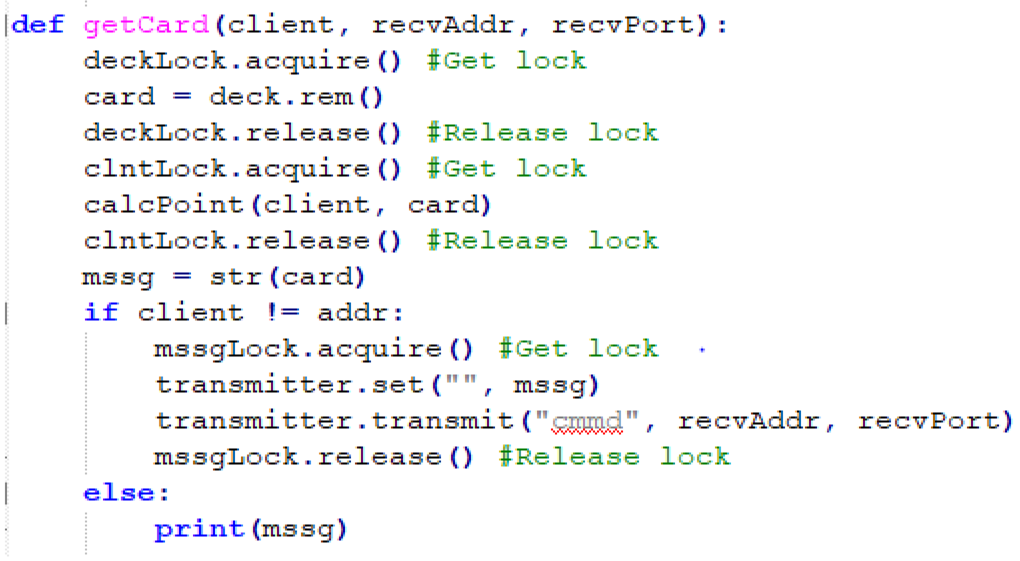
If the player presses “Hit” another card is displayed and his points are actualised. If the player presses “Stick” he loses his turn and it is the turn of the next player.

Only if the player is still having two cards, he can press “double down”, “insurance” or “surrender”. If he presses “double down”, the bet is doubled, a new card is displayed and it is the turn of the next player. If he presses “insurance”, he gets a side bet. If he presses “surrender” (only if the dealer has an ace), he receives half of his initial bet.



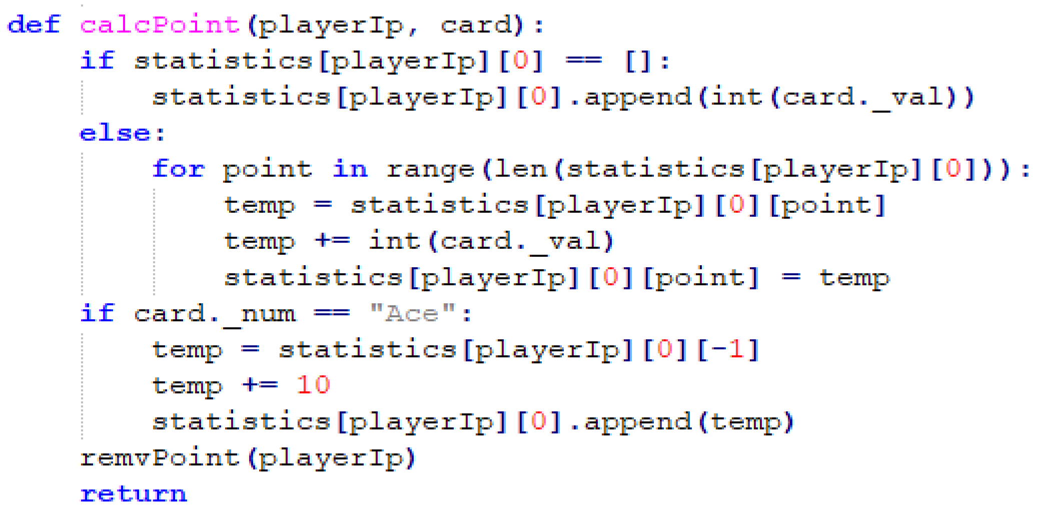
```
elif message[1] == "cmmd":
    #The player wants to play
    if indx < maxClients and message[0] == clientOnIp:
        if message[2] == "hit":
            getCard(message[0], clientAddr, port)
        elif message[2] == "stick":
            indx += 1
        elif message[2] == "double_down" and statistics[clientOnIp][1] == 0:
            getCard(message[0], clientAddr, port)
            indx += 1
        #
        #add insurance
        #
        elif message[2] == "surrender" and statistics[clientOnIp][1] == 0:
            clntLock.acquire() #Get lock
            bet50 = statistics[clientOnIp][3]/2
            print(bet50)
            funds = statistics[clientOnIp][2]
            funds -= bet50
            statistics[clientOnIp][2] = funds
            clntLock.release() #Release lock
            indx += 1
        clntLock.acquire() #Get lock
        temp = statistics[clientOnIp][1]
        temp += 1
        statistics[clientOnIp][1] = temp
        clntLock.release() #Release lock
```

The function **getCard()** removes a card from the desk and gives the card to the player. The message is transmitted and card is displayed in the screen of the player. The function calls calcPoint() that calculates the new points of the players after the display of the new card. As it can be seen when a function is going to be called, it has to get the lock and after the execution release the lock. That is how the threads work.



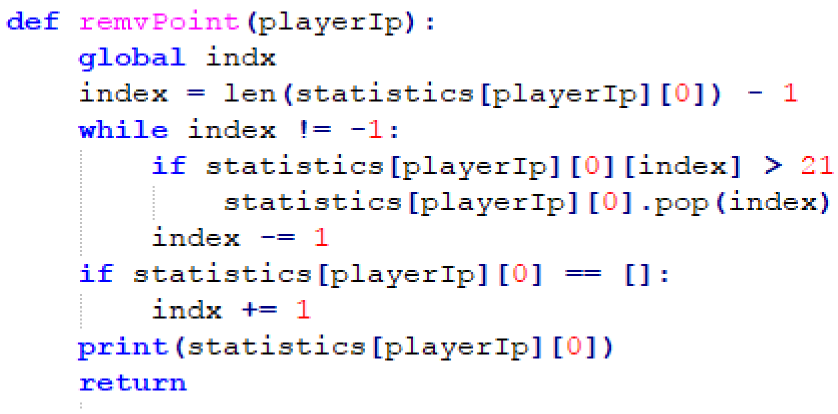
```
def getCard(client, recvAddr, recvPort):
    deckLock.acquire() #Get lock
    card = deck.rem()
    deckLock.release() #Release lock
    clntLock.acquire() #Get lock
    calcPoint(client, card)
    clntLock.release() #Release lock
    mssg = str(card)
    if client != addr:
        mssgLock.acquire() #Get lock
        transmitter.set("", mssg)
        transmitter.transmit("cmmd", recvAddr, recvPort)
        mssgLock.release() #Release lock
    else:
        print(mssg)
```

The function **calcPoint()** calculates the points that a player has. After the calculation the function remvPoint() is called to see if the points of the player are over 21 or not.



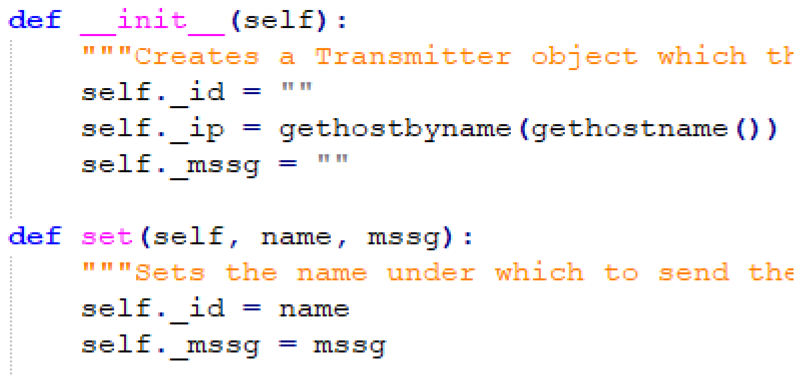
```
def calcPoint(playerIp, card):
    if statistics[playerIp][0] == []:
        statistics[playerIp][0].append(int(card._val))
    else:
        for point in range(len(statistics[playerIp][0])):
            temp = statistics[playerIp][0][point]
            temp += int(card._val)
            statistics[playerIp][0][point] = temp
    if card._num == "Ace":
        temp = statistics[playerIp][0][-1]
        temp += 10
        statistics[playerIp][0].append(temp)
    remvPoint(playerIp)
    return
```

The function **remvPoint()** looks the points of a player. If a player is over 21 in points he loses and his points are removed. The points are printed out.



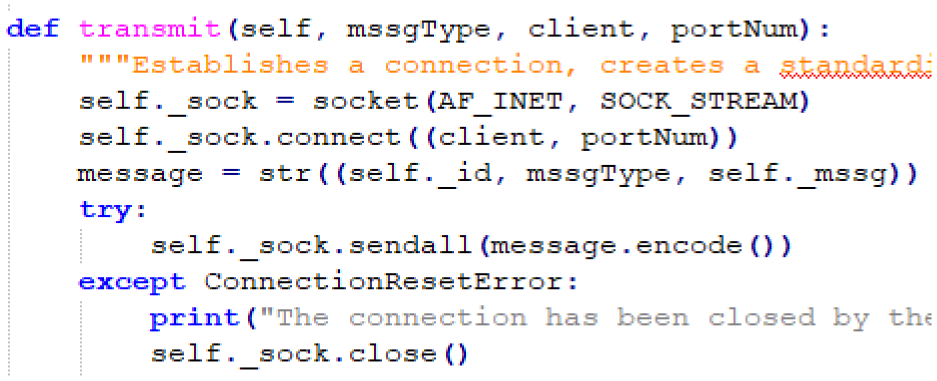
```
def remvPoint(playerIp):
    global indx
    index = len(statistics[playerIp][0]) - 1
    while index != -1:
        if statistics[playerIp][0][index] > 21:
            statistics[playerIp][0].pop(index)
        index -= 1
    if statistics[playerIp][0] == []:
        indx += 1
    print(statistics[playerIp][0])
    return
```

The class **Transmitter(object)** creates a temporary connection, displays a message and closes the connection. First of all, the transmitter object is created to allow the host to communicate with the players. With the function **set()** the name of the player, with which the messages are sent, is set.



```
def __init__(self):
    """Creates a Transmitter object which tl
    self._id = ""
    self._ip = gethostbyname(gethostname())
    self._mssg = ""

def set(self, name, mssg):
    """Sets the name under which to send the
    self._id = name
    self._mssg = mssg
```

The function **transmit()** establishes a socket connection, creates a standardised message and sends it over the connection.



```
def transmit(self, mssgType, client, portNum):
    """Establishes a connection, creates a standard:
    self._sock = socket(AF_INET, SOCK_STREAM)
    self._sock.connect((client, portNum))
    message = str((self._id, mssgType, self._mssg))
    try:
        self._sock.sendall(message.encode())
    except ConnectionResetError:
        print("The connection has been closed by the
        self._sock.close()
```

The **broadcast()** function transmits a message to every player calling the function transmit(). The message is displayed in the screen of each player.

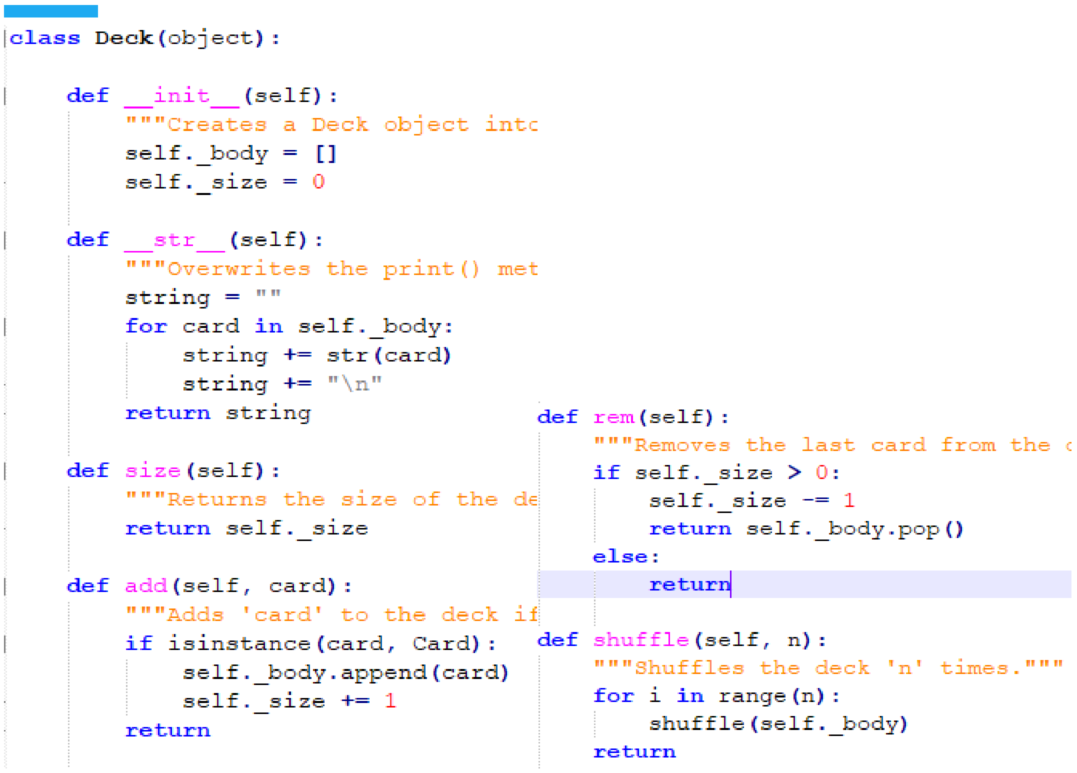
```python
def broadcast(self):
    """Sends a message to every player."""
    for client in clients:
        try:
            self.transmit("chat", client, chat)
        except:
            pass
        finally:
            self._sock.close()
    return
```

deckSetUp.py

First we create the cards and the desk. Each **card** is created with a number, suit and value for the game (attributes of the cards). Then a method to display on the screen the characteristics of the cards is created.

```python
class Card(object):

    def __init__(self, cardNum, cardKin, cardVal):
        """Creates a Card object with a number, suit and value."""
        self._num = cardNum
        self._kin = cardKin
        self._val = cardVal

    def __str__(self):
        """Overwrites the print() method to neatly print out the variables
        if self._num == "Ace":
            return "Ace of %s, worth 1 or 11" % (self._kin)
        else:
            return "%s of %s, worth %s" % (self._num, self._kin, self._val)
```

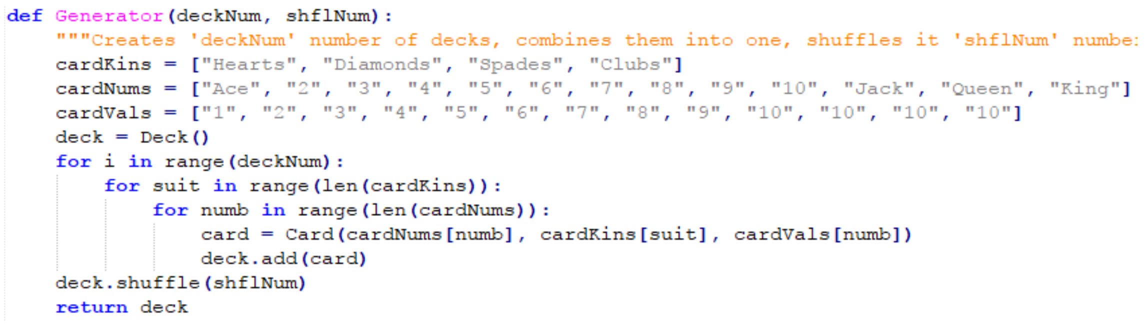
The **desk** object is create with some methods to add cards, remove cards or shuffle the desk. The method size() returns the size of the desk. The method add() adds a card in the corresponding desk. The method rem() removes a card from the desk. The method shuffle() shuffles the desk the number of times that it receives.



```
class Deck(object):

    def __init__(self):
        """Creates a Deck object intc
        self._body = []
        self._size = 0

    def __str__(self):
        """Overwrites the print() met
        string = ""
        for card in self._body:
            string += str(card)
            string += "\n"
        return string

    def size(self):
        """Returns the size of the de
        return self._size

    def add(self, card):
        """Adds 'card' to the deck if
        if isinstance(card, Card):
            self._body.append(card)
            self._size += 1
        return
```

```
def rem(self):
    """Removes the last card from the c
    if self._size > 0:
        self._size -= 1
        return self._body.pop()
    else:
        return

def shuffle(self, n):
    """Shuffles the deck 'n' times."""
    for i in range(n):
        shuffle(self._body)
    return
```

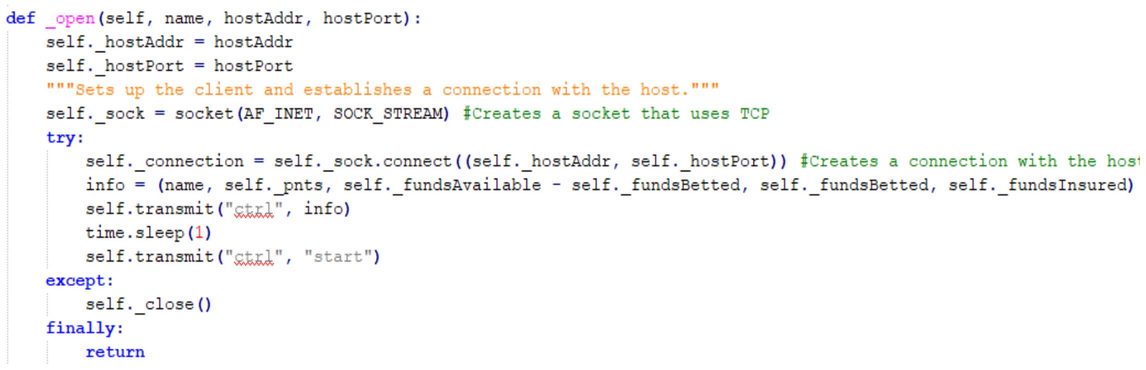
The method **Generator()** is the method that creates a deck, with a number of desks inside that is shuffled a number of times that is established.



```
def Generator(deckNum, shflNum):
    """Creates 'deckNum' number of decks, combines them into one, shuffles it 'shflNum' numbe
    cardKins = ["Hearts", "Diamonds", "Spades", "Clubs"]
    cardNums = ["Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King"]
    cardVals = ["1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "10", "10", "10"]
    deck = Deck()
    for i in range(deckNum):
        for suit in range(len(cardKins)):
            for numb in range(len(cardNums)):
                card = Card(cardNums[numb], cardKins[suit], cardVals[numb])
                deck.add(card)
    deck.shuffle(shflNum)
    return deck
```

client.py

When a player executes client.py a client is created and the GUI window is showed. Once that the GUI is opened, the player has to introduce his name, the Address of the host and the Port of the host. When the player presses the button “Connect” the method **_open()** is called. With this method a client is set up and it establishes a connection with the host. This method sends two “ctrl” messages mentioned before, the message “start”: the player wants to play, and the information of the player.



```
def _open(self, name, hostAddr, hostPort):
    self._hostAddr = hostAddr
    self._hostPort = hostPort
    """Sets up the client and establishes a connection with the host."""
    self._sock = socket(AF_INET, SOCK_STREAM) #Creates a socket that uses TCP
    try:
        self._connection = self._sock.connect((self._hostAddr, self._hostPort)) #Creates a connection with the hos
        info = (name, self._pnts, self._fundsAvailable - self._fundsBetted, self._fundsBetted, self._fundsInsured)
        self.transmit("ctrl", info)
        time.sleep(1)
        self.transmit("ctrl", "start")
    except:
        self._close()
    finally:
        return
```

If a player presses the button “disconnect” the **_close()** function is called and the connection is closed: the player does not want to play more.

The following methods show the different options that the player can choose when he is playing. The method **message()** sends “message” to the host to be broadcasted.

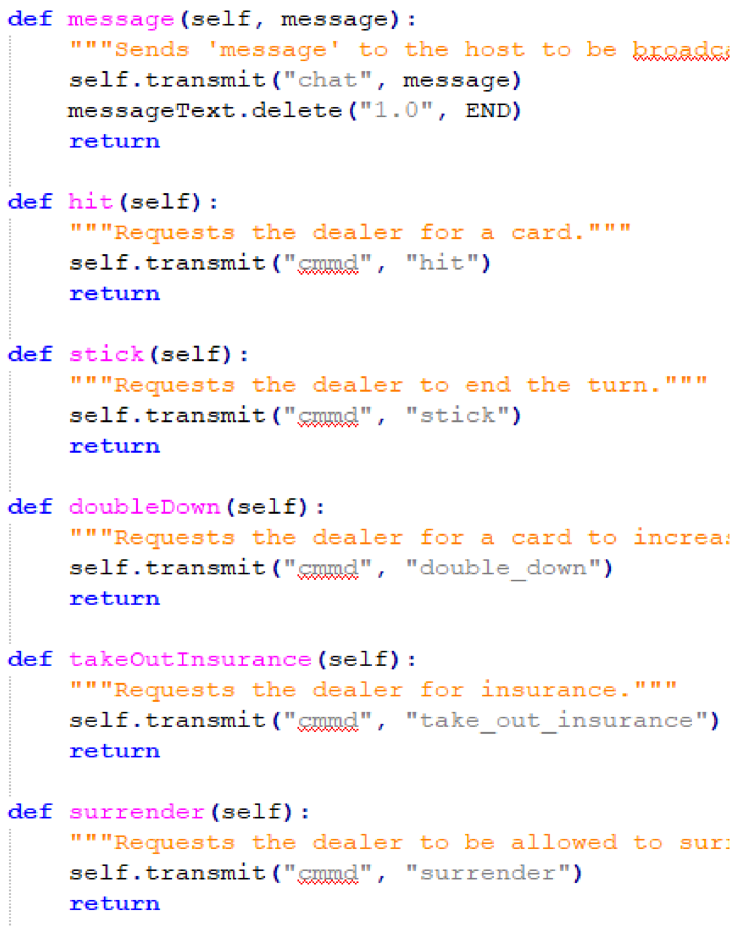
If the player press “**Hit**”, the method **hit()** sends a request to the dealer for another card. A new card will be displayed in the corresponding window.

If the player press “**Stick**”, the method **stick()** requests the dealer to end the turn, then the player loss his turn and is the turn of the next player.

The player can press “**Double Down**”, the method **doubleDown()** sends a request to the dealer (only if he has two cards) to double the bet, receive another card and loss his turn (hit and stick).

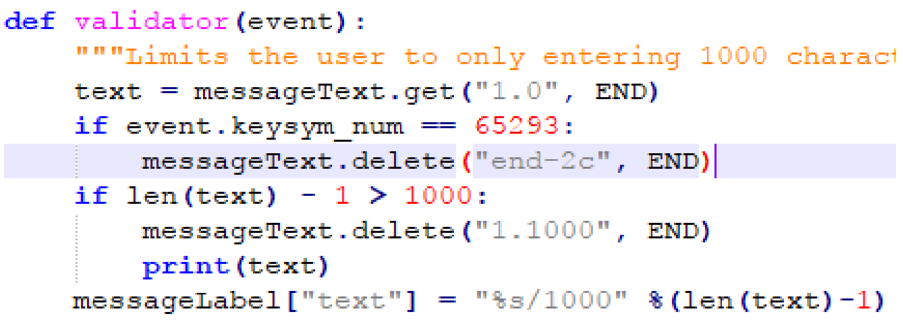
The player can press “**Insurance**”, the method **takeOutInsurance()** sends a request to the dealer for get an insurance.

Finally if the player press “**Surrender**”, the method **surrender()** sends a request to the dealer to receive half of his initial bet.



```
def message(self, message):
    """Sends 'message' to the host to be broadc
    self.transmit("chat", message)
    messageText.delete("1.0", END)
    return

def hit(self):
    """Requests the dealer for a card."""
    self.transmit("cmmd", "hit")
    return

def stick(self):
    """Requests the dealer to end the turn."""
    self.transmit("cmmd", "stick")
    return

def doubleDown(self):
    """Requests the dealer for a card to increa
    self.transmit("cmmd", "double_down")
    return

def takeOutInsurance(self):
    """Requests the dealer for insurance."""
    self.transmit("cmmd", "take_out_insurance")
    return

def surrender(self):
    """Requests the dealer to be allowed to sur
    self.transmit("cmmd", "surrender")
    return
```

The method **validator()** limits the size of the messages that the players can send. The maximum number of characters is 1000 into 'messageText' and updates 'messageLabel' to accurately keep track of the number of characters entered.



```
def validator(event):
    """Limits the user to only entering 1000 charact
    text = messageText.get("1.0", END)
    if event.keysym_num == 65293:
        messageText.delete("end-2c", END)
    if len(text) - 1 > 1000:
        messageText.delete("1.1000", END)
        print(text)
    messageLabel["text"] = "%s/1000" %(len(text)-1)
```

Operation Team

Our team originally consisted of 9 members, randomly assigned to this group. The team is structured as follows:

- Sean McCarthy - **Product Owner**
- Conor Sheil - **Scrum Master**
- Botond Kreicz - **Developer**
- Maria Martinez - **Developer**
- Dylon Condon - **Developer**
- David Crowley - **Developer**
- Oisin O'Riordan - **Developer**
- Aaron rossiter - **Developer**
- Jason power - **Developer** (Exemption)

On the day of our first meeting (in which not all members were present) on 18/01/2018, the group decided that Sean McCarthy would be the Product Owner and Conor Sheil would be scrum master. Both of these positions were volunteered for by these members. Both candidates for Scrum Master and Product Owner accepted the responsibility for these roles. The rest of the members of the team were assigned as Developers.

The team's main method of communication was through Facebook Messenger, and at the two weekly meetings. A Facebook messenger group chat allowed any user to speak to the rest of the group at any time and from any location.

The team used Trello to keep track of the scrum process. Trello's simple yet effective layout allowed the team to remain up to date with any changes or tasks they have been assigned. The Scrum Master was responsible for setting up and maintaining the Trello board.

The team used Google Drive and an application called FileZilla to upload work completed so that other team members may alter or use it. Any files uploaded through FileZilla would automatically be added to the site, this allowed users to remotely add any work completed to the site itself. FileZilla also allowed for the team to add new or updated code and have it's effect show immediately.

Operation Schedule

The first team meeting occurred on 18/01/2018, although not all members of the team were present, the team decided on a weekly schedule in which meetings would occur twice a week. The meetings would take place every week on Monday at 15:05 and Thursday at 15:05. The schedule for the meetings was made this way to ensure that all members of the team would be available, due to the team having lectures on these days. With this approach it was expected that all members would be present with the exception of special cases (e.g. sick, work).

The entire project took place over an 8 week period. This meant that there would be total of 8 sprints in which the team would work. Each sprint would begin on a Thursday and end on the following Thursday. The Monday between these dates would serve as progress updates to see how the members of the team were progressing with their work. This would allow for other team members to offer insight and help to any members of the team who were struggling.

One of the primary difficulties encountered was the lack of attendance at the meetings. This led ultimately to disorganisation and several team members losing track of time, at some stages members were removed from their assigned task and the task was given to another member to ensure it was completed. Due to this recurring problem the project suffered. This caused more problems when multiple members of the team were assigned to one task, which resulted in several members having a much higher workload than other members assigned to the same task.

Through the use of Trello, members had the ability to keep up to date with due dates and tasks to be completed in the current sprint. Trello provides users with options to increase the urgency of some tasks. Trello also allowed the members of the team to see various other key-dates which were approaching such as the in-class presentations.

Project Review

This project gave all members involved an insight into how Scrum is an effective model for the software development process. Although there were many challenges faced throughout the course of the project, the experience was new to most members, with the exception of two members who had used Scrum in a workplace environment previously.

The game has two different versions: multiplayer and singleplayer. The singleplayer version is available to play in the user's browser and is made in JavaScript. We initially wanted the entire game to be available in the browser, however we were unable to do so due to restrictions. The singleplayer version is played on a HTML canvas element, in which the user uses mouse-clicks and number-keys to play. This version is complete with artwork and functionality, it is also available to anyone with access to a browser. The JavaScript version of the game was very time consuming, party due to the team's familiarity with the language.

The multiplayer game is also available for download from the website, this version allows users to play a simpler version of the game with other players. Having the multiplayer version of the game run in python allowed the team to achieve functionality in a smaller space of time due to already being familiar with the language.

The website on which the game is found is of a simple layout. The website is fully functional and comes complete with a login/registration system. The user can create an account on the website and login again at a later date. The website also features a tutorial which was written by a member of the team. The pages of the site use PHP to output HTML, this allowed the team to reuse PHP code from a single file throughout the whole site.