Texas Hold'em

Sævar Ingi Sigurdarson
CSM0120 - Programming for Scientists

26th October 2020

Contents

1 Executive Summary

This is a python program made to play a simplified version of the famous poker game "Texas Hold'em". The program asks users to input a name and if the user wants to play or quit. Depending on the users input, the game either plays or quits. If the user types anything else the program prompts the user to try again until the answer is start or quit. Once started the program creates a deck of 52 cards that get shuffled and dealt out two cards to the user (that they can see) the AI (that the user cannot see) and three cards to the flop (that the user can see). Once the hands and flop are dealt, the program writes both hands and the flop into a file named "pokerhandhistory.txt" with clear labels to say which is which.

When the user has received their hand and sees the flop, they are then asked if they would want to fold. The user can either type yes or no in relation to the fold question, if the user types "yes" then the hand is folded, and the deck reshuffled as well as the hands and flop getting re-dealt. The user can fold five times. If the user types "no" the program continues the game and evaluates the users and the AI's hands separately with the same flop and determines who the winner is. The program then prints out who won and exits. If the user has folded five times in a row and tries to fold again, the program says that the upper limit was exceeded and plays the hand that was dealt, evaluates the hands and prints the winner.

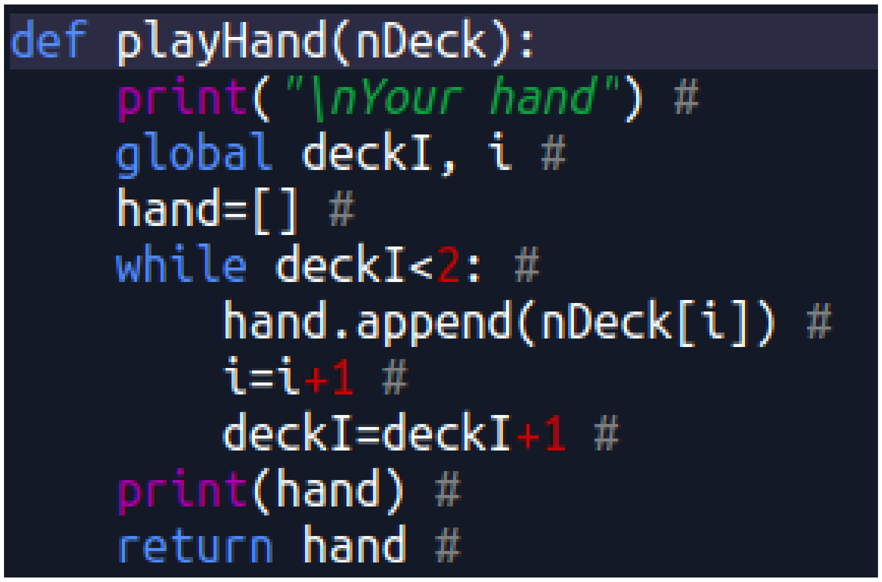
2 Technical Overview

2.1 Data Structures

For the deck in this assignment, the brief specified that it had to be stored in a directory, however since directories cannot have the order altered, it was decided to make a copy of the deck and have that as a list as that can be shuffled and used more like a deck of cards. The hands and flop were made into separate lists once the hands were dealt. After the hands had been separated into the users hand with the flop and the AI's hand with the flop, each and would then need to be separated into ranks and suits, and again lists were used for that as that made it easier to work with when figuring out the values of each hand.

2.2 Loops

A mixture of for and while loops are used, as well as some different ways to get the same effect without using loops. For adding cards from the directory to the lists (as in when adding the hands and the flop) a while loop is used instead of a for loop as it was easier to implement a specific amount of iterations and keep track of how many cards had been used from the deck. As for the iterating through the crated lists to separate them into ranks and suits a for loop was used as for that the whole list was going to be used. Another way the program imitates loops is with if statements and calling the functions that they are in. This can cause problems (which were not noticed till late) with unwanted repetition happening and functions being called many times. Having a if statement that only enters once (and changes the condition for entry in it self) can be used to avoid that, however that is not good programming practice. Figure 1 shows the function for dealing out the users hand.

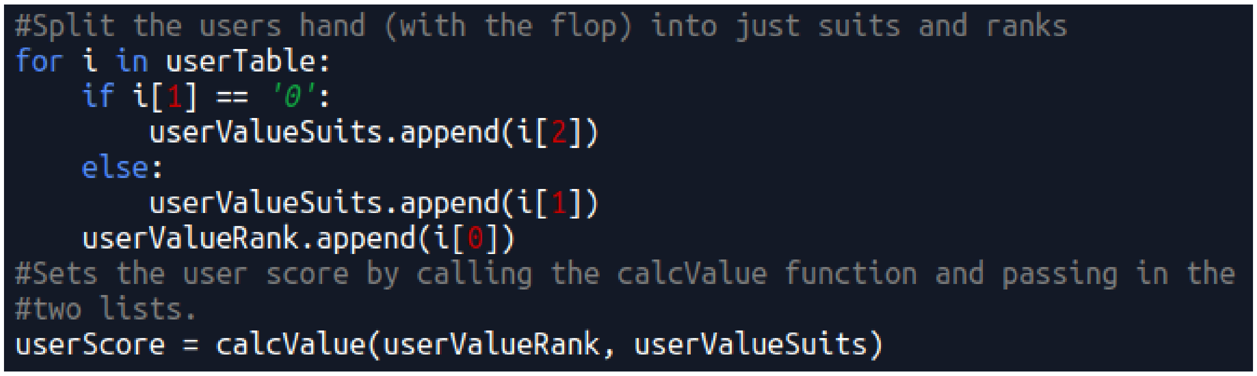


```
def playHand(nDeck):
    print("\nYour hand") #
    global deckI, i #
    hand=[] #
    while deckI<2: #
        hand.append(nDeck[i]) #
        i=i+1 #
        deckI=deckI+1 #
    print(hand) #
    return hand #
```

Figure 1: Function for assigning cards to the users hand

2.3 Special Cases

When processing input, the program reads in from the keyboard and then puts the input into lower cases to accommodate if different users like to write with capital cases or if the caps lock key is down when typing. This is not the case when it comes to the name input as the user should be allowed to specify that themselves. Another special case is when either of the hands have a ten of some suit in, as the program split the suits and ranks for the scoring of the hand. This was then handled with putting in an if statement to check if the second index value was a zero, it would then assign the third index to the suit value to avoid having zero as a suit. Figure 2 shows the for loop and if statement.



```
#Split the users hand (with the flop) into just suits and ranks
for i in userTable:
    if i[1] == '0':
        userValueSuits.append(i[2])
    else:
        userValueSuits.append(i[1])
    userValueRank.append(i[0])
#Sets the user score by calling the calcValue function and passing in the
#two lists.
userScore = calcValue(userValueRank, userValueSuits)
```

Figure 2: Function for splitting the user hand into lists of ranks and suits

2.4 Outputs

The program has different outputs throughout the game that gets played. Most of the output that the user sees is produced with print statements, however, what those statements display is determined by other factors. One such factor is the number of folds the user has done, as well as the program knowing what the user wants to be called. The folds are counted and kept in a global variable to make sure that the user can not fold more than five times. Another printing output is the winner of the hand that is played, and as the program notes the name that the user inputs, the program prints out the users name if they win, and that is determined by a function that compares the scores of each hand with the flop.

3 Software Testing

3.1 Testing the code as a whole

When testing the program to see if it would play the game as expected a few hard-coded lines were added to make sure that the desired outcome would occur. One of the places where this would be is when the user was expected to input text, the code was altered to make it so that if nothing was entered the program would have a default answer (in one case this did make it through to the final program regarding the name of the user). The main one was when the program prompted the user to start and instead of having to type that every time a small change occurred; the tester only needed to press the enter key. That was one of the major ways that was used to help make testing of the code easier. Repeated playing of the game was also a big way to test the program and making sure that it worked, as well as getting other people (not from the course) to test it out and report any problems regarding the program.

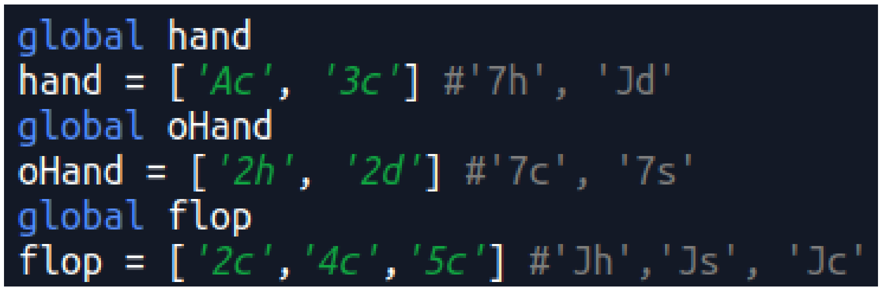
3.2 Functions

3.2.1 Card dealing

For the testing of card dealing and displaying, it came down to how to convert the directory deck into a list and then making sure that once the ”new” deck was there, there would be something in place to keep track of the amount of cards dealt(see Figure 1). Once the hands started displaying something, the program was run repeatedly with minor tweaks between in case something didn’t look right. One issue that arose (after a while, for the ”10” card was not often drawn) was that the “10” cards would display ”0” as a suit, as they had three indexes in the list and the dealing was only looking at the first two indexes. That was handled with an if statement and tested with assigning the hands to make sure that the hand had a “10” of something in it. Figure 2 shows the implementation of getting around hands with a ”10” in them. It is fairly simple and is just a for loop iterating over the hand and flop list. It has an if statement to check if there is a ”10” in the hand, and if there is then look at the next index of said element and add that as the suit.

3.2.2 Card values and scores

For card values and scoring of the hands, the hands were split into their suits and ranks to get an appropriate value for the hand (the splitting method can be seen in figure 2). once the hands had been split into ranks and suits, the calculation of the hand score was a matter of counting the amount of suits and then how many pairs, three of a kind or four of a kind ranks there were. This was done by using the ".count()" function as can be seen in figure 5. Once the scores were received for both hands they were sent to a function that had a very simple if statement that compared them against each other and then returned the winner as a string for the program to print. To test that this had worked, certain hard coded values were implemented, as well as jumping over creating the deck and shuffling. The test hands were made as global variables to make it easy to change and test more. Figure 3 shows the hard coded vaues used for testing.

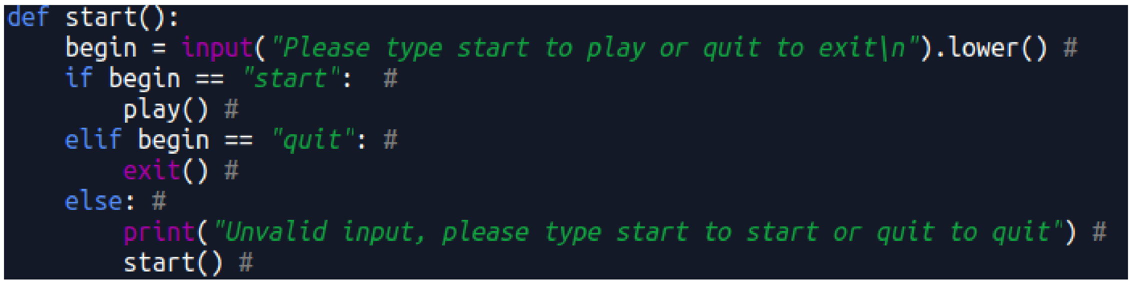


```
global hand
hand = ['Ac', '3c'] #'7h', 'Jd'
global oHand
oHand = ['2h', '2d'] #'7c', '7s'
global flop
flop = ['2c','4c','5c'] #'Jh','Js', 'Jc'
```

Figure 3: Added global variables to test that the programs hand calculations functioned properly

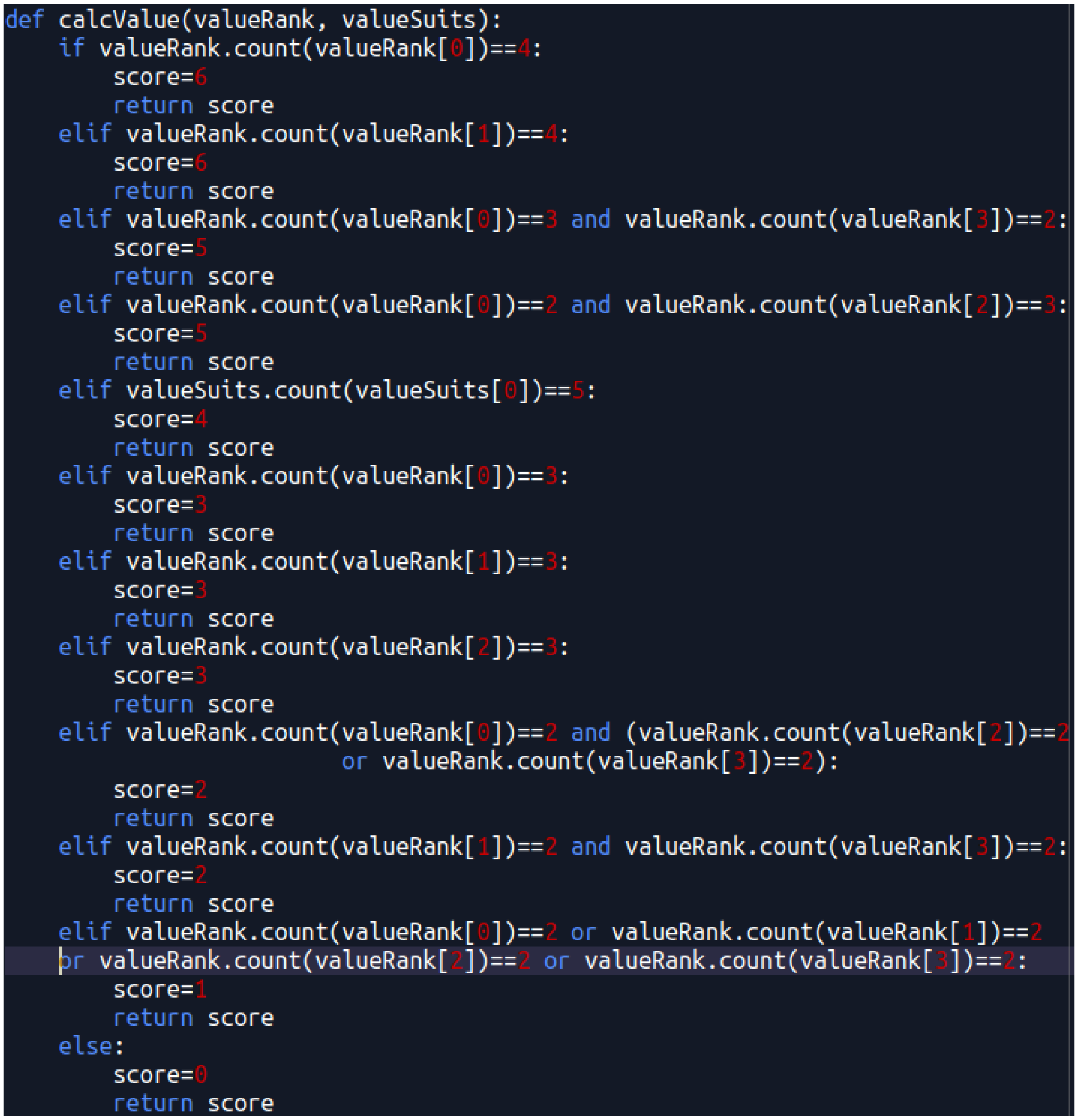
3.2.3 Input

For input handling, the ".lower()" function was added to make it able for the user to type any form of the desired word and the program still understanding it. This helps prevent accidental CapsLock and if the user prefers to capitalise every word. To test the robustness of the input, someone who has not done a lot of programming was asked to test out the input sections to see if the texts made sense and if the tester could crash the program. The start function can be seen in figure 4. Another way of writing this function would have been with a while loop instead of re calling the function if there was an invalid input.



```
def start():
    begin = input("Please type start to play or quit to exit\n").lower() #
    if begin == "start":  #
        play() #
    elif begin == "quit": #
        exit() #
    else: #
        print("Unvalid input, please type start to start or quit to quit") #
        start() #
```

Figure 4: The start function asking the user to input "start" or "quit" to either run or exit the program



```
def calcValue(valueRank, valueSuits):
    if valueRank.count(valueRank[0])==4:
        score=6
        return score
    elif valueRank.count(valueRank[1])==4:
        score=6
        return score
    elif valueRank.count(valueRank[0])==3 and valueRank.count(valueRank[3])==2:
        score=5
        return score
    elif valueRank.count(valueRank[0])==2 and valueRank.count(valueRank[2])==3:
        score=5
        return score
    elif valueSuits.count(valueSuits[0])==5:
        score=4
        return score
    elif valueRank.count(valueRank[0])==3:
        score=3
        return score
    elif valueRank.count(valueRank[1])==3:
        score=3
        return score
    elif valueRank.count(valueRank[2])==3:
        score=3
        return score
    elif valueRank.count(valueRank[0])==2 and (valueRank.count(valueRank[2])==2
                     or valueRank.count(valueRank[3])==2):
        score=2
        return score
    elif valueRank.count(valueRank[1])==2 and valueRank.count(valueRank[3])==2:
        score=2
        return score
    elif valueRank.count(valueRank[0])==2 or valueRank.count(valueRank[1])==2
    or valueRank.count(valueRank[2])==2 or valueRank.count(valueRank[3])==2:
        score=1
        return score
    else:
        score=0
        return score
```

Figure 5: Statement for calculating the score

4 Reflection and Future Work

The assignment went well and is believed to have fulfilled all requirements from the brief. However, after reflecting and learning new things during the course, a few changes would be made, one of those would be implementing a class for the cards. This would give the program a bit more of an object orientation and allow the comparing of hands to check for the same cards easier. Another change, after reflecting on the assignment, would be to try to get rid of all global variables as that could hinder the value being changed by something that shouldn't change it. And it would also make the changing of variables easier to follow as a global variable could be anywhere.

The biggest change that would happen in the future if the program was going to be changed and redone, would be reducing the amount of functions that essentially do the same thing, such as the dealing of hands and the flop in the program. This program has three functions that look very similar, and work very similarly, namely the dealing functions that loop through a certain number of times and add cards to lists. This could be done in one function that gets called multiple times and passes in the amount of cards and where in the deck cards have been removed. Another drastic change would be when the evaluation for the hands takes place, as there is some repeated code there as well. The logic would be moved to a new function and then the users table hand (hand with flop) and the AI's table hand would be passed in separately and the function would then call the calculation function (see figure 5) and that would then return the score of the user and AI to then be compared.