BLACKJACK SIMULATION USING PYTHON

By: Ben Kacikanis

Abstract:

The point of this specific project was to code a blackjack game, as well as explore various strategies including a card counting strategy as well as a static strategy of standing on everything greater than 17. The card counting implementation was to create a dictionary of all the cards in the deck and subtract every card that has been in play. In addition to tracking the deck, I calculated the probability of a "Hit" on a card value that would not result in a "Bust". I then tested various probability thresholds (55,60,70,80,90) to identify the most optimal strategy. This report will be heavily focused on the code implementation. To evaluate the various strategies, I have decided to use win percentage as the performance metric. The major findings was that the counting cards method with a probability threshold of 60% was the most optimal strategy with an average win rate of ~ 47%.

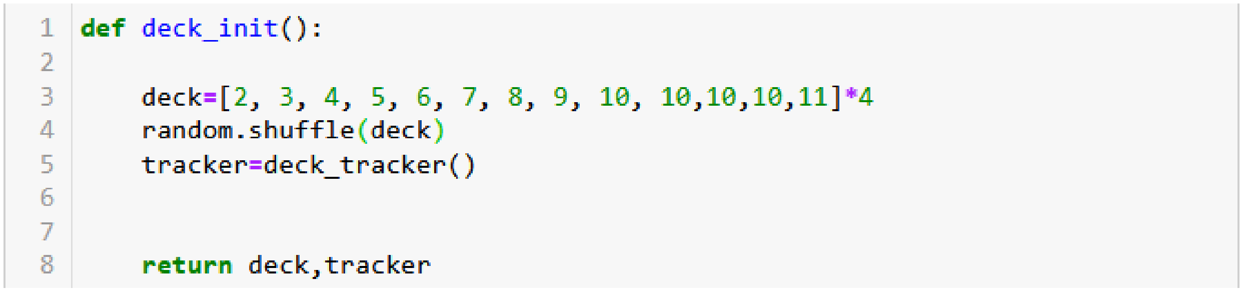
Black Jack Background:

Blackjack is a probability-based game that consists of a dealer and player. The game consists of 3 key events which need to be optimized to increase the probability of winning. The events in chronological order are:1) a player and dealer being dealt 2 cards, 2) "Hit" which results in the player/dealer obtaining a card, 3) "Stand" which ends the player /dealer turn. If both parties stand, the party with the highest total win's, however if a party obtains a total hand of greater than 21 the other party wins. The challenge of the game is that the dealer has a distinct advantage, and more sub-optimal player strategy further reduce winning percentage. This report will further explore causes for dealer/house advantage, potential player advantages, a thorough python-based implementation of blackjack, strategy evaluation, followed by future considerations.

Main Findings:

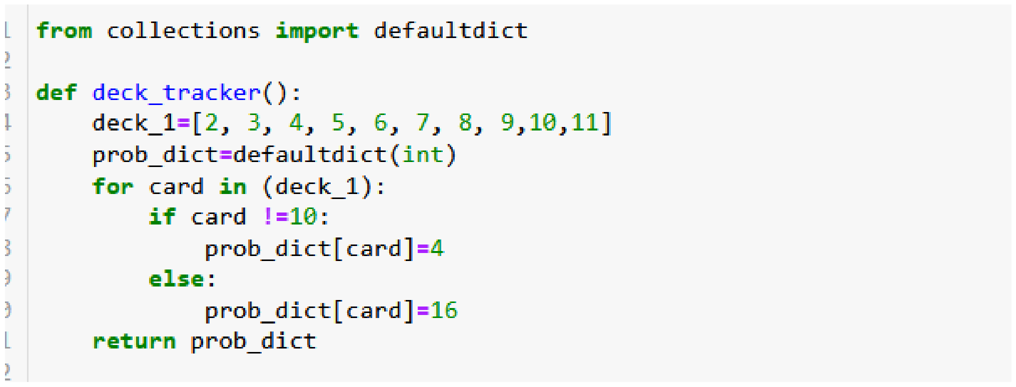
Code Implementations:

The initial task at hand was code development. In order to code a blackjack game I had to develop various helper functions. The first of which was a function to initiate and shuffle a new deck. This was done by creating a list, replacing each face card with a 10, and shuffling the list.



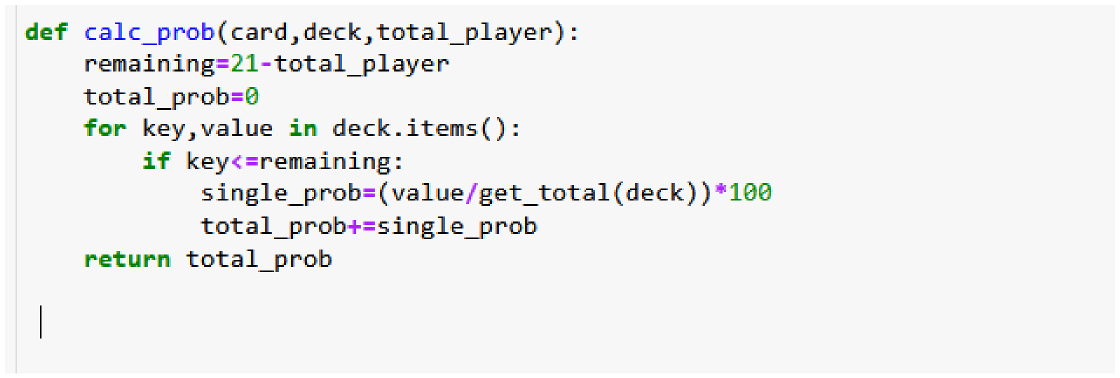
```python
def deck_init():

    deck=[2, 3, 4, 5, 6, 7, 8, 9, 10, 10,10,10,11]*4
    random.shuffle(deck)
    tracker=deck_tracker()


    return deck,tracker
```

When it came to the counting card implementation the approach was to have a dictionary with a key being each card and the value being the total count of the card in the deck. For 10 we had a total of 16 to take into account 10,J,Q,K.



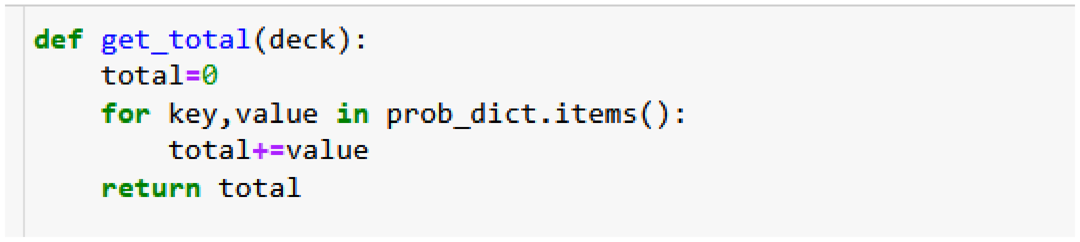
```python
from collections import defaultdict

def deck_tracker():
    deck_1=[2, 3, 4, 5, 6, 7, 8, 9,10,11]
    prob_dict=defaultdict(int)
    for card in (deck_1):
        if card !=10:
            prob_dict[card]=4
        else:
            prob_dict[card]=16
    return prob_dict
```

Another function was created to calculate the probability of the next card given the value required of the next card, and the counts of each card left in the deck. More specifically I subtracted 21 from the value of the players total hand. I then calculated the probability of getting each card less than the highest value before busting. This was done by iterating over the keys in the deck and if the card is less than the remaining value, calculate the probability of obtaining that specific card. The probability calculation was is essentially getting one card or the other card. Since the events are mutually exclusive, the addition rule was used of adding the probability of each event.



```python
def calc_prob(card,deck,total_player):
    remaining=21-total_player
    total_prob=0
    for key,value in deck.items():
        if key<=remaining:
            single_prob=(value/get_total(deck))*100
            total_prob+=single_prob
    return total_prob
```

In addition, another helper function was used to get the total number of remaining cards in the deck, which was used as the divisor in the probability calculations.



```python
def get_total(deck):
    total=0
    for key,value in prob_dict.items():
        total+=value
    return total
```

The last of the helper functions was to initiate and update a record of the game actions. This was a dictionary of the player hand, the dealer hand, the player action that resulted in the win/loss, the player probability as well as a one hot encoded variable of win & lose.

```python
def record_init():
    record = defaultdict(list,{ key:[] for key in ('Player_Hand','Dealer_Hand','Player_Action','Player_Prob','Win','Loss')
    return record
def update_record(record,phand,dhand,action,prob,win,loss):
    record['Player_Hand'].append(phand)
    record['Dealer_Hand'].append(dhand)
    record['Player_Action'].append(action)
    record['Player_Prob'].append(prob)
    record['Win'].append(win)
    record['Loss'].append(loss)
    return record
```

In terms of implementing the blackjack game it was important to implement and initiate variables to keep track and handle various conditions within the game. Loss and Win were each variables initiated at 0 and were updated accordingly, and used to calculate winning percentage. We have variable deck and prob_dict which are initiated from the deck_init function. Variables player_hand starts off as an empty list and appends every player card before a win/loss occurs. Player total is initiated as 0 and is a sum of the player hand. Lastly there is p_stand which is a Boolean variable to indicate whether the player will hit or stand. The same variables for the dealer were also implemented. We have two parameters also included called percent_thresh which indicates what likelihood the player will hit or stand on based on the remaining cards in the deck, and card values needed. The last variable was num_games which indicates how many games were to be played before the loop breaks.

```python
import random
from collections import defaultdict

loss=0
win=0
deck,prob_dict=deck_init()
player_hand=[]
player_index=0
player_total=0

percent_thresh=55
num_games=100
dealer_hand=[]
dealer_index=0
dealer_total=0
d_stand=False
p_stand=False
record_sheet=record_init()
```

The blackjack game structure is as follows. The game is contained within two loops, one being a while loop which checks if the num_games variable has been met, and nested inside is a for loop that iterates over every card in the deck. Once every card in the deck has been played the for loop is exited and the while loop condition is checked, then a new deck and probability dictionary is initiated and the game continues.

Inside of the for loop the game is broken down into multiple conditions. The first step was convert the first index of the deck into a card using list indexing. Then the prob_dict variable containing the count of all cards in the deck was adjusted by subtracting 1 count from the key associated to the card. **See Figure A in Appendix**

Following the adjusting of the card counting/ tracking dictionary, the code deals two cards to the player and the dealer. This is done using the player_index/dealer_index variables and test if the index is less than 2 using an if statement. Inside the if statement for the dealer and player conditions, we append the card to the player_hand/dealer_hand as well as adjust the player_total/dealer_total. In addition two nested conditions are included. One to see if the player/dealer hand > then the threshold we have played which will change the p_stand/d_stand varaible to be true, as well as a condition to see if the player_hand/dealer_hand is equal to 21 which would result in a win/loss and reset all variables. **See Figure B & C in Appendix**

The next two conditions check if p_stand/d_stand variables are equal to False. The calc_prob function was also used and assigned as the player_prob variable calcuate the probability of the next card and later test against the threshold. Nested conditions were coded to check if the player/dealer was above/below the set threshold. If so this will indicate that the action is to "Hit" and the player/dealer will append a card and adjust the total, if more than the threshold but less than 22 the p_stand/d_stand flag was set to False, and if greater than 21 than the loss/win was updated and all variables were reset to default and a continue statement was used to restart the for loop. **See Figure D & E in Appendix**

Ultimately the remaining steps of the black jack implementation consisted of an if statement to check for both p_stand and d_stand variables to equal True which indicates both parties "Stand". When that condition is met a nested set of if statements were used to determine if the player_total or dealer_total was greater which caused the player to win or lose. Lastly the initial variables were all reset to default values to start the game again. **See Figure F in Appendix**

Findings:

Two important considerations were taken into account when implementing blackjack. Initially was the order of the game. I had initially accidently implemented the dealer to go first, in terms of receiving cards and hitting/standing. However the result gave the player the favourite winning slightly more than 50% of the time which went agiainst the house favourite logic. That was later changed to reflect the correct order of the player going first. Prior to implementing the code logic I was not as aware of the game and wondered why the house had an advantage if the actions/events of each party were the same. After implementing and running the game the house has the advantage because the player because since the player goes first, they are put into the position th make the mistake first and hit over 21 without and dealer actions required.

Another important consideration was the game logic. In the code implementation above I mentioned that if the player/ dealer variable for stand was False then they would hit append a card until their thresholds were met. The dealer in blackjack has a constant threshold/ playing logic of standing on 17 and greater. However, in this code implementation we explored various player stratgies. Those being if the player had the same stragety and stand on a total hand of 17 and greater, as well as if the calculated probability of the next card bringing the player total to <=21 based on the player total as well as remaining cards in the deck.

Lastly each iteration of the implementation consisted of 100 games. I had run each strategy for 5 rounds and averaged the result which resulted in 500 games for each strategy.

Strategy Comparisons:

The statistics based on the strategies can be seen below. The lowest win percentage was when the player had the same stategy as the dealer and hit only up to 16. This resulted in the player winning ~ 40% of the time. In addition, the 55% and 80% likelihood threshold scored very similarily while the 70 % and 80% also scored similarly at ~ 42 and 44 respectively. The best strategy so explored was hitting when the likelihood was 60% which resulted in ~ a 47% win rate which was still in favour of the house.

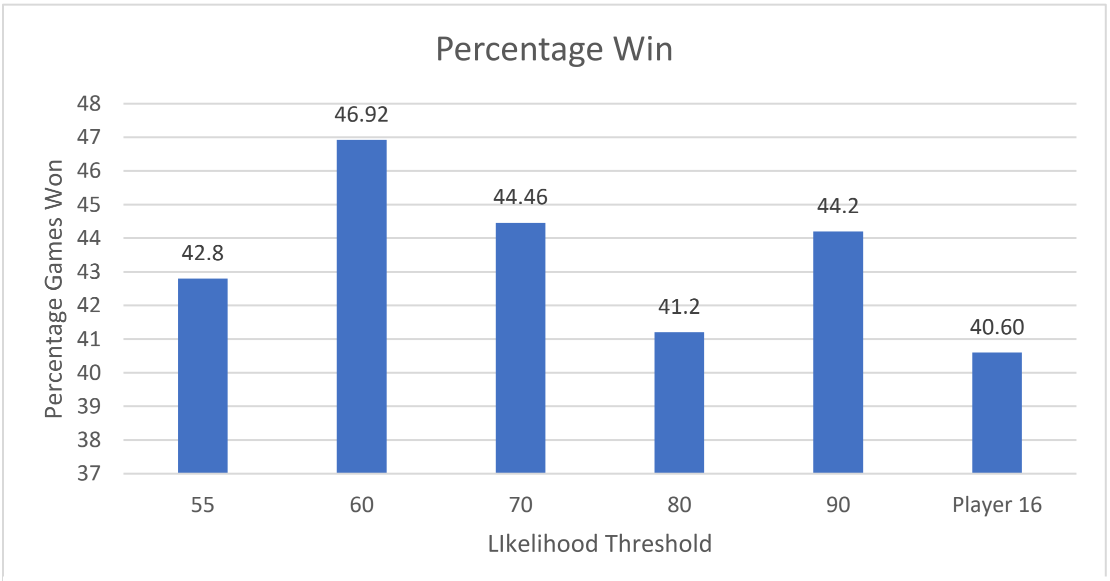


Figure 1: Bar chart of win percentage for each stragegy

A further examination into the distribution of the wins/ lossess brought more insight onto the advantage/disadvantage of each strategy. For Figure 2 which visualizes the wins vs losses for the 60% threshold approach we see that there are clustered wins more towards the later portions when more cards have been used in the deck compared to the begininng portions when we have brand new decks. We are less likely to hit a high probability threshold in the beginning since there are many more cards in the deck/ thus more options of choosing any card and lowering the probability. This differs when comparing against a fixed strategy of a limit of 16 to hit. We would see a more uniform number of wins throughout compared to the clusters we see in the probabilistic card tracking appraoch.

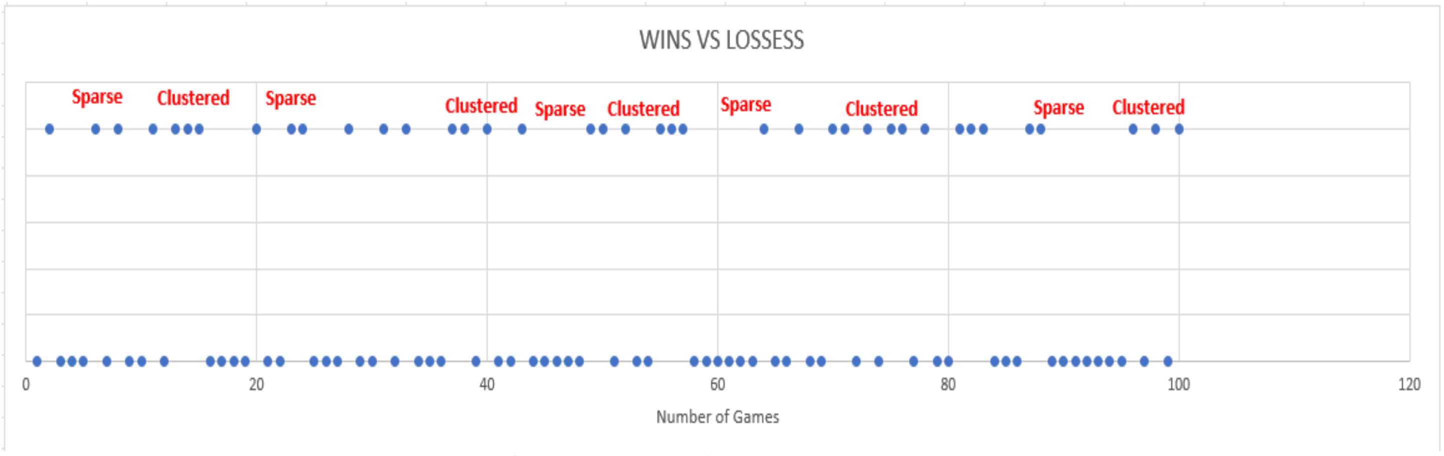


Figure 2: Distribution of wins Vs losses for the 60% threshold stragey

Future Improvements:

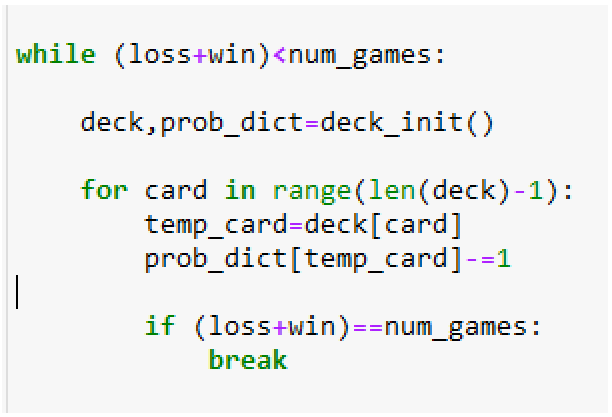
Upon examining the distribution of win’s amongst the 60% threshold strategy as well as the static stand on everything over 17 stratgey, future approach may be a hybrid of both. I propose performing an analysis to investigage at what point the counting card strategy is causing the most losses. More specifically a change point analysis would be done to identify the greatest change in the winning probability. Once the largest loss probability points have been identified using the counting cards strategy, we can replace it with a more static approach like standing when card value<=16. This may further increase the win percentage to beat the house.

Another future consideration to examine is our success metric. The current metrics we are using is the win percentage. However, a better metric to use, especially when applying the counting card strategy is total winnings. I would have to incorporate betting which would require further construction of the code. However, in blackjack not all wins and losses are counted equally. More specifically we can implement a betting array based on different thresholds. Even though we may lose slightly more than the house, we can offset the losses by the size of our bets. For example, we can still implement a probability threshold to hit on, however if we are 75% sure on a specific hand, we may place a much larger bet compared to our 60% threshold to hit.

Conclusions:

In conclusion for this project I had constructed a BlackJack simulation using Python. The implementation included 2 pieces of logic, one being a static approach of hitting on 16 and less for dealer and player, while another player strategy was a counting cards approach with hitting on different probability based thresholds. The findings were that the counting card approach worked best at a threshold of 60%, however there is more work to be done including a change point analysis, as well as betting implementation.

APPENDIX:



```
while (loss+win)<num_games:

    deck,prob_dict=deck_init()

    for card in range(len(deck)-1):
        temp_card=deck[card]
        prob_dict[temp_card]-=1

        if (loss+win)==num_games:
            break
```

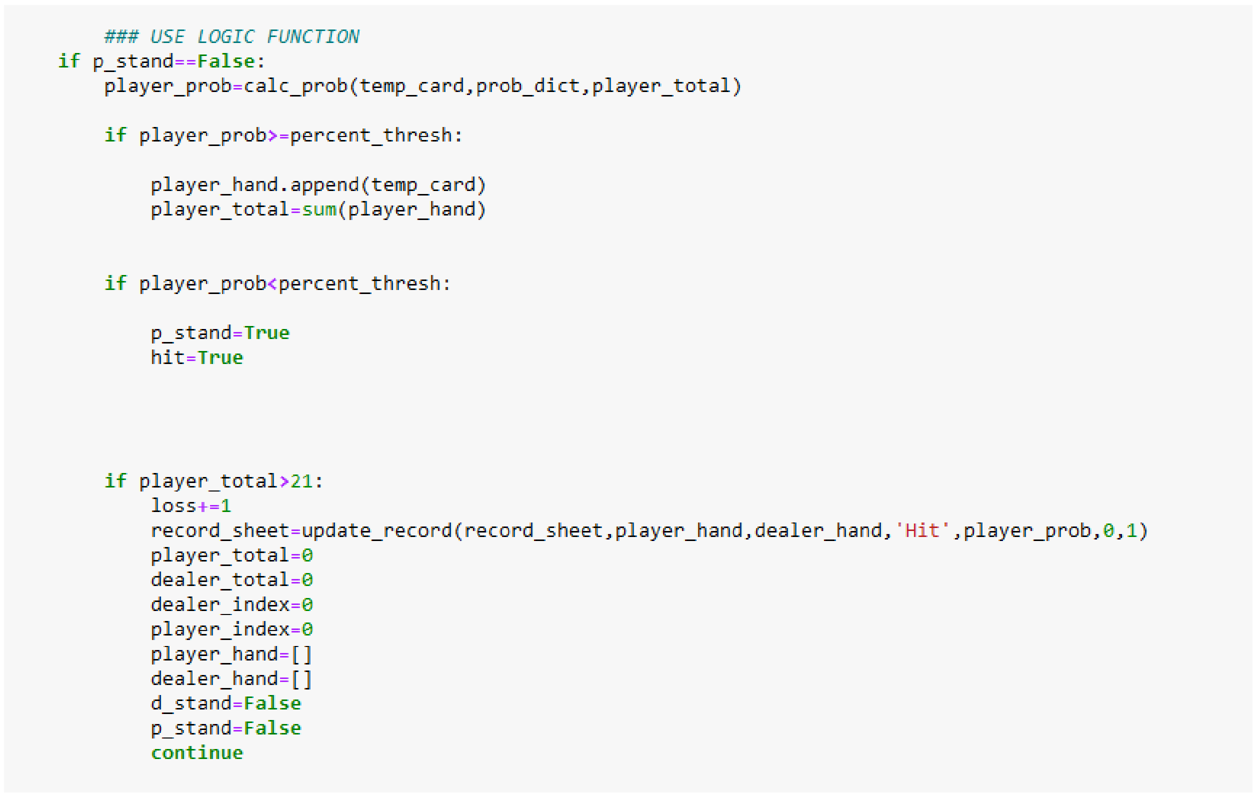
FIGURE A: BEGINNING THE GAME

```python
##Initiate_Player


    if player_index<2:

        player_hand.append(temp_card)
        player_total=sum(player_hand)

        if player_index==1:
            player_prob=calc_prob(temp_card,prob_dict,player_total)
            if player_prob<percent_thresh:
                p_stand=True


        if sum(player_hand)==21:
                win+=1
                record_sheet=update_record(record_sheet,player_hand,dealer_hand,'Start','Start',1,0)
                dealer_index=0
                player_index=0
                player_total=0
                dealer_total=0
                player_hand=[]
                dealer_hand=[]
                d_stand=False
                p_stand=False
                continue
        player_index+=1

        continue
```

FIGURE B: Dealing Player

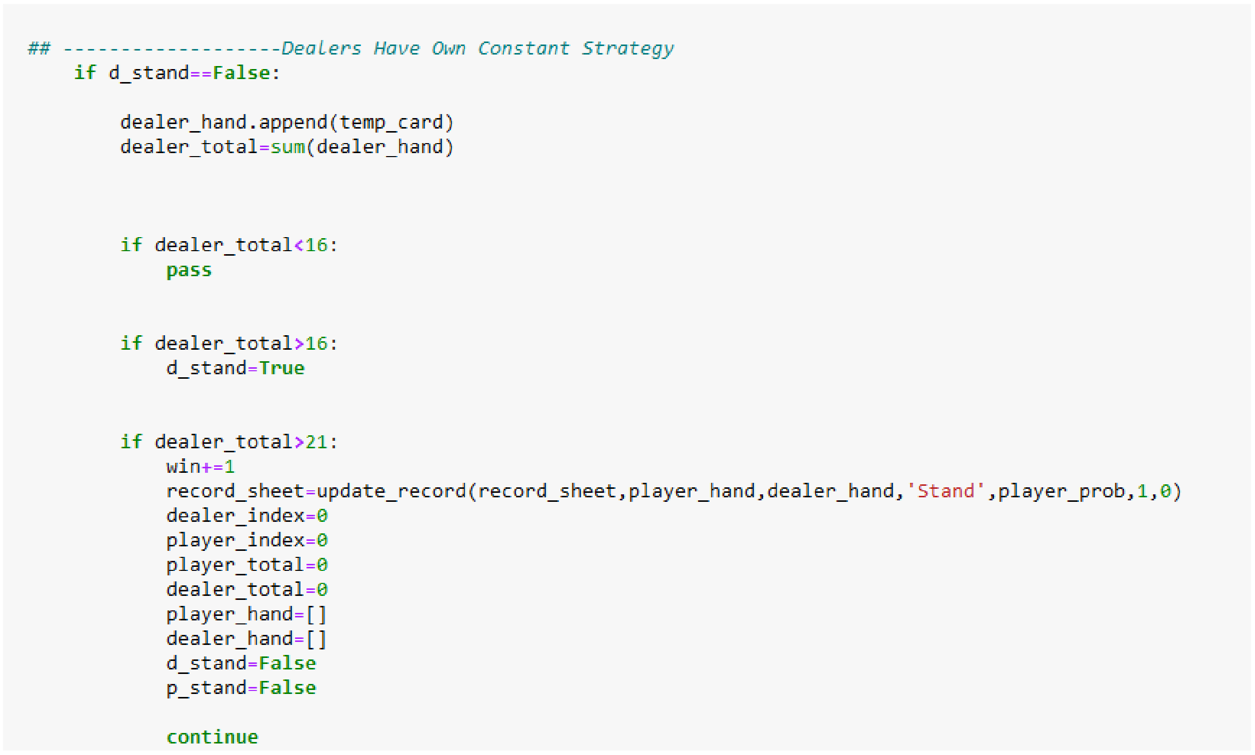
```python
##Initiate_Dealer
if dealer_index<2:
    dealer_index+=1
    dealer_hand.append(temp_card)
    dealer_total=sum(dealer_hand)

    if dealer_total>16:
        d_stand=True

        if sum(dealer_hand)==21:
            loss+=1
            record_sheet=update_record(record_sheet,player_hand,dealer_hand,'Start','Start',0,1)
            dealer_index=0
            player_index=0
            player_total=0
            dealer_total=0
            player_hand=[]
            dealer_hand=[]
            d_stand=False
            p_stand=False
            continue


    continue
```

FIGURE C: DEALING FOR THE DEALER



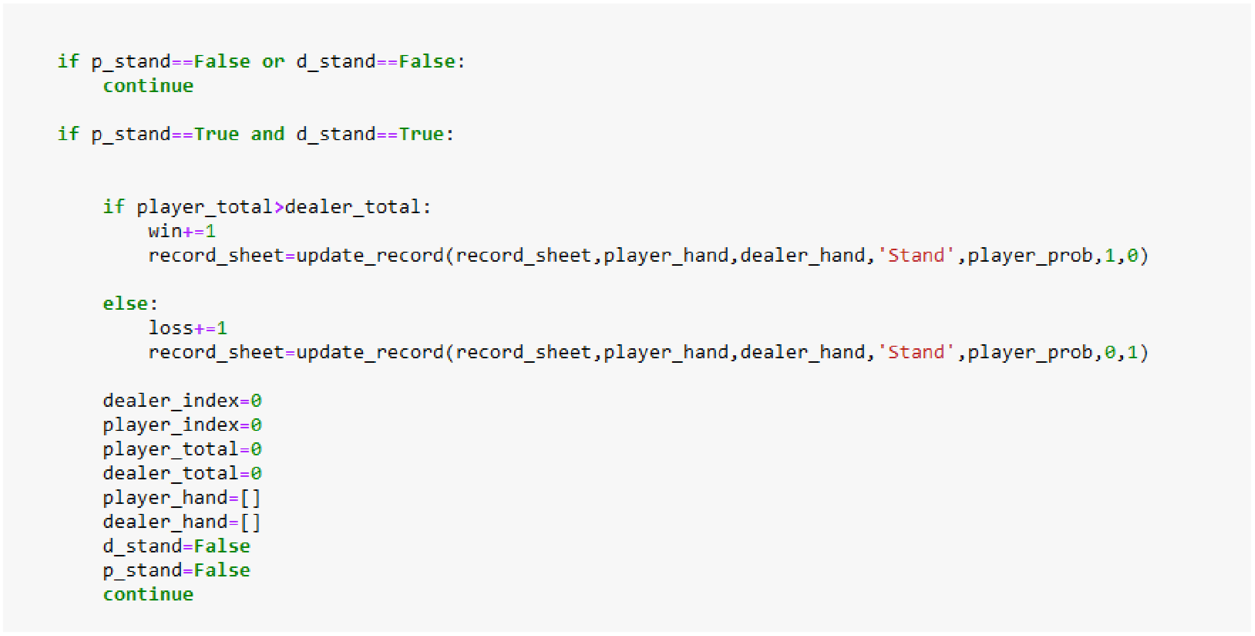
```
        ### USE LOGIC FUNCTION
    if p_stand==False:
        player_prob=calc_prob(temp_card,prob_dict,player_total)

        if player_prob>=percent_thresh:

            player_hand.append(temp_card)
            player_total=sum(player_hand)


        if player_prob<percent_thresh:

            p_stand=True
            hit=True




        if player_total>21:
            loss+=1
            record_sheet=update_record(record_sheet,player_hand,dealer_hand,'Hit',player_prob,0,1)
            player_total=0
            dealer_total=0
            dealer_index=0
            player_index=0
            player_hand=[]
            dealer_hand=[]
            d_stand=False
            p_stand=False
            continue
```

FIGURE D: PLAYER LOGIC CODE USING
PERCENT THRESHOLD APPROACH



```
## -------------------Dealers Have Own Constant Strategy
    if d_stand==False:

        dealer_hand.append(temp_card)
        dealer_total=sum(dealer_hand)



        if dealer_total<16:
            pass


        if dealer_total>16:
            d_stand=True


        if dealer_total>21:
            win+=1
            record_sheet=update_record(record_sheet,player_hand,dealer_hand,'Stand',player_prob,1,0)
            dealer_index=0
            player_index=0
            player_total=0
            dealer_total=0
            player_hand=[]
            dealer_hand=[]
            d_stand=False
            p_stand=False

            continue
```

FIGURE E: DEALER LOGIC CODE USING STATIC 16 VALUE



```
    if p_stand==False or d_stand==False:
        continue

    if p_stand==True and d_stand==True:


        if player_total>dealer_total:
            win+=1
            record_sheet=update_record(record_sheet,player_hand,dealer_hand,'Stand',player_prob,1,0)

        else:
            loss+=1
            record_sheet=update_record(record_sheet,player_hand,dealer_hand,'Stand',player_prob,0,1)

        dealer_index=0
        player_index=0
        player_total=0
        dealer_total=0
        player_hand=[]
        dealer_hand=[]
        d_stand=False
        p_stand=False
        continue
```

FIGURE F: EVALUATING PLAYER VS DEALER TOTAL WHEN BOTH PARTIES STAND