

03-Milestone Project 2 - Complete Walkthrough Solution

July 30, 2019

1 Milestone Project 2 - Complete Walkthrough Solution

This notebook walks through a proposed solution to the Blackjack Game milestone project. The approach to solving and the specific code used are only suggestions - there are many different ways to code this out, and yours is likely to be different!

1.1 Game Play

To play a hand of Blackjack the following steps must be followed: 1. Create a deck of 52 cards 2. Shuffle the deck 3. Ask the Player for their bet 4. Make sure that the Player's bet does not exceed their available chips 5. Deal two cards to the Dealer and two cards to the Player 6. Show only one of the Dealer's cards, the other remains hidden 7. Show both of the Player's cards 8. Ask the Player if they wish to Hit, and take another card 9. If the Player's hand doesn't Bust (go over 21), ask if they'd like to Hit again. 10. If a Player Stands, play the Dealer's hand. The dealer will always Hit until the Dealer's value meets or exceeds 17 11. Determine the winner and adjust the Player's chips accordingly 12. Ask the Player if they'd like to play again

1.2 Playing Cards

A standard deck of playing cards has four suits (Hearts, Diamonds, Spades and Clubs) and thirteen ranks (2 through 10, then the face cards Jack, Queen, King and Ace) for a total of 52 cards per deck. Jacks, Queens and Kings all have a rank of 10. Aces have a rank of either 11 or 1 as needed to reach 21 without busting. As a starting point in your program, you may want to assign variables to store a list of suits, ranks, and then use a dictionary to map ranks to values.

1.3 The Game

1.3.1 Imports and Global Variables

**** Step 1: Import the random module. This will be used to shuffle the deck prior to dealing. Then, declare variables to store suits, ranks and values. You can develop your own system, or copy ours below. Finally, declare a Boolean value to be used to control while loops. This is a common practice used to control the flow of the game.****

```
suits = ('Hearts', 'Diamonds', 'Spades', 'Clubs')
ranks = ('Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine', 'Ten', 'Jack', 'Queen', 'King', 'Ace')
values = {'Two':2, 'Three':3, 'Four':4, 'Five':5, 'Six':6, 'Seven':7, 'Eight':8, 'Nine':9, 'Ten':10, 'Jack':10, 'Queen':10, 'King':10, 'Ace':11}
```

```
[1]: import random

suits = ('Hearts', 'Diamonds', 'Spades', 'Clubs')
ranks = ('Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine', 'Ten', 'Jack', 'Queen', 'King', 'Ace')
values = {'Two':2, 'Three':3, 'Four':4, 'Five':5, 'Six':6, 'Seven':7, 'Eight':8, 'Nine':9, 'Ten':10, 'Jack':10, 'Queen':10, 'King':10, 'Ace':11}

playing = True
```

1.3.2 Class Definitions

Consider making a Card class where each Card object has a suit and a rank, then a Deck class to hold all 52 Card objects, and can be shuffled, and finally a Hand class that holds those Cards that have been dealt to each player from the Deck.

Step 2: Create a Card Class A Card object really only needs two attributes: suit and rank. You might add an attribute for “value” - we chose to handle value later when developing our Hand class. In addition to the Card’s `__init__` method, consider adding a `__str__` method that, when asked to print a Card, returns a string in the form “Two of Hearts”

```
[2]: class Card:

    def __init__(self,suit,rank):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        return self.rank + ' of ' + self.suit
```

Step 3: Create a Deck Class Here we might store 52 card objects in a list that can later be shuffled. First, though, we need to *instantiate* all 52 unique card objects and add them to our list. So long as the Card class definition appears in our code, we can build Card objects inside our Deck `__init__` method. Consider iterating over sequences of suits and ranks to build out each card. This might appear inside a Deck class `__init__` method:

```
for suit in suits:
    for rank in ranks:
```

In addition to an `__init__` method we’ll want to add methods to shuffle our deck, and to deal out cards during gameplay. OPTIONAL: We may never need to print the contents of the deck during gameplay, but having the ability to see the cards inside it may help troubleshoot any problems that occur during development. With this in mind, consider adding a `__str__` method to the class definition.

```
[3]: class Deck:

    def __init__(self):
        self.deck = [] # start with an empty list
        for suit in suits:
```

```

        for rank in ranks:
            self.deck.append(Card(suit,rank)) # build Card objects and add
→them to the list

    def __str__(self):
        deck_comp = '' # start with an empty string
        for card in self.deck:
            deck_comp += '\n '+card.__str__() # add each Card object's print
→string
        return 'The deck has:' + deck_comp

    def shuffle(self):
        random.shuffle(self.deck)

    def deal(self):
        single_card = self.deck.pop()
        return single_card

```

TESTING: Just to see that everything works so far, let's see what our Deck looks like!

```

[4]: test_deck = Deck()
     print(test_deck)

```

```

The deck has:
Two of Hearts
Three of Hearts
Four of Hearts
Five of Hearts
Six of Hearts
Seven of Hearts
Eight of Hearts
Nine of Hearts
Ten of Hearts
Jack of Hearts
Queen of Hearts
King of Hearts
Ace of Hearts
Two of Diamonds
Three of Diamonds
Four of Diamonds
Five of Diamonds
Six of Diamonds
Seven of Diamonds
Eight of Diamonds
Nine of Diamonds
Ten of Diamonds
Jack of Diamonds
Queen of Diamonds

```

King of Diamonds
Ace of Diamonds
Two of Spades
Three of Spades
Four of Spades
Five of Spades
Six of Spades
Seven of Spades
Eight of Spades
Nine of Spades
Ten of Spades
Jack of Spades
Queen of Spades
King of Spades
Ace of Spades
Two of Clubs
Three of Clubs
Four of Clubs
Five of Clubs
Six of Clubs
Seven of Clubs
Eight of Clubs
Nine of Clubs
Ten of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Clubs

Great! Now let's move on to our Hand class.

Step 4: Create a Hand Class In addition to holding Card objects dealt from the Deck, the Hand class may be used to calculate the value of those cards using the values dictionary defined above. It may also need to adjust for the value of Aces when appropriate.

```
[5]: class Hand:
    def __init__(self):
        self.cards = [] # start with an empty list as we did in the Deck class
        self.value = 0 # start with zero value
        self.aces = 0 # add an attribute to keep track of aces

    def add_card(self, card):
        self.cards.append(card)
        self.value += values[card.rank]

    def adjust_for_ace(self):
        pass
```

TESTING: Before we tackle the issue of changing Aces, let's make sure we can add two cards to a player's hand and obtain their value:

```
[6]: test_deck = Deck()
test_deck.shuffle()
test_player = Hand()
test_player.add_card(test_deck.deal())
test_player.add_card(test_deck.deal())
test_player.value
```

[6]: 14

Let's see what these two cards are:

```
[7]: for card in test_player.cards:
      print(card)
```

Nine of Hearts
Five of Hearts

Great! Now let's tackle the Aces issue. If a hand's value exceeds 21 but it contains an Ace, we can reduce the Ace's value from 11 to 1 and continue playing.

```
[8]: class Hand:

      def __init__(self):
          self.cards = [] # start with an empty list as we did in the Deck class
          self.value = 0 # start with zero value
          self.aces = 0 # add an attribute to keep track of aces

      def add_card(self, card):
          self.cards.append(card)
          self.value += values[card.rank]
          if card.rank == 'Ace':
              self.aces += 1 # add to self.aces

      def adjust_for_ace(self):
          while self.value > 21 and self.aces:
              self.value -= 10
              self.aces -= 1
```

We added code to the add_card method to bump self.aces whenever an ace is brought into the hand, and added code to the adjust_for_aces method that decreases the number of aces any time we make an adjustment to stay under 21.

Step 5: Create a Chips Class In addition to decks of cards and hands, we need to keep track of a Player's starting chips, bets, and ongoing winnings. This could be done using global variables, but in the spirit of object oriented programming, let's make a Chips class instead!

```
[9]: class Chips:

      def __init__(self):
          self.total = 100 # This can be set to a default value or supplied by a
→user input
          self.bet = 0
```

```

def win_bet(self):
    self.total += self.bet

def lose_bet(self):
    self.total -= self.bet

```

A NOTE ABOUT OUR DEFAULT TOTAL VALUE: Alternatively, we could have passed a default total value as an parameter in the `__init__`. This would have let us pass in an override value at the time the object was created rather than wait until later to change it. The code would have looked like this:

```

def __init__(self, total=100):
    self.total = total
    self.bet = 0

```

Either technique is fine, it only depends on how you plan to start your game parameters.

1.3.3 Function Defintions

A lot of steps are going to be repetitive. That's where functions come in! The following steps are guidelines - add or remove functions as needed in your own program.

Step 6: Write a function for taking bets Since we're asking the user for an integer value, this would be a good place to use try/except. Remember to check that a Player's bet can be covered by their available chips.

```

[10]: def take_bet(chips):

    while True:
        try:
            chips.bet = int(input('How many chips would you like to bet? '))
        except ValueError:
            print('Sorry, a bet must be an integer!')
        else:
            if chips.bet > chips.total:
                print("Sorry, your bet can't exceed", chips.total)
            else:
                break

```

We used a while loop here to continually prompt the user for input until we received an integer value that was within the Player's betting limit.

A QUICK NOTE ABOUT FUNCTIONS: If we knew in advance what we were going to call our Player's Chips object, we could have written the above function like this:

```

def take_bet():
    while True:
        try:
            player_chips.bet = int(input('How many chips would you like to bet? '))
        except ValueError:
            print('Sorry, a bet must be an integer!')

```

```

else:
    if player_chips.bet > player_chips.total:
        print("Sorry, your bet can't exceed",player_chips.total)
    else:
        break

```

and then we could call the function without passing any arguments. This is generally not a good idea! It's better to have functions be self-contained, able to accept any incoming value than depend on some future naming convention. Also, this makes it easier to add players in future versions of our program!

Step 7: Write a function for taking hits Either player can take hits until they bust. This function will be called during gameplay anytime a Player requests a hit, or a Dealer's hand is less than 17. It should take in Deck and Hand objects as arguments, and deal one card off the deck and add it to the Hand. You may want it to check for aces in the event that a player's hand exceeds 21.

```

[11]: def hit(deck,hand):

        hand.add_card(deck.deal())
        hand.adjust_for_ace()

```

Step 8: Write a function prompting the Player to Hit or Stand This function should accept the deck and the player's hand as arguments, and assign playing as a global variable. If the Player Hits, employ the hit() function above. If the Player Stands, set the playing variable to False - this will control the behavior of a while loop later on in our code.

```

[12]: def hit_or_stand(deck,hand):
        global playing # to control an upcoming while loop

        while True:
            x = input("Would you like to Hit or Stand? Enter 'h' or 's' ")

            if x[0].lower() == 'h':
                hit(deck,hand) # hit() function defined above

            elif x[0].lower() == 's':
                print("Player stands. Dealer is playing.")
                playing = False

            else:
                print("Sorry, please try again.")
                continue
            break

```

Step 9: Write functions to display cards When the game starts, and after each time Player takes a card, the dealer's first card is hidden and all of Player's cards are visible. At the end of the hand all cards are shown, and you may want to show each hand's total value. Write a function for each of these scenarios.

```

[13]: def show_some(player,dealer):
        print("\nDealer's Hand:")
        print(" <card hidden>")

```