Lab 5: BlackJack

Evan Conley

June 6, 2018

# Problem Statement

Create a text-based BlackJack game that utilizes the Card and Chipbank objects from Lab 4.

Requirements:

* An ace shall have a conditional value/rank based on context.
* Cards will have to be randomly selected from a deck.
* A card’s suit, rank and value shall be able to be retrieved if the card is face up.
* Dealers and players will have similar actions to be performed on hands, but only the dealer will have to ability to deal cards.
* A player shall first need to bring money to the table in order to place wagers.
* The system shall determine if the player or dealer has a hand closer to 21, or if their is a tie.
* A player’s card will be drawn from the deck face up.
* The dealer shall have their first card drawn facedown and their second card drawn face up.
* The system shall make updates to the player’s balance based on whether or not they win a hand, lose, or tie.
* The system shall stop a round with a player loss if they ‘bust’ (go over 21).
* The system shall make the dealer hit if they have a total of 16 or below.
* The system shall assess if the player is dealt 21, and will check the dealer’s hand for a tie. If the dealer does not have 21, the system shall grant a player win to the player and update the user’s chip bank with twice the wager.
* The system shall return half a wager if a player ties with dealer.
* The system shall return no money (loss of wager) if they player loses

Assumptions:

* Players will enter the amount of cash they bring to the table for their chip bank.
* The game will only support one player in this iteration.
* The card class will have to be updated with setters in order to change the value of ace cards.
* A chip bank cannot have a negative balance.
* A chip bank cannot be initialized at a zero value, or else the player would have nothing to wager.
* A player cannot make a wager that would bring them to a negative balance.
* If a player reaches zero dollars, they can no longer play.
* It is assumed that a the cash a player brings to the table must be in whole dollar amounts.

Constraints:

* Limited to handling only an individual playing against the dealer.
* Player cannot determine whether they want aces high or low, this will be determined, simply, by if 11 will make the player bust or not.
* There is no minimum buy-in for games.

# Planning

In making the decisions for what the player and dealer should be able to do, I decided to utilize classes, where the Dealer class inherits from the Player class. This allowed me to simplify the Dealer class quite a bit, allowing me to just formulate any additional functionality required. The blackjack game is run through the blackjack.py file and it will create Dealer and Player objects to complete the game. I decided that the game should allow the user a menu to decide what to do each turn, and when they stay/stand, the dealer’s turn begins. To make it to where Python doesn’t just quick-print everything to screen, I want to use the time module to slow the Dealer’s turn. Also, in order to keep the console clean for each hand, I am using a countdown timer and the os module to check for what OS the user has, and then clearing the terminal through whatever means should be used for that OS.

Function Specifications:

NOTE: For information regarding the ChipBank and Card classes, see Lab 4. Further, I removed the ChipBank logging as it didn’t seem helpful for this lab exercise.

Player Class:

Signature: **\_\_init\_\_(self,amount)**

Input: This function acts on the Player class and creates a Player object. It takes an

amount for the Player’s chipbank as input.

Output: None

Description: This is the constructor of the Player class.

Algorithm: This creates a Player object with flags that are associated with calls to hit, stay,

or look at the Dealer’s current face up cards. Also initializes a chipbank and uses the Chipbank class to error check for a positive integer value.

Signature: **signal(self,action)**

Input: This function acts on the Player class. It takes an

action for the player to perform as input.

Output: None

Description: This is a signaling function that will let the player indicate their wish of an action to

perform.

Algorithm: This takes the action (a character) and matches against different cases. It will set

flags indicating to a Dealer the action that is desired.

Signature: **quit(self)**

Input: This function acts on the Player class.

Output: Outputs a goodbye message to screen.

Description: This is the quit function that a player can call to end the game.

Algorithm: This outputs a goodbye message to the user and then uses a sys.exit() call to

stop the game.

Signature: **get\_hit(self)**

Input: This function acts on the Player class.

Output: None

Description: Function returns the state of the hit\_flag associated with a player object.

Algorithm: Return the player’s hit flag value

Signature: **get\_stay(self)**

Input: This function acts on the Player class.

Output: None

Description: Function returns the state of the stay\_flag associated with a player object.

Algorithm: Return the player’s stay flag value

Signature: **get\_hit(self)**

Input: This function acts on the Player class.

Output: None

Description: Function returns the state of the hit\_flag associated with a player object.

Algorithm: Return the player’s hit flag value

Signature: **get\_check(self)**

Input: This function acts on the Player class.

Output: None

Description: Function returns the state of the check\_flag associated with a player object.

Algorithm: Return the player’s check flag value

Signature: **get\_chips(self)**

Input: This function acts on the Player class.

Output: None

Description: Function returns the state of the chip bank associated with a player object.

Algorithm: Return the player’s chipbank by calling the \_str\_ function of the ChipBank class

on a player’s chipbank object.

Signature: **get\_winnings(self, amount)**

Input: This function acts on the Player class and the amount of winnings to be added to

a player’s chipbank.

Output: Prints the amount to be added to the user’s chip bank.

Description: Retrieves the winnings from a game and updates the user’s chip bank.

Algorithm: Prints the amount won and then utilizes a call to the deposit function of the

Chipbank class acting on the player’s chipbank.

Signature: **set\_hit(self)**

Input: This function acts on the Player class.

Output: Print that the user wishes to hit.

Description: Function sets the hit flag to True.

Algorithm: Print’s that the player wishes to hit and then sets the hit flag to True.

Signature: **set\_stay(self)**

Input: This function acts on the Player class.

Output: Print that the user wishes to stay.

Description: Function sets the stay flag to True.

Algorithm: Print’s that the player wishes to stay and then sets the stay flag to True.

Signature: **set\_check(self)**

Input: This function acts on the Player class.

Output: Print that the user wishes to Check the dealer’s upturned cards.

Description: Function sets the check flag to True.

Algorithm: Print’s that the player wishes to stay and then sets the check flag to True.

Signature: **options(self)**

Input: This function acts on the Player class.

Output: Print the options that are available for Player action.

Description: Prints the options available for Play action.

Algorithm: Print the player\_options class variable of the available options of actions.

Signature: **take\_card(self, card)**

Input: This function acts on the Player object and takes a Card object as an input.

Output: None

Description: Take a drawn card.

Algorithm: This will append the card object to the player’s hand variable

Signature: **add\_up(self)**

Input: This function acts on the Player class.

Output: Prints the total of the face up cards in a player’s hand.

Description: Returns the total point value of the cards in the player’s hand.

Algorithm: This function will iteratively check each card in the player’s hand and then, if it is

faceup, will add to player’s total point value for their hand. Returns the total.

Signature: **look\_down(self)**

Input: This function acts on the Player class.

Output: Prints the cards and point value of the user’s hand.

Description: Iteratively prints the face value of each of the card objects in the player’s hand,

and then calls to add\_up() to show the total points in the user’s hand.

Algorithm: Print’s that the player wishes to stay and then sets the check flag to True.

Signature: **instant\_win\_check(self)**

Input: This function acts on the Player class.

Output: None

Description: Function determines if a player has a total value of 21.

Algorithm: Calls add\_up function on the player’s hand and determines if the total is equal to

21. Returns true if so, and false otherwise.

Signature: **bust(self)**

Input: This function acts on the Player class.

Output: None

Description: Function determines if a player has a total value greater than 21.

Algorithm: Calls add\_up function on the player’s hand and determines if the total is greater

Than 21. Returns true if so, and false otherwise.

Signature: **reset(self)**

Input: This function acts on the Player class.

Output: None

Description: Function to discard the player’s current hand.

Algorithm: Sets the player’s hand member variable to an empty list.

Signature: **instant\_win\_check(self)**

Input: This function acts on the Player class.

Output: None

Description: Function determines if a player has a total value of 21.

Algorithm: Calls add\_up function on the player’s hand and determines if the total is equal to

21. Returns true if so, and false otherwise.

Signature: **show\_chips(self)**

Input: This function acts on the Player class.

Output: Returns the chip bank associated with a player to screen.

Description: Function shows the current state of the chip bank.

Algorithm: Calls self.get\_chips() and then prints the current state of the chip bank.

Dealer

NOTE: In addition to inheriting the functions above the Dealer class has the added functionality:

Signature: **first\_deal(self, player)**

Input: This function acts on the Dealer class and takes another player object as input.

Output: None

Description: Function to deal out the first cards of the game to the player and dealer.

Algorithm: Calls the Player class funtion take\_card on the player and self, each twice with

the initialization of a random Card object as the card to take. Then sets the

second card in the Dealer’s hand to face down, so that it is hidden from the player.

Signature: **react(self,player)**

Input: This function acts on the Dealer class and takes another player object as input.

Output: None

Description: Function to react to player’s set action flags.

Algorithm: Reviews the values of the player’s flags and finds which is set. If ‘hit’, then

deal out a card by randomly initializing a card and calling player.take\_card(card)

to deal it. Then set the flag to false and check if the player has hit 21 or busted. In

either case, return true indicating the dealer’s turn, else, return false.

If ‘stay’, then unset the stay flag and return True.

If ‘check’, then look\_down() and print current cards, set the player’s check flag to

False and then return False.

# Implementation and Testing

After deciding on the way in which to structure the constructors for each of the classes, I began to plan out the game with different win and loss conditions. I decided that blackjack game would be held in the blackjack.py and created a cascading structure, where each conditional ultimately leads to a win or loss for the player. At the end of each hand, the user is asked if they wish to play again. If so, the console screen is cleared to start the new hand. If not, they are thanked for playing and the system exits. I tested this by, essentially, playing over and over again until I was confident based upon the randomized outputs that I had successfully handled all found cases for scores and their comparative analysis of the player score and the dealer’s. See Figure 1 for a sample run of the blackjack game.

|  |
| --- |
|  |
| *Figure 1: Sample run of the blackjack game* |

|  |
| --- |
|  |
|  |

# Reflection and Refactoring

Though I think my solution is robust/capable, and I added functionality for screen clearing and using the time functionality to keep the game feeling a bit smoother, I feel I could have cleaned it up by further modularizing the code. I think to do this, I should have create a game class and added functionality to that, and then blackjack.py should have been a run of the initialized game type. This would have cleaned up tests and allowed me to more fully integrate tests for win/loss conditions. Also, although I am happy with the overall product, I think I did do some non-Pythonic ways of handling things. I should have used properties and setters, instead of the explicit get and set methods I created. Further, I should have fixed the setters to do both switching flags to false and true, and just fixed the print statements for hitting and staying elsewhere. Then, I wouldn’t have to do the somewhat unsafe methodology of changing the flags via direct call in the Dealer class.