DSC 20 Project: Blackjack!

Total Points: 100 + EC (10 points)
Submission due (SD time, NO LATE SUBMISSIONS):

- **Checkpoint: Monday, March 7th, 11:59 pm**
- **Final Submission: Wednesday, March 16th, 11:59 pm**

Starter Files

Download final_project.zip. Inside this archive, you will find the starter files for this project.

Use this post to find a teammate if you wish!

Part 0. Coding and Docstring Style

Style Guide: Link here

Testing:

At any point of the homework, use the following command to test your work:

```
>>> python3 -m doctest file_name.py
```

Also you can call one function at the time using:

```
>>> python3 -i file_name.py
```

Checkpoint Submission

You can earn at most **10** points extra credit in total by submitting the checkpoint by the due date above. In the checkpoint submission, you should complete

Part 1, 2, and 3

and submit **card.py, hand.py,** and **shuffle.py** files to gradescope.

Checkpoint submission is graded by completion, which means you can get full points if your code can pass some simple sanity check (no tests against edge cases). Note that in your final submission, you should still submit these questions, and you may modify your implementation if you noticed any errors.

Introduction:

In this project, you will implement a Blackjack game using object-oriented programming techniques.

Before we go into the implementation details, let's go over what Blackjack is. Blackjack is a card game typically played at casinos where each player competes against the dealer to get a score closest to 21, without going over 21. At the start of the game, each player and the dealer are given 2 cards. A person's set of cards is referred to as their ***hand***. The player's own cards (their hand) are visible to themselves and only one of the dealer's cards is visible to everyone. After everyone receives their cards, the dealer asks each player whether they would like to ***hit*** (take another card) or ***stand*** (keep their current cards). Each player can hit (take a card) as many times as they would like to get their score as close to 21 as possible, without going over. Afterwards, the dealer reveals their card and decides to hit or stand. Lastly, the final scores are calculated and the winner is announced.

There are a few more rules to the game that will be described later but this should be enough to get started.

This project consists of 5 major classes: **Card, Deck, PlayerHand, Shuffle,** and **Blackjack**. Additionally, **DealerHand** is a subclass of the PlayerHand and has special restrictions that set it apart from the PlayerHand.

As an overview of how the classes interact, ***Blackjack*** is the main class that ties everything together. In a game of Blackjack, we are given a standard ***deck*** of 52 ***cards***. A game of Blackjack can have multiple rounds. For every round of Blackjack, the deck is ***shuffled*** and cards are passed to the ***dealer and player hands***.

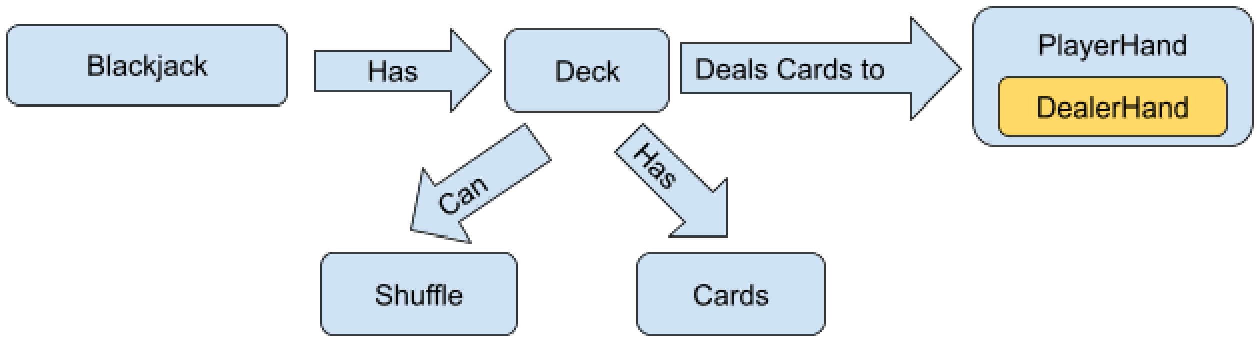


Figure. Overview Diagram of how the classes interact.

Glossary of important Blackjack terms is included at the end of this document.

Part 1: Card Class

Figure. Playing Card with Rank Ace (A) and Suit Spades (♠)

A playing card is the simplest component to any card game and is where we will start this project.

Question 1.1 : Instance attributes of **Card** class

A. **rank (int or str):** Given by the constructor's argument. After initialization, `rank` indicates the rank of the card. The rank represents the value of a playing card and is one of the two main parts of a playing card. The possible values range from 2-10 (inclusive both ends) and include "A" (for Ace), "J" (for Jack), "Q" (for Queen), and "K" (for King).
B. **suit (str)**: Given by the constructor's argument. After initialization, `suit` indicates the suit of the card. The suit represents the category a card belongs to and is the other main part of a playing card. The possible values are "hearts", "spades", "clubs", and "diamonds".
C. **visible (boolean)**: Given by the constructor's argument. After initialization, `visible` indicates whether the card, specifically its rank and suit, is visible. The purpose of this attribute will be more evident in __str__ and __repr__.

Question 1.2 : Methods of **Card** class

A. Constructor (__init__):

The constructor has parameters described in Question 1.1 and the attributes names are the same as these parameters. You should write assert statements according to each attribute description above.

B. Getters

- **a. get_rank(self)**
- **b. get_suit(self)**

C. Representation methods

a. __str__(self)

Returns the string representation of a Card instance as ASCII art containing the rank and suit.

Here, the symbol of the suit will be used instead of the full name. The symbols are as follows: Hearts (♥), Spades (♠), Clubs (♣), or Diamonds (♦).

If the card is not visible, question marks are put in place of the actual rank and suit.

Examples of printed out cards:

```
 ____
|A   |
| ♠  |
|__A|
 ____
|?   |
| ?  |
|__?|
```

More examples are provided in the doctests.

b. __repr__(self)

Returns the representation in this format:

(<rank>, <suit>)

If the card is hidden, question marks are put in place of the actual rank and suit. Examples are provided in the doctests.

D. Comparison methods

a. __lt__(self, other_card)

Playing cards are comparable by their suit and rank. Oftentimes, players will sort their hand to make it easier to calculate the score.

To compare two cards, we will first compare the ranks and only compare the suits if the ranks are equal.

In Blackjack, the suit does not matter so for comparison purposes, we arbitrarily set the sorted order to be the alphabetical order: (*lowest*) Clubs (♣), Diamonds (♦), Hearts (♥), Spades (♠) (*highest*).

For ranks, the sorted order is: (*lowest*) 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace (*highest*)

E. Setter:

a. **set_visible(self, visible)**
Requirements: assert statement

<u>Part 2</u>: PlayerHand & DealerHand Classes

A hand refers to the cards held by a participant. In BlackJack, there are two kinds of participants: player and dealer. The main difference is that the game starts off such that all the cards in a player's hand are visible to the player whereas only the first card in a dealer's hand is visible. Besides that, they share very similar characteristics. So, we define the **PlayerHand** class which is then **<u>inherited</u>** by the **DealerHand** class.

Question 2.1 : Instance attributes of **PlayerHand** class

A. **cards (list):** The cards are not provided in the constructor but you should initialize a list `cards` which will contain the cards of the hand when they're added.

Question 2.2 : Methods of **PlayerHand** class

B. **Constructor (__init__):**
The constructor has no parameters and it only initializes the *cards* instance attribute.

C. **Getters**
  a. **get_cards(self)**
  Getter method that returns the list of cards.

D. **Core Functionality**
  a. **sort_hand(self)**
  Sorts the cards in ascending order.
  b. **add_card(self, *cards)**
  Adds all the cards passed as arguments to the hand, then sorts the hand in ascending order.
  Requirements: assert statement

E. **Representation methods**
  a. **__str__(self)**
  Returns the string representation (ASCII art) of all the cards in the hand, with each card on a new line.
  b. **__repr__(self)**
  Returns the (tuple) representation of all the cards in the hand, with each card separated by a *space* on the same line.

Question 2.3 : Instance attributes exclusive to **DealerHand** class.

A. **hand_visible (boolean):** This attribute indicates whether ~~all the cards in the hand are visible or not~~ the hand was revealed or not.

Question 2.4 : Methods exclusive to **DealerHand** class

B. **Constructor (__init__):**
The constructor inherits the attributes of its parent class, which is **PlayerHand**. In addition, it initializes the instance attribute `hand_visible`. Remember that a dealer's hand is *not* visible by default.

C. **Core Functionality**
   a. **add_card(self, *cards)**
   Adds cards to the hand unsorted while making sure that all but the first card in `cards` are *not* visible. This is the default behavior when the dealer's hand is not visible. If the dealer's hand is visible, then the cards can be added as is and sorted just like it was done in **PlayerHand**.
   Requirements: assert statement
   b. **reveal_hand(self)**
   Makes all the cards in the hand visible and sorts the hand in ascending order.

Part 3: Shuffle Class

As you probably already know, shuffling cards is a very common operation and there are many ways in which one can shuffle a deck of cards. We implement two such techniques.

NOTE: The top of a deck refers to the card that is on top of the deck when the deck is flipped (face down), and it is given by the first element in the list.

Question 3.1 : Modified Overhand Shuffle

In our modified overhand shuffle, you take `num` cards from the middle of the deck and put them on the top of the deck. Then, you take `num - 1` cards from the middle of the deck and put them on the top of the deck. This process goes on until `num` reaches 0.

Notes:
When the number of `cards` is <u>even</u> and `num` is <u>odd</u>, the extra card should be taken from the bottom of the top half of the deck. For eg, when `num` = 3, then take 2 cards from the bottom of the top half, and 1 card from the top of the bottom half of the deck.

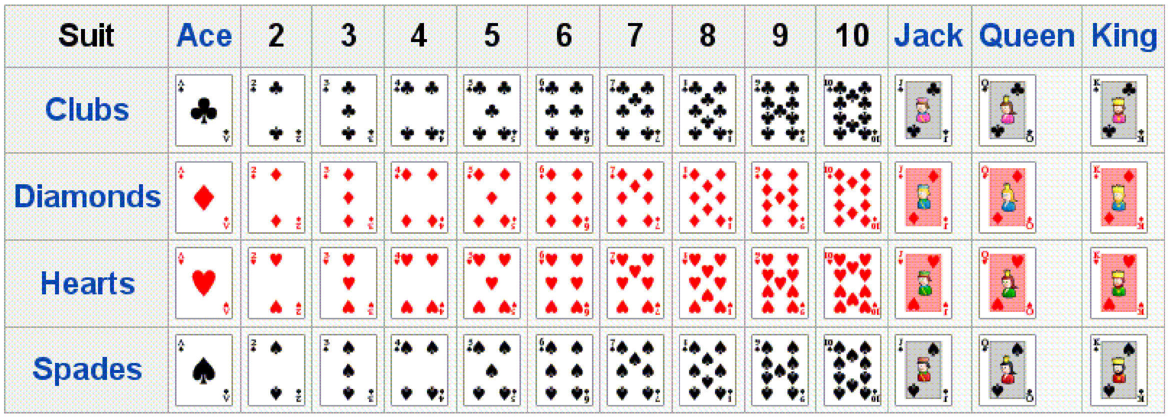
When the number of `cards` is <u>odd</u> and `num` is <u>even</u>, the extra card should be taken from the bottom of the top half of the deck after removing the middle card. For example, if `cards` = [1, 2, 3, 4, 5] and `num` = 2, then the middle card is 3 and the extra card is 2, so you should pick 2 and 3.

Implement this shuffling technique in **modified_overhand(cards, num)** which takes in a list (of cards) and a number (of elements to pick from the middle of the list). You <u>must use recursion</u> to solve this.
Requirements: assert statements, recursion

Question 3.2 : <u>Mongean Shuffle</u>

Description for the Mongean shuffle is [here](). Implement this shuffling technique in **mogean(cards)** which only takes in a list (of cards). Again, you <u>must use recursion</u> to solve this problem.
Hint: you can use helper functions.
Requirements: recursion

Part 4: Deck Class



Suit	Ace	2	3	4	5	6	7	8	9	10	Jack	Queen	King
Clubs													
Diamonds													
Hearts													
Spades													

Figure. All possible cards in a standard deck of playing cards.

Now that you know how to create cards, you're ready to create a deck, which is a set of 52 cards, one for each combination of suits and ranks (13 ranks * 4 suits = 52).

Question 4.1 : Instance attributes of **Deck** class

A. **cards (list):** NOT provided by the constructor's argument. `cards` is a list of Card objects, which is how the Cards in the Deck will be stored. After initialization, `cards` should contain 52 cards in ***ascending*** order, one for each combination of suits and ranks.

Question 4.2 : Methods of **Deck** class

B. **Constructor (__init__):**
   The constructor should initialize and populate cards as described in Question 4.1.
   Requirements: **no explicit for/while loops,** use list comprehension.

C. **Getters:**
   a. **get_cards(self)**
      Getter method that returns the list of cards.

D. **Core Functionality**
   a. **deal_hand(self, hand)**
      Takes the first card from the deck and adds it to the given hand.

      If the hand is visible, the card should be added in sorted order. Notice that Hand has an add_card() method (may be helpful to reduce redundancy).

      You can assume that the deck has at *least* one card.
      Requirements: assert statement for the input parameter

   b. **shuffle(self, **shuffle_and_count)**
      Shuffles the deck using one or both methods defined in the Shuffle class. If both shuffle methods are to be used, *modified overhand* should always be performed before *mongean*.

      After this method is called, the cards instance attribute should contain the shuffled deck.

      Parameter:
         shuffle_and_count: keyword arguments containing the shuffle type (either *modified_overhand* or/and *mongean*) and the number of times the shuffle should be performed.
      Requirements: assert statements

Part 5: Blackjack Class

Woohoo! We are now ready to implement our Blackjack game.

As we mentioned earlier, there are some additional rules and details to the game we haven't discussed in the introduction: (1) how scoring works and (2) how betting works.

How Scoring Works

Calculating the score and finding the winner are discussed in 5.2.G.B and 5.2.G.D. Please read these sections to understand scoring in Blackjack.

Note: Natural Blackjacks (a hand that receives a score of 21 by having an Ace and a card valued at 10) will not be considered for our implementation.

For another overview of the game, please check out this 3 minute YouTube Video by WikiHow: How to Play Blackjack.

How Betting Works (for our implementation)

For the first round of Blackjack, the player must bet $5 from their wallet.

For every round the player wins, they earn their bet amount and the bet required increases by $5 for the next round. For every round the player loses, the player loses their bet amount and the bet required decreases by $5 for the next round. If the game ends in a tie, no money is moved and the bet amount stays the same.

The lowest the player can bet is $5 and they should stop playing if they do not have enough money for the current bet requirement.

Example: Let's say we start with a wallet of $5

Round	Bet (at start of game)	Game Result	Wallet (at end of game)
1	$5	Player Wins	$5 (starting wallet) + $5 (bet) = $10
2	$10	Player Wins	$10 + $10 = $20
3	$15	Tie	$20
4	$15	Player Loses	$20 - $15 = $5 :(
5	$10	Not enough money for a game.	$5

Question 5.1 : Instance attributes of **Blackjack** class

A. **deck (Deck):** NOT provided by the constructor's argument. You should initialize this attribute by yourself. `Deck` is a Deck object, which contains the cards that will be used to play Blackjack.

B. **wallet (int or float):** The amount of money the player has to start with and will be used for betting. This value may increase or decrease as the player wins and loses a game.
C. **game_number (int):** An ***auto-increment*** unique identifier of this Blackjack object. This is not the same as the number of rounds played.
D. **log (str):** NOT provided by the constructor's argument. You should initialize this attribute by yourself as an empty string. The log records all major game activity including errors with starting the game, the player and dealer initial hands, the card that was pulled, etc. More details will be provided later.

Note 1: Auto-increment means that a new unique id number is generated when the instance is initialized, and you can compare the initialization time of two instances of this class by just comparing their auto-generated id. In this project, we will start from 1 (i.e. the first Blackjack game initialized has game number 1), and increment by 1 after each initialization (i.e. the second Blackjack game has game number 2, the third Blackjack game has game number 3, etc.). You can create a class attribute to implement this feature. (call it num_games)

Note 2: Feel free to add extra instance attributes if you think they'll be useful.

Question 5.2 : Methods of **Blackjack** class

E. **Constructor (__init__):**
   The constructor has parameters described in Question 1.1 and the attributes names are the same as these parameters. You should write assert statements according to each attribute description above.
F. **Getters:**
   a. **get_log(self)**
      Getter method for the game log.
G. **Functionality**
   a. **reset_log(self)**
      Clears the game log by setting it to an empty string.
   b. **calculate_score(hand)**
      Calculates and returns the score of the given hand.

      The score is calculated by summing up all the ranks of each card in a hand. Jacks, Queens, and Kings have a value of 10. Aces have a value of 1 or 11. The value of each Ace card is dependent on which value would bring the score closer (but not over) 21. If all possible combinations of Ace values result in a score over 21, the score returned is the score closest to 21.

      Ex #1:
      Given hand: Ace, Ace
      Score: 12 (11 + 1)
      The second Ace card is valued at 1 because valuing it at 11 would bring the score to 22 (over 21).

Ex #2:
Given hand: Jack, 10, Ace, Ace
Score: 22 (10 + 10 + 1 + 1)
The score is over 21 regardless of how we value the Ace cards. Thus, we use the Ace values that bring the score closest to 21.
Requirements: assert statement, **no explicit for/while loops,** use list comprehension, use map/filter.

c. hit_or_stand(self, hand, stand_threshold)

Deal cards to the given hand until the score has reached or surpassed the stand threshold OR there are no more cards in the deck.

For every card pulled, update the log to include
'<Dealer or Player> pulled a (<rank>, <suit>)\n'

For <Dealer or Player>, the log entry will say 'Dealer' if the hand is a DealerHand and 'Player' if otherwise.

Hint: isinstance() may be useful here.

d. determine_winner(self, player_score, dealer_score)

Determine who won the Blackjack round given the player and dealer scores. Return 0 if the Blackjack round ended with a tie, -1 if the dealer won, or 1 if the player won. Update the game log accordingly. More details provided below.

Parameters:
player_score (int): Score the player has after summing up all the ranks of their hand.
dealer_score (int): Score the dealer has after summing up all the ranks of their hand.

Player Winning:
The player wins the game if the player has a closer score to 21, without going over, OR the dealer **busts** (has a score over 21).

If the game ends with the player winning, return 1 and add to the game log:
'Player won with a score of <player's score>. Dealer lost with a score of <dealer's score>.\n'

Dealer Winning:
The dealer wins the game if the dealer has a closer score to 21, without going over, OR the player busts (has a score over 21).

If the game ends with the dealer winning, return -1 and add to the game log:

'Player lost with a score of <player's score>. Dealer won with a score of <dealer's score>.\n'

Tie:
The game is tied if (1) the dealer and the player have the same score OR (2) both player and dealer busts.

If the game ends with a tie, return 0 and add to the game log:
'Player and Dealer tie.\n'

e. **add_to_file(self, player_hand, dealer_hand, result)**

This function writes the summary of every round in a Blackjack game to ./game_summaries/game_summaryX.txt, where X is the game number. For any Blackjack game, the .txt will look like this:

```
ROUND 1:
Player Hand:
 ____
|10  |
| ♠ |
|__10|
 ____
|A   |
| ♣ |
|__A|
Dealer Hand:
 ____
|7   |
| ♣ |
|__7|
 ____
|Q   |
| ♠ |
|__Q|
Winner of ROUND 1: Player

ROUND 2:
.
.
.
```

NOTE: When writing to a file, be careful about what "mode" you open the file in. Also, while using `open()` to write to the file, make sure to set the parameter `encoding = "utf-8"` .

f. **play_round(self, num_rounds, stand_threshold)**

This is the main function that will be called to play Blackjack num_round times. This may seem like a lot of steps but all the methods we've implemented thus far have prepared us for this!

Parameters:

`num_rounds`: The number of rounds to play Blackjack. Though, if there are issues with playing a round at any point. The game will terminate early before playing all rounds.

`stand_threshold`: score threshold for when the player will stand (ie player stands if they have a score >= this threshold)

Requirements: assert statements.

For each round of Blackjack, the following happens:

1. Check that there are enough cards for a single game.
    a. The minimum number of cards needed for a game is 4 (2 cards dealt to the dealer and another 2 dealt to the player).
    b. If there isn't, add to the log "Not enough cards for a game." and stop playing.
2. Check if there is enough money in the wallet to place a bet.
    a. If there isn't, add to the log "Wallet amount $<wallet amount> is less than bet amount $<bet amount>." and stop playing.
3. Shuffle the deck
    a. Shuffle the deck using modified overhand then mongean.
    b. randint() has been imported to randomize the number of times to call each shuffle. Please use this function to randomly select a number between 0-5 (inclusive on both ends) for both shuffles.
    c. **NOTE**: The order randint() is called matters. Please make sure that the first time randint() is called is for mongean and the second time is for modified overhand.
4. Alternate dealing cards between the player and dealer, with the player being dealt first.
    a. Both player and dealer should have 2 cards each after this point.
5. Add to the log the player and dealer cards
    a. 'Player Cards: <card 1> <card 2>'
    b. 'Dealer Cards: <card 1> <card 2>'
6. Player decides to hit or stand using the stand threshold.
7. Dealer's card is revealed and added to the log as
    a. 'Dealer Cards Revealed: <card 1> <card 2>'
8. Dealer decides to hit or stand using the dealer's stand threshold.
    a. Per the rules of Blackjack, dealers must stand if they receive a score of 17 or higher.

9. Update the wallet and bet amount based on the winner and betting rules. (see How Betting Works)
10. Write the results to the game summary.

Note: If the player plays the number of rounds intended OR the rounds end prematurely, the bet is reset to the starting bet of $5 for the next round played.

Submission

Yay! You made it through to the end of this project.

By now, you are ready to submit the project via Gradescope. You may submit more than once before the deadline; only the final submission will be graded. Stay on the submission page until the results show, to make sure that your submission is successfully processed by the Autograder. If your submission fails, the tests are not able to run and you would receive a 0.

Term Glossary

Bust: When the score is over 21.

Hand: A participant's set of cards.

Hit: Action of getting another card from the deck to add to your hand. Hitting can be done multiple times if needed.

Picture Card: A card with a rank of King, Queen, or Jack.

Score: The sum of all the ranks in a participant's hand. Picture cards are counted as 10. Aces can be 1 or 11, depending on which yields a better score.

Stand: Action of not getting another card to add to your hand. Standing will end your turn.