CS220 Project 2: O-O Blackjack

In this project, you will write a multi-class program that simulates the card game blackjack. The game is a simplified version of the "real" game. Your program will handle multiple players. A short description of the rules can be reviewed here (we will be using a simplified version):

`http://www.bicyclecards.com/how-to-play/blackjack/`

Objectives:

- Gain experience with how to write a multi-class program in Python.
- Gain experience implementing an Object-Oriented design.
- Use individual classes to provide a means of implementing a strategy pattern.
- Gain experience using data structures in Python.
- Maintain data integrity in a shared resource.
- Gain experience overriding special methods.

1 Introduction

The program is a simulation of a blackjack game using some simplifications. The game consists of a dealer and a number of players. Each player can implement its own playing strategy. The dealer, itself a kind of player, manages the flow of the game. These are the dealer's main duties:

- Deals cards to the players
- Manages their bets
- Deals itself a hand
- Assesses the winners and losers and collects or pays out the money.
- Provides a string representation of the game.

This version of the program runs one game. Each player bets once, at the beginning of the game. You could modify it to be able to run more than one round, however; that is not required for this assignment.

The main elements in the program are the dealer, players, card deck, and money for betting. Each player, including the dealer, has a hand containing the cards they are dealt. Each player also has an initial amount of money to use for betting. Money will be represented by positive integers.

The deck of cards is modeled by a slightly modified version of the FrenchDeck class described in the text Fluent Python. As in the text, each card in the deck is modeled by a named tuple called Card.

A hand of cards is modeled by the Hand class.

A player's finances, amount available for betting, the "balance" and the amount that has been bet is modeled by the PlayerBank class.

Each player is uniquely identified by its "handle", a string, such as: 'ned21', 'lia9', 'ibad', etc. This handle is used to identify the player in a data structure. It need not be an attribute of a Player instance.

A player is modeled by the Player interface. Actually, this interface models a player's strategy for how it makes decisions about playing the game. Some examples of playing strategies: a player that is conservative in its choices, a player that takes more risk, a player that calculates probabilities based on the cards that have been dealt, etc. Of interest would be a completely random player, or a player that tries to break the game in various ways. Each instance of a player class implements the following methods:

make_bet how much to wager.

want_card take another card from dealer.

use_ace_hi consider the ace as 11, high, or 1, low in scoring a hand.

Note that there are several ways you can implement the player strategy pattern. In Python, you do not have to use an abstract class or interface to achieve the strategy pattern. It is enough that you define player classes that implement the three method listed above and in the Player interface description below in the Code Design section.

The dealer is modeled by the Dealer class. The dealer does participate in the game in a constrained manner, so the dealer class contains an instance of the DealerPlayer class, a subclass of Player. The dealer also maintains the players in a data structure where the player handle is the key, and the values are instances of: Player, Hand, PlayerBank.

Why not encapsulate a player's hand and money in the Player class? In "real life" the player makes choices- what to bet and whether to take a card or not, but a player does not manage the hand or the money, the dealer does. In some forms of blackjack, a player may not touch her cards at all. In code, if a player's strategy is exposed, in the form of a set of public methods, for any arbitrary implementation, it would be possible for that code to make modifications to the hand or money since they would be instance or class variables.

In this program, the player's strategy needs to be isolated from making changes to the player's hand or money, yet the player needs to see the hand and money to make decisions. In other words, the player view is read-only, the dealer view is read/write. One way to accomplish this (and there are others) is to encapsulate the player's hand and money concerns in classes, passing copies to the player. Care must be taken that the copies do not contain references to the copied objects as the player could make changes that would affect the objects the dealer manages. Copies of complex data types with references are shallow copies, while copies that truly copy all data are deep copies.

2 Rules and Flow of the Game

The object of the game is to get a score of 21 or less using the French deck of 4 suits and ranks 2-10, Jack, Queen, King, Ace. A card is scored by its rank. The face cards, 'J', 'Q', 'K', are worth 10, and the ace, 'A' can be scored as 11 ("high") or 1 ("low"), depending on the player's choice. The suits are not used in scoring. If the score of a player's cards, the hand, exceed 21 the player is bust, and loses his bet. The goal is to get as close to 21 without going over. The dealer also has a hand. If the dealer and player have the same score the dealer wins. The game proceeds as follows. The players make a wager that is less than or equal to their stake- the money that start with (we call it the balance, as in a bank account). Note that this is the only time a bet will be made in our program. The dealer deals two cards to each player face-up (visible to all). The dealer also receives two cards, one face-down, not visible to anyone but the dealer. In normal blackjack, a natural is a face card and an ace. This is a special hand and is worth more than another way of getting 21. In this program, we are not implementing this rule.

2.1 Dealing to Players

The dealer processes each player in turn. The player may ask for another card if she is below 21. If she receives another card and the score of her hand is >21 she is bust and her bet is collected by the dealer. The player may request cards as long as she is below 21. The player may also request that aces in her hand be scores as 11, "high", or as 1, "low". Scoring an ace low allows for more cards to be taken.

For example, if these cards were dealt to a player: 5, 7, his score is 12. He asks for another card and receives and ace. His hand is now 5, 7, A, with a score of 23. If the ace is scored low, the

score is 13. He wants to get closer to 21, so he asks for another card and receives 7. His hand is now 5, 7, A, 7, with a score of 20. Needless to say, he declines any more cards. In "real" blackjack, a player may "split" her hand if she is deal two cards of the same rank on the initial deal. We are not considering that rule in this program.

2.2 Dealer Deal

After the dealer has dealt to all players, she deals to herself. The dealer must follow specific rules and has no choices: must take cards until score is >= 17. Aces count as 11 unless the score goes over 21, then aces count low and dealer must take cards until her score is 17 or greater.

2.3 Settle Bets

After the dealer's hand is dealt, the scores are compared and bets are settled. If a player is bust they lose their wager. If a player is 21 or below, he wins if his score is greater than the dealer's score. Of course, if the dealer is bust (over 21) the player wins if he is below 21. A tie score with the dealer means the player loses. If a player wins, the dealer pays double her bet.

We are also not considering blackjack options such as "doubling down" or any side bets in this program.

3 Code Design

The following class definition makeup the design of this project. You are to implement these classes.

Class FrenchDeck in module french_deck

This class is provided and includes the Card tuple definition. The method remove_card is called by the dealer to get a card at random from the deck. The constructor allows a seed to be passed in to initialize the random number generator.

Class Dealer in module dealer

This class runs the game. It contains the players, their hands, and bank objects. It also manages the list of cards dealt- cards visible to all players- as well as its own hand. It does not need a PlayerBank instance in this version of the game.

Attributes:

_deck: the FrenchDeck instance.

_dealer: an instance of a subclass of Player that implements the deale's rules.

_dealer_hand: an instance of Hand that contains the dealer's cards.

_players: contains key-value pairs, where the key is the unique player handle and the value associated with the key is <Player, Hand, and PlayerBank>.

cards_dealt: a list of Cards that have been dealt "up"- visible to all players. This list (copy) is passed to players in case they want to use this data for strategizing.

Methods:

A constructor that initializes the instance with a FrenchDeck instance passed in. This allows the deck object to be instantiated with a seed by an outside test class. The dealer is initialized to an instance of a subclass of Player. Call it DealerPlayer. Put all the dealer's "strategy" in those abstract method implementations. Do this in the Dealer class constructor. The players is initialized to an empty dictionary, and the cards_dealt is also initialized to an empty list.

add_player: takes a string- a unique name, or "handle" for the player, as well as Player and PlayerBank instances as parameters, does not return a value. It creates a new Hand instance and stores the three classes in the "players" data structure using the player's handle as the key. The method raises a RuntimeError if the player handle is already a key in the data structure.

take_bets: no parameters or return. Iterates through all players and calls their make_bet method, passing in their balance (since this is not stored in the player class). It then enters the bet into the PlayerBank object by calling the enter_bet method, passing in the player's bet. Raise a RuntimeError if a player's bet exceeds the player's balance.

deal_initial_hand: no parameters or return. Deals two cards from the deck to each player- the cards are stored in the player's Hand instance with add_card. The cards are also stored in the cards_dealt list. The dealer also gets two cards, though the second card is dealt down, so it does not go in the cards_dealt list.

deal_player_hands: no parameters or return. This method implements the rules described above in the Dealing to Players section. Each player is asked if it wants a card and if it wants to calculate the score using ace high or low. The Player methods are want_card and use_ace_hi. The dealer passes copies of the required data to the player (see Player interface). The dealer updates the player's hand with the score after the player is processed, i.e. has stopped taking cards or has bust. Remember that only the dealer can update a player's hand with cards and score. Remember that all cards dealt should be added to the cards_dealt list.

deal_dealer_hand: no parameters or return. This method implements the rules described in the Dealer Deal section above. The process is similar to the player deal. Be sure to update the dealer's hand and the cards_dealt list.

settle_bets: no parameters or return. Implements the rules described in the Settle Bets section above. The player's score is gotten from the Hand object and compared to the dealer's score. The PlayerBank methods pay_winner and bust are called to update the player's bank object.

str: override the `__str__` special method to produce the output shown in figure 2 below.

Notes: The "players" data structure is easily handled by a Python dictionary. The dealer is represented by a Player and Hand instance but no PlayerBank. You could add this, but for simplicity it is not necessary as we aren't keeping track of the house account. The dealer-related classes could be stored with the rest of the players or not. The Hand and PlayerBank hold data and methods that are modifiable by the dealer but not by the player. You will want to define private helper methods in this class.

Interface Player

Methods:

make_bet: takes a PlayerBank object and returns an int- the amount of the bet.

use_ace_hi: takes a Hand object as parameter- the player's hand (a copy), and return True if the player want's the ace to be scored as 11, False if the ace should be scored as 1.

want_card: takes the following parameters: a Hand object- the player's hand, a PlayerBank object- the player's bank account, the dealer's hand- a list of one Card- this is a single card that was dealt up, and the `cards_dealt` list, the list of Cards that are visible to all players. Remember that any objects like lists or classes passed in must be copies. Returns True if a card should be dealt (a "hit"), False otherwise ("stand").

Class PlayerBank in module player_bank

Attributes:

_balance: the current amount available for betting, an int. Initialized in constructor.

_bets_placed: the amount of bets placed, an int. Default = 0

_is_winner: True if player wins, False otherwise. Default =False

Methods:

constructor: initializes the balance (this is the initial "stake").

pay_winner: an amount passed in to be added to player's balance. Set is_winner to True. No return.

bust: sets is_winner to False

get_balance: return the balance.

get_wager: return the bets_placed.

enter_bet: takes an int parameter-the amount the player has bet. adds to bets_placed, subtracts from balance. Raises aRuntimeError if the bet is greater than the balance.

str: override the `__str__` special method to produce the output shown in figure 2 below.

Class Hand in module hand

Attributes:

_cards: a list of Cards

_score: the score of the players hand.

Methods:

constructor: initializes the cards list to an empty list and score to 0.

add_card: adds a Card to the cards list.

get_score: return the score.

set_score: overwite the score passed in.

has_ace: True if an ace is in the hand, False otherwise.

score_hand: take a boolean parameter- if True score with ace = 11, if False ace = 1.

get_cards: return a copy of the cards list.

len: override the `__len__` special method to return the number of cards in the hand.

str: override the `__str__` special method to produce the output shown in figure 2 below.

Class ConservativePlayer in module conservative_player

Implements the Player interface. Also, override the `__str__` special method to produce the output shown in figure 2 below.

Class RandomPlayer in module random_player

Implements the Player interface. Also, override the `__str__` special method to produce the output shown in figure 2 below.

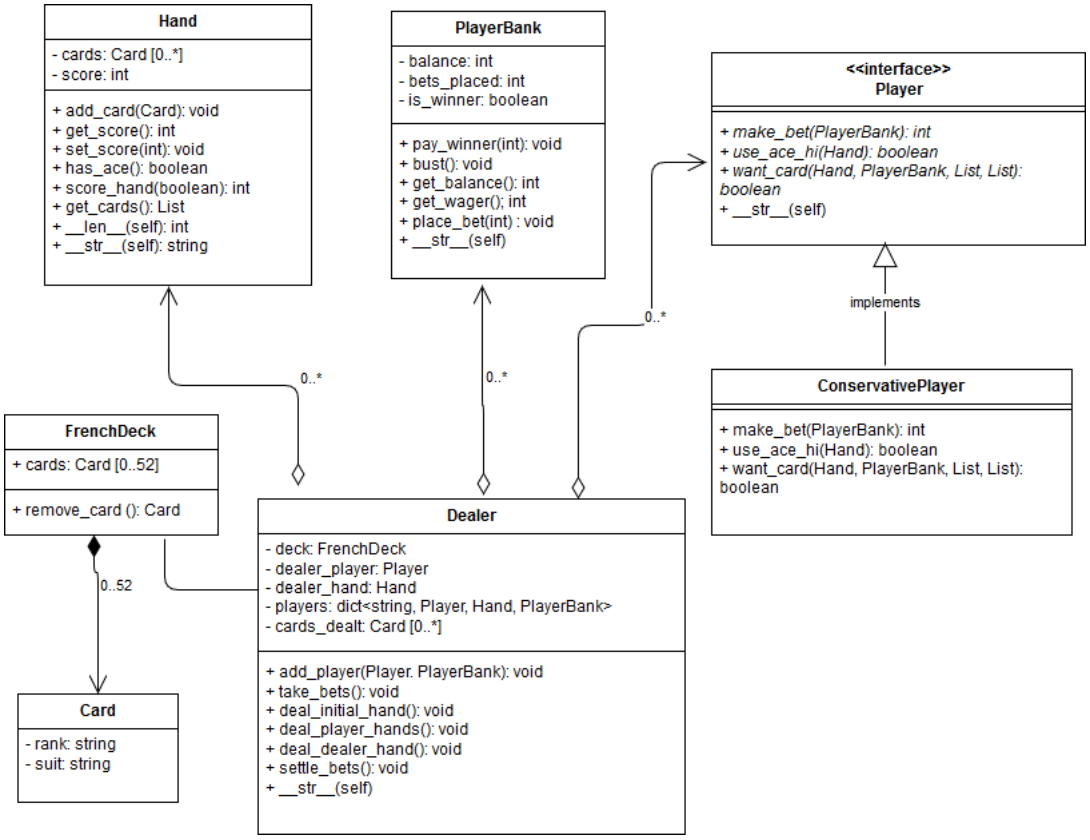

Figure 1: UML Class Diagram

4 Testing

Suggested: implement a Tester module that creates an instance of FrenchDeck with a seed, passes it to an instance of Dealer:

```
dealer = Dealer(FrenchDeck(12345))
```

Add players to the dealer:

```
dealer.add_player('lia9', ConservativePlayer(), PlayerBank(100))
```

Note that you might want to create a DealerPlayer class that carries out the rules the dealer must follow when dealing to itself. This class can be instantiated in the Dealer class. You can process this object separately from the other players in the deal_dealer_hand() method. Then make the method calls to play a game and view the output:

```
dealer.take_bets()
dealer.deal_initial_hand()
dealer.deal_player_hands()
dealer.deal_dealer_hand()
dealer.settle_bets()
print(str(dealer))
```

Use the script provided to zip up your code and submit it to Moodle.

Output Examples

```
$$$$$$  Game Summary   $$$$$$
Dealer:
score: 21
Card(rank='6', suit='clubs') Card(rank='2', suit='hearts') Card(rank='7', suit='clubs') Card(rank='6', suit='spad

Player: ned21
score: 27
Card(rank='7', suit='spades') Card(rank='K', suit='diamonds') Card(rank='K', suit='spades')
Player assets:
bet 10 balance 90 bust.

Player: lia9
score: 25
Card(rank='5', suit='spades') Card(rank='J', suit='diamonds') Card(rank='J', suit='clubs')
Player assets:
bet 10 balance 90 bust.
```

Figure 2: Output Example 1

```
$$$$$$  Game Summary   $$$$$$
Dealer:
score: 18
Card(rank='J', suit='spades') Card(rank='2', suit='clubs') Card(rank='6', suit='clubs')

Player: ned21
score: 19
Card(rank='6', suit='hearts') Card(rank='Q', suit='diamonds') Card(rank='3', suit='hearts')
Player assets:
bet 10 balance 110 winner!

Player: lia9
score: 20
Card(rank='Q', suit='hearts') Card(rank='J', suit='clubs')
Player assets:
bet 10 balance 110 winner!
```

Figure 3: Output Example 2