Lab 4: Casino Night
Kyle Calvert, Evan Conley
May 21, 2018

1. Problem Statement

Create a Card class to represent cards in a 52 card deck and a ChipBank class to represent a collection of chips for wagering.

Requirements:

- A card shall have a suit, rank, and value.
- A card shall be have the option to be face up or face down.
- A card's suit, rank and value shall be able to be retrieved if the card is face up.
- A chip bank shall have a balance.
- A chip bank's balance shall be able to displayed in terms of chip amounts and total cash worth.
- A chip bank's balance may be accessed to make a withdrawal or deposit.
- A chip bank's transaction requests will be logged in a text file.
- A chip bank may be set to log transactions or not.

Assumptions:

- Each card will have will have an value from 0 to 51 associated with it; the identifying number for each card will be unique.
- A card will be initialized as face up.
- The card class will have no setter functions, as its value is assumed to remain constant for its lifetime in a game.
- A chip bank cannot have a negative balance, and, therefore, must be initialized at 0 or greater.
- A chip bank cannot make a withdrawal if it would create a negative balance.
- A chip bank is initialized with the option to log transactions.

Constraints:

- Card class has no guard against creation of duplicate cards.

2. Planning

In discussing how to assign each Card a unique identifying number to be initialized with, we decided to create a modular structure. In order to do this, we created a list of tuples that associated value and rank of cards. Then, based upon the ranges of 0-12, 13-25, etc. are associated with a different suit. Then, the number is modded by 12, and used to index into the list of tuples to grab a specific card value and rank, giving us all the information needed from input to create a Card. As to the ChipBank class, in order to keep track of the chip values and amounts, we used a 4-tuple structure and dictionary, respectively. In order to print out the chips and amounts required for a specific ChipBank balance, we utilize integer division and iterate through all potential chip values to test against.

Function Specifications:

Card Class:

Signature: **__init__(self,card_number)**
Input: The function acts on the object calling it, the Card (self), and takes in an integer input (card_number) to determine what information from our class deck_values list should be used to populate the Card.
Output: None
Description: This is the constructor of the card class. It will take an integer value from 0 to 51 and create the Card object associated with that number.
Algorithm: The card_number is checked against a set of ranges. If it is below 0 or greater than 51, a ValueError is raised. If it is within bounds, it will check ranges of 13 numbers (0-12, 13-24, etc.) to determine suit of the Card object. Then, the card_number input will be modded by 12 to get it within range of the deck_values class member list. The Card is then given the value of the tuple in the deck_values list indexed by the modded card_number. The card will also be set with face_up as True, allowing it to be viewed.

Signature: **face_up(self)**
Input: The function acts on the object calling it, the Card (self).
Output: None
Description: This function allows the face_up value associated with a Card object to be set to True.
Algorithm: The Card object's face_up field is accessed and assigned the value of True.

Signature: **face_down(self)**
Input: The function acts on the object calling it, the Card (self).
Output: None
Description: This function sets the Card object to face_down.
Algorithm: The Card object face_up field is accessed and set to False.

Signature: **get_suit(self)**
Input: The function acts on the object calling it, the Card (self).
Output: Returns the suit field associated with the Card object.
Description: This is a getter function created to retrieve a Card object's suit.
Algorithm: The Card's suit field is accessed and returned to the caller.

Signature: **get_rank(self)**
Input: The function acts on the object calling it, the Card (self).
Output: Returns the rank field associated with the Card object.
Description: This is a getter function created to retrieve a Card object's rank.
Algorithm: The Card's rank field is accessed and returned to the caller.

Signature: **get_value(self)**
Input: The function acts on the object calling it, the Card (self).
Output: Returns the value field associated with the Card object.
Description: This is a getter function created to retrieve a Card object's value.
Algorithm: The Card's value field is accessed and returned to the caller.

Signature: **__str__(self)**
Input: The function acts on the object calling it, the Card (self).
Output: None
Description: This is a method to create a String value that summarizes the information related to a Card object.
Algorithm: The Card object's face_up field is checked to determine if the Card object's information should be visible. If it is face_up, return a summary of the Card object's suit, rank, and value. If it is not, then return the String "<facedown" to alert the user that the values are hidden.

ChipBank:
Signature: **__init__(self,amount)**
Input: The function acts on the object calling it, the ChipBank (self) and an integer value representative of the ChipBank's account balance (amount).
Output: None
Description: This is a constructor method, creating a ChipBank object.
Algorithm: The amount parameter is checked to ensure that it is greater than or equal to 0. If it is, then the ChipBank is initialized with an account balance of the passed in amount, a chip_value tuple and a dictionary relating the chip types and the amount of each chip needed to achieve the balance amount. Also, the ChipBank is set to log all transactions using the should_log field.

Signature: **no_log(self)**
Input: The function acts on the object calling it, the ChipBank (self).

Output: None
Description: This sets the boolean value associated with the should_log field to False.
Algorithm: The ChipBank's should_log field is accessed and set to False.

Signature: **log(self)**
Input: The function acts on the object calling it, the ChipBank (self).
Output: None
Description: This sets the boolean value associated with the should_log field to True.
Algorithm: The ChipBank's should_log field is accessed and set to True.

Signature: **withdraw(self,amount)**
Input: The function acts on the object calling it, the ChipBank (self), and takes in the integer amount of a withdrawal from the ChipBank balance as an input (amount).
Output: None
Description: This function makes a withdrawal of size amount from a ChipBank object's balance.
Algorithm: The resultant balance of the ChipBank after the proposed withdrawal is checked. If it is less than 0, then an error is raised. In order to handle the aforementioned error, if the ChipBank object is set to log transactions, it will open up the constant file handle "banklog.txt" with writing privileges and append a note about a withdrawal that would've exceeded the ChipBank object's balance. Else, the amount is withdrawn from the ChipBank balance. Then, if the ChipBank is set to log transactions, the constant file handle "banklog.txt" is opened with writing privileges. Then, a note is appended to show the current balance and the withdrawn amount.

Signature: **withdraw(self,amount)**
Input: The function acts on the object calling it, the ChipBank (self), and takes in the integer amount of a withdrawal from the ChipBank balance as an input (amount).
Output: None
Description: This function makes a withdrawal of size amount from a ChipBank object's balance.
Algorithm: The amount is deposited into the ChipBank balance. Then, if the ChipBank is set to log transactions, the constant file handle "banklog.txt" is opened with writing privileges. Then, a note is appended to show the current balance and the deposit amount.

Signature: **get_balance(self)**
Input: The function acts on the object calling it, the ChipBank (self).
Output: None
Description: This is a getter function created to retrieve a ChipBank object's balance.
Algorithm: The ChipBank's balance field is accessed and returned to the caller.

Signature: **_init_chip_amounts(self)**
Input: The function acts on the object calling it, the ChipBank (self).
Output: None
Description: This is a helper function called by the count_chips method to reset the chip amounts.
Algorithm: The chip_amounts dictionary associated with a ChipBank object is iterated through and all chip amounts are set to 0.

Signature: **chip_amounts(self,amount)**
Input: The function acts on the object calling it, the ChipBank (self) and an associated amount to check the amount of chips against.
Output: None
Description: This method returns the amount of each type of chip needed to correctly add up to the current ChipBank value.
Algorithm: The method iterates over the chip_values associated with a certain chip. It, then, utilizes integer division on the amount passed in to check the number of each chip that can be used to sum up to the total balance. After each iteration, it sets the chip_amounts field of the ChipBank object to the amount ascertained from the integer division.

Signature: **set_balance(self,amount)**
Input: The function acts on the object calling it, the ChipBank (self) and an integer amount.
Output: None
Description: This is a setter function created to set a ChipBank object's balance.
Algorithm: The ChipBank's balance field is accessed and set to the amount specified.
Signature: **__str__(self)**
Input: The function acts on the object calling it, the ChipBank (self).
Output: None
Description: This method returns a String object summarizing the data about a ChipBank object. Specifically, it will output the current balance and the chip amounts required to sum up to that the value.
Algorithm: The ChipBank's balance field is accessed the chip_amounts function is called to ascertain the number of chips required to sum up to the given balance. After this point, a String object is constructed to summarize the information and is returned to the caller.

3. Implementation and Testing

After deciding on the way in which to structure the constructors for each of the classes, we split up work on the methods and discussed our attempts to code the structures required. Ultimately, we decided to try to modularize some of the functional requirements by making helpers (particularly for the ChipBank class). Also, we realized we had approached our initial solution to the finding the chip amounts quite differently. Evan utilized a recursive methodology and Kyle utilized the integer division form. Ultimately, it was decided that the integer division form was more direct and efficient, so it was implemented in the lab. In addition to the given test cases, we also developed a few to test the capabilities of logging transactions for the ChipBank. Overall, we were able to pass all provided and self-developed test cases. Output of our running code can be seen in Figure 1. Also, the contents of banklog.txt after running the test cases can be seen in Figure 2.

```
econmang@econDebian:~/Documents/CS/SchoolFiles/cs356/hw/labs/lab_4$ python tester.py
Card: 2 of Hearts, Value: 2
Card: 3 of Hearts, Value: 3
Card: 4 of Hearts, Value: 4
Card: 5 of Hearts, Value: 5
Card: 6 of Hearts, Value: 6
Card: 7 of Hearts, Value: 7
Card: 8 of Hearts, Value: 8
Card: 9 of Hearts, Value: 9
Card: Jack of Hearts, Value: 10
Card: Queen of Hearts, Value: 10
Card: King of Hearts, Value: 10
Card: Ace of Hearts, Value: 11
Card: 2 of Diamonds, Value: 2
Card: 3 of Diamonds, Value: 3
Card: 4 of Diamonds, Value: 4
Card: 5 of Diamonds, Value: 5
Card: 6 of Diamonds, Value: 6
Card: 7 of Diamonds, Value: 7
Card: 8 of Diamonds, Value: 8
Card: 9 of Diamonds, Value: 9
Card: Jack of Diamonds, Value: 10
Card: Queen of Diamonds, Value: 10
Card: King of Diamonds, Value: 10
Card: Ace of Diamonds, Value: 11
Card: 2 of Diamonds, Value: 2
Card: 3 of Clubs, Value: 3
Card: 4 of Clubs, Value: 4
Card: 5 of Clubs, Value: 5
Card: 6 of Clubs, Value: 6
Card: 7 of Clubs, Value: 7
Card: 8 of Clubs, Value: 8
Card: 9 of Clubs, Value: 9
Card: Jack of Clubs, Value: 10
Card: Queen of Clubs, Value: 10
Card: King of Clubs, Value: 10
Card: Ace of Clubs, Value: 11
Card: 2 of Clubs, Value: 2
Card: 3 of Clubs, Value: 3
Card: 4 of Spades, Value: 4
Card: 5 of Spades, Value: 5
Card: 6 of Spades, Value: 6
Card: 7 of Spades, Value: 7
Card: 8 of Spades, Value: 8
Card: 9 of Spades, Value: 9
Card: Jack of Spades, Value: 10
Card: Queen of Spades, Value: 10
Card: King of Spades, Value: 10
Card: Ace of Spades, Value: 11
```

Figure 1: Running Code

Figure 2: Contents of banklog.txt

4. Reflection and Refactoring

Our solution initially counted chips using hard coded values and recursion to step through different chip values. In our first refactor, we decided to take a more modular approach that doesn't use any hard coded values. We initialized nested tuples to hold the chip colors and their values, then made a new function to initialize the chip amounts to 0 that uses the self._chip_values. We can then count chips by initializing the chip amounts then assigning each amount by integer division on the amount by the chip value. Before moving to the next value we use a modulo operation to reduce the amount. We repeat this for all chips. This allows the program to be more flexible and we can easily add new chip colors by editing the code in one spot.

In comparison, I felt that our solution was quite similar to the other teams, as well, with the main difference being our choice in iterators. For the future, we noted that we may have to change the initialized value for the face_up field of the Card class to False. This is due to the fact that, in the next lab, we are to be implementing a game of Blackjack, so the initial values of the cards of other players will need to be hidden when placed on the table. Further, we thought that we should structure the class files differently next time, grouping methods of similar purpose next to one another, or orienting them sectionally. For example, the getters/setters should be placed together and helper functions could be placed next to the expected calling functions, or should be grouped together in a helper functions section of the class.