Homework 5

MPCS 51042 – Python Programming

Due: November 10th 2020, 11:59 pm CT

Initial Setup

Make sure to perform a pull upstream inside your repository. This will grab the distribution code for hw5. The command is the following:

```
$ git pull upstream master
```

Style Guide

For this homework and all future homework assignments, we will follow the style guide used by the undergraduate Python course. It's located here: https://classes.cs.uchicago.edu/archive/2020/fall/12100-1/style-guide/index.html

Problem 1: `ESArray` Class

In JavaScript, the Array datatype (somewhat equivalent to Python's `list`) has a number of methods which `list` in Python does not have. For this problem, you are to write a subclass of `list` that provides some of these methods.

Place this solution inside `hw5/esarray/esarray.py`

Specifications

- The class should be named `ESArray` and should inherit from the built-in `list` class.

  Note: You are primarily extending the implementation of the `list` class. By inheriting from `list`, you have access to all its methods (including its `__init__` method) and `ESArray` is a `list`. This is very important to understand in order for you to implement this class.

- The `join` method should accept a string, `s` as its only argument (other than `self`) and return a string that results from joining each item in the list by the string `s`.
- The `every` method tests whether all items in the list pass a test implemented by a provided function.
- The `for_each` method executes a provided function function once for each item in the list.
- The `flatten` method returns a new `ESArray` with all list-like items in the original list flattened. That is, calling `flatten()` on a nested list returns a single list with all items appearing the their original order but with no nesting.

Example Interaction

```
In [1]: from esarray import ESArray

In [2]: x = ESArray([1, -3, 10, 5])

In [3]:  x.join('**')
Out[3]: '1**-3**10**5'

In [4]: x.every(lambda v: v > 0)
Out[4]: False

In [5]: x.every(lambda v: isinstance(v, int))
Out[5]: True

In [6]: x.for_each(print)
1
-3
10
5

In [7]: y = ESArray([[3, 4], [5], 6, [7, [8, [9, 10]]]])

In [8]: y.flatten()
Out[8]: [3, 4, 5, 6, 7, 8, 9, 10]
```

Problem 2: Blackjack Game

For Problem 2, you will gain more practice building classes in Python and also give you the opportunity to build an actual application on your own. However, you will still need to implement a few required classes and methods.

In this application, you will build a simple version of the card game, Blackjack with a `tkinter` GUI. The classes in the game will include: a `Card`, a `Deck`, a `Hand` and a `Game`. On your own (without my guidance), you will determine how to put these classes together to build the actual game.

Simplified Blackjack Rules

The rules for Blackjack can be quite extensive (https://en.wikipedia.org/wiki/Blackjack); however, for this assignment, you are required to implement a simplified rule set.

There will be only two players in the game: the player and the dealer. In our version of Blackjack, The value of cards two through ten is their pip value (2 through 10). Face cards (Jack, Queen, and King) are all worth ten. An ace is worth 11 points. Play should use the following process:

1. The player and the dealer should each receive two cards dealt from a deck.
2. The player should play their entire turn before the dealer.
    - The player is shown the value of their hand (the sum of the card values) and choose whether to take a card (hit) or stop at the current value (stand).
    - If the player chooses to hit, then if the new value of their hand is greater than 21, the player loses (busts). If the value is less than 21, the player repeats the process until they stand or bust.
3. If the player did not bust, the dealer plays.
    - The dealer will continue to take a card (hit) until the value of their hand is 17 or greater.

4. If player and dealer both avoided a bust, then the hand with the highest value wins the game. If both hands are of equal value, then the hand is a tie (push).

Classes Requirements

You are **required** to implement the following classes and their associated methods and properties. However, I am not specifying all the attributes you need to declare or how exactly each method is implemented. You need to determine the appropriate implementations and use **all required implementations** in your solution. You may define additional methods and import other modules with no restrictions.

Card class

Implement a `Card` class, which should hold all information the game cards. A card can be one of four suits (i.e., Clubs, Diamonds, Spades, or Hearts). In Blackjack, we do not care about the suit for a card; however, we need to specify the suit in order to load images into the GUI. A card also holds a value between 2-11. The `Card` class should have the following attributes/methods/properties:

- Define the following constant class attributes:

```
CLUBS = "clubs"
DIAMONDS = "diamonds"
HEARTS = "hearts"
SPADES = "spades"
```

- Define a class attribute called `card_images`. This class attribute will hold a collection of `tk.PhotoImage` objects. You will load this data structure inside the class method `load_images`. The type of this data structure is up to you to decide. Come back to this definition when you have a good understanding on what data structure you want to use.
- Define an `__init__(suit,value)` method. The constructor must take in a suit which is either: `Card.CLUBS`, `Card.DIAMONDS`, `Card.HEARTS`, or `Card.SPADES`. It also takes in a `value` between 2-11. Please note that 11 represents an Ace. You should store these inside instance attributes. You may add additional arguments to help with loading the card image.
- Define a class method called `load_images`. It takes no arguments. This method will load all images inside the `hw5/blackjack/images` directory as `tk.PhotoImage` objects. You will place these objects inside your `card_images` data structure. `card_images` will have 52 `tk.PhotoImage` objects.
- Define an `@property` named `value`. This a getter property that returns the card value. This property is only a getter and should not have a setter.
- Define an `@property` named `image`. This returns the `tk.PhotoImage` object from the `card_images` data structure that represents this Card's instance. For example, if `card = Card(Card.CLUBS,2)` then `card.image` returns the `tk.PhotoImage` object from `card_images` that is the image from the file `2_of_clubs.gif`. **Note**: An Ace is stored as "1" in the image file name.

Place this solution inside `hw5/blackjack/card.py`.

Hand class

Implement a `Hand` class, which should hold a collection of cards. The `Hand` class must have at least the following attributes/methods/properties:

- Define an `__init__()` method. It takes in no arguments but it must define an attribute that will represent your collection of cards. You must decide the appropriate type for this attribute.
- Define a `reset()` method. It takes in no arguments and clears your collection of cards.

- Define a `add(card)` method. It takes in a `Card` object and adds the card to the collection of cards.
- Define a `@property` called `total`. This method returns the sum of the values of the cards in the hand.
- Define a `draw(self, canvas, start_x, start_y, canvas_width, canvas_height)` method. This method draws the hand of cards on to the `canvas` starting at the location specified (i.e., `start_x` and `start_y`). Draw the cards horizontally along the x-axis. The method takes in the `canvas_width` and `canvas_height` but you may not need to use them in your implementation. Make sure to add a small offset between the cards (e.g. about 5 pixels). You will want to import the `import tkinter as tk` package and use the `canvas.create_image(...)` method to draw the card images on to the canvas.

Place this solution inside `hw5/blackjack/hand.py`.

Deck class

Implement a `Deck` class, which should hold a collection of cards and be able to shuffle and deal the cards. The `Deck` class must have at least the following attributes/methods/properties:

- Define an `__init__()` method. It takes in no arguments but it must define an attribute(s) that will represent the deck of cards. The constructor builds a deck of 52 cards. Each of the 4 suits must have cards with values 2-9 and 11 (Ace card), and 4 cards with the value 10 (i.e. 10 card, Jack, Queen, and King). This means each suit has a 13 cards. You should also keep track of all the cards you have dealt already.
- Define a `deal()` method. This method removes the top card on the deck and returns it.
- Define a `@property` called `size`. This method returns the number of cards left in the deck.
- Define a `shuffle` method. This method randomly shuffles all the already dealt cards and places them at the bottom of the deck. You can use the `shuffle` method from `random` to help with implementing this method.

Place this solution inside `hw5/blackjack/deck.py`.

The GUI

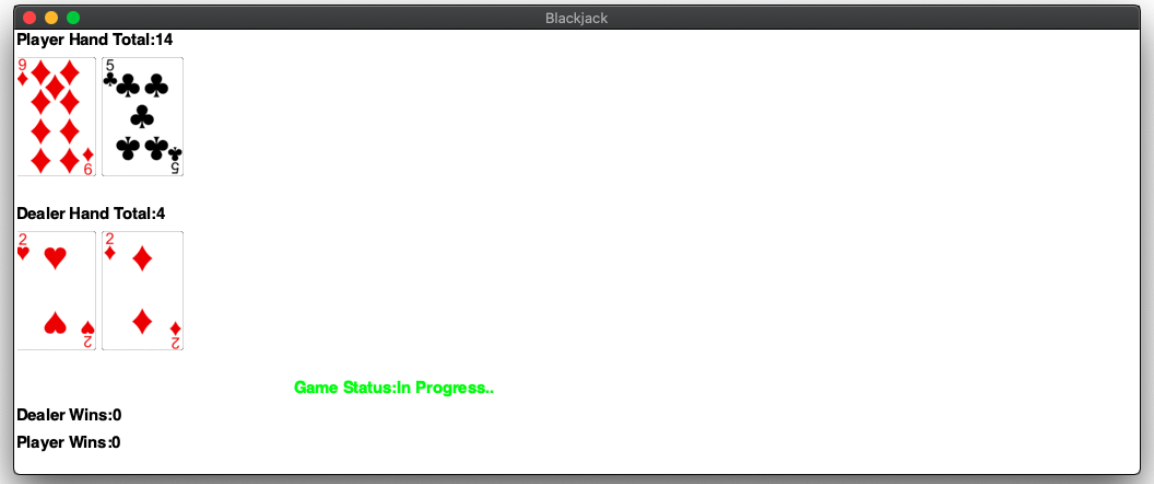
This application is your first exposure to implementing a simple GUI application. The Blackjack game will have a player (i.e., the user) and dealer (the "computer"). The user will interact with the GUI by pressing the following keys:

- `"h"`: The user wants to hit on their hand.
- `"s"`: The user wants to stand on their hand.
- `"r"`: The user wants to start a new game (i.e., reset the game.).

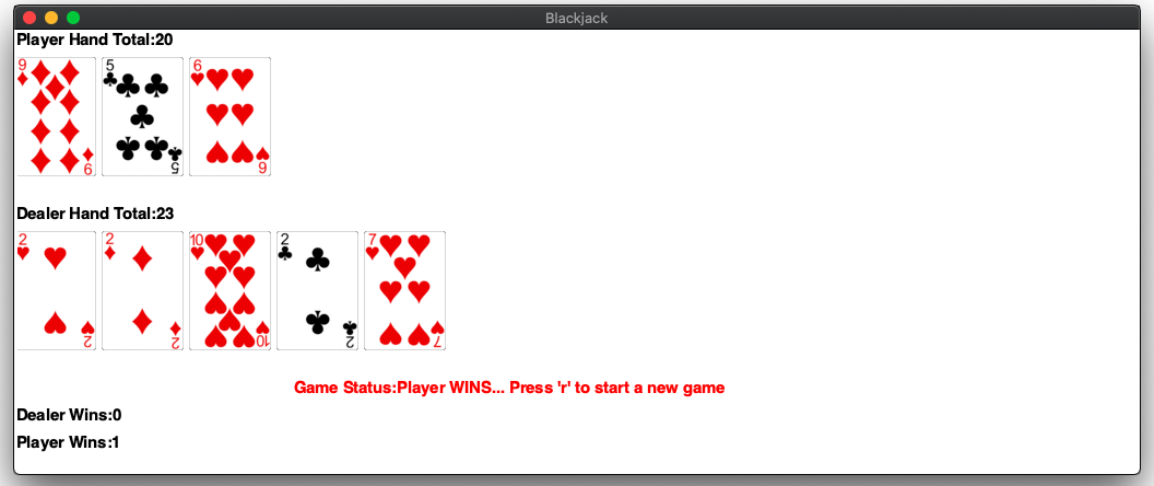
The application will look fairly basic and is not required to look fancy in any way. However, you are required to have the following components:

1. Text showing the player's total hand count.
2. Text showing the dealer's total hand count.
3. A game status text indicating the current state of the game.
4. The number of times the player has won.
5. The number of times the dealer has won.

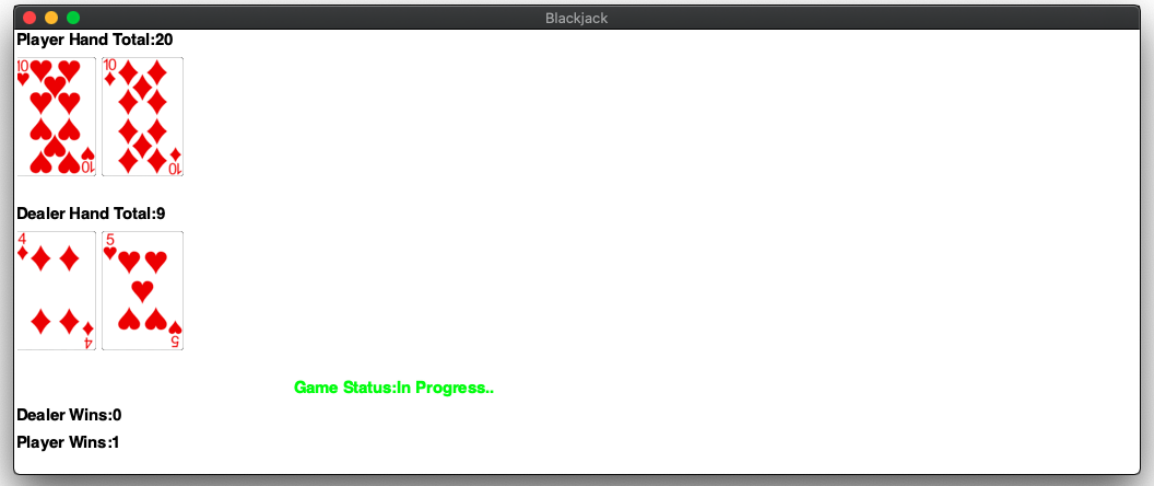
Here's an image of what my game looks like when we start the application in the terminal (i.e., `ipython blackjack.py`)



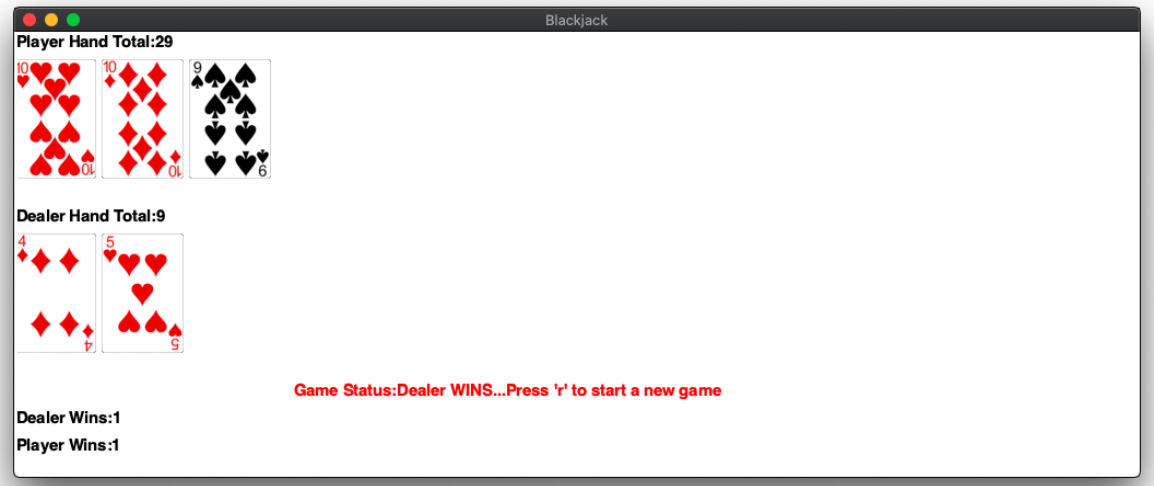
This image was after I hit ("h") and then I decided to stand ("s") because my score was at 20. After standing, the dealer draws their cards up until their total hand is 17 or more. I won the game because the dealer busted on their last card. Notice my wins went up by one.



This image was after I reset "r" the the game (i.e., start a new game of Blackjack).



This image was after I hit "h" and then I busted. The dealer won and notice their wins went up by one.



Here are some additional `tKinter` information you should know about (also look over the discussion session video, where I describe `tKinter`):

- You can create text using the `canvas.create_text`.

```
game_status = "In Progress..." # the text keyword argument sets the text
text_color = "green" # Use fill to set the text color to green or "red" to set it to red
canvas.create_text(x_loc,y_loc, fill=text_color, font=playerFont,
                   text=f'Game Status:{game_status}')
```

- Use `canvas.delete(tk.ALL)` to clear the entire canvas. I recommend that each time you update the GUI you clear the screen and redraw everything. It will make the implementation much easier. For example,

```
...
# Time to update the GUI
canvas.delete(tk.ALL)
canvas.create_text(...,text=f'Player Hand Total:{...}')
## ... Draw player hand ...
canvas.create_text(...,text=f'Dealer Hand Total:{...}')
## ... Draw the Dealer hand ...
#etc
...
```

- You can change the font (if you wish) by using the following code:

```
...
my_font = font.Font(family='Helvetica', size=15, weight='bold')
canvas.create_text(...,font=my_font, text=f'Dealer Hand Total:{...}')
...
```

The `blacjack.py` file represents the driver for the game. It already contains code for initially setting up the `tkinter` and also defines a `GameGUI` abstract base class that further initializes the `tkinter` window for the Blackjack game. **You cannot modify the `GameGUI` class at all.**

The Blackjack Class

Implement a `BlackJack` class that inherits from `GameGUI`, which implements a simple version of the card game and displays the game state to the player. The `BlackJack` class must have at least the following attributes/methods/properties:

- Define an `__init__(window)` method. It must initialize the `GameGUI` superclass with the `window` argument. After initializing, its up to you to determine the attributes for the class. Make sure to use the classes you defined above and to call `Card.load_images()` to load in the game images. However, you must keep track of the following:
  - The number of times the Player has won.
  - The number of times the Dealer has won.
  - The game status. The status text in the GUI should should indicate:
    - `"In Progress..."`: if a game is in session.
    - `"Player WINS... Press 'r' to start a new game"`: if the player wins the game.
    - `"Dealer WINS... Press 'r' to start a new game"`: if the dealers wins the game.
    - `"TIE Game...Press 'r' to start a new game"`: if the game is a tie.

    The color of the text should be green while the game is in progress or red if someone has won.
- Define a `reset()` method. This method should restart the game. This method is automatically called whenever the user hits the "r" key. Think of this method as the one that gets called when a new game wants to be played. This means you must reinitialize and update the GUI based on the starting of a new game.
- Define a `player_hit()` method. This method is automatically called whenever the user hits the "h" key. The user is requesting to perform a hit on their hand. This means that the method must draw a card from the deck and add it to the player's hand. After adding the card to the user's hand:
  1. If the player's hand is over 21 (i.e., a bust) then the player loses. The GUI should indicate that that dealer has won and increment the dealer's win amount. A new game only starts when the user hits "r"; therefore, if the user keeps hitting "h" or "s" then game and GUI remains unchanged.

2. If the player's hand is under 21, then the game needs to update the GUI. The dealt card should be shown and the player's hand total should be updated.

- Define a `player_stand()` method. This method is automatically called whenever the user hits the "s" key. The user is requesting to perform a stand on their hand. This means that the method must continuously add cards to the dealer's hand until their hand is greater than or equal to 17. Now, the game needs to decide who wins:
  - The dealer loses if their hand is over 21 (i.e., a bust). The GUI should indicate that the player has won and increments the player's wins.
  - The player wins if it has a higher hand than the dealer. The GUI should indicate that the player has won and increments the player's wins.
  - The dealer wins if if has a higher hand than the player. The GUI should indicate that the dealer has won and increments the dealers's wins.
  - If both hands are of equal value, then the hand is a tie (push).The GUI should indicate that game was a tie. No ones wins are incremented.

  A new game only starts when the user hits "r"; therefore, if the user keeps hitting "h" or "s" then game and GUI remains unchanged.

- The deck should not be reset after each game. When the deck has **13** cards remaining then you must **shuffle** the deck.