Project 2: How does it rank?

Overview

In this project, you will continue with modeling the game of poker. In the previous project, you identified and counted the occurrences of different kinds of poker hands. In this project, you'll take it one step further and directly compare hands. You'll also learn about and use Object-Oriented Programming (OOP).

Objectives

- Learn to program using OOP.
- Get more experience with refactoring
- Get more practice modularizing and using helper functions.
- Get more practice choosing good variable and function names.
- Practice testing and debugging.

Ranking the hands

The kinds of hands we are dealing with are the same ones we had in the previous project. Here they are again, listed in order from most valuable to least valuable (see cardplayer.com for the definitions):

1. flush
2. two pair
3. pair
4. high card (all other hands)

So, a flush is better than 2 pair, which is better than a pair, etc... However, in a real game of poker, two players might have the same kind of hand. In those cases, we need to decide who has the best hand (or decide that there's a tie). The ranks of the cards determine the relative worth of two hands *of the same kind*. So, for example:

- A pair of aces is better than a pair of fours, because aces rank higher than fours.
- A pair of 9s and a pair of 4s is better than a pair of 9s and a pair of 3s (4s outrank 3s)
- A pair of fives and a pair of eights is better than a pair of sevens and a pair of sixes (the highest pair (eights) outranks the highest pair in the other hand (sevens)).
- A pair of 10s and a pair of 2s with a King high card outranks a pair of 10s and a pair of 2s with a 4 high card.

- A 9-high flush outranks a 6-high flush.
- A King-Queen-high flush outranks a King-Jack-high flush.

The Mission

In this project, you will make a simple game that repeatedly:

1. Draws two new hands from a deck,
2. Shows the hands to the player, asking them which is worth more (or if they have the same value),
3. If the player was correct, they get one point and can continue.
4. If the player is incorrect, the game is over and the total score should be indicated.
5. The game is also over if there are not enough cards left to play another round.

Project Requirements

OOP

In an OOP program, we create some objects and then do things with those objects. In order to create objects, you first need classes that specify the properties and behavior of the objects you will create.

You should **refactor** the part of your Project 1 code that models cards, decks, and hands so that it uses classes and objects. You **must have** the following classes (you can have others as well, but you must have these):

- **Card** (in module **card.py**), an instance of which models a single card. You should have:
    - A non-default constructor which takes a rank (int from 2 through 14) and suit (String "Diamonds", "Clubs", "Hearts", or "Spades") as parameters, ***in that order***
    - Getter methods for getting the rank and suit
    - Other methods???
- **Deck** (in module **deck.py**), an instance of which models a single deck of cards. You should have:
    - A constructor
    - A method to deal a single card (i.e. remove a card from the deck and return it)
    - A method to shuffle the deck
    - Other methods???
- **PokerHand** (in module **poker_hand.py**), an instance of which models a 5-card hand of cards. You should have:

- the following non-default constructor:
  `__init__(self, card_list)`
  where `card_list` is a list of Cards (i.e. a list of Card objects) that should be in the new-constructed hand. ***Be sure to copy this list instead of just storing a reference to it.***
- A method to add a card to a hand
- `compare_to`, which compares this hand to another PokerHand. Here's the required signature:

```
def compare_to(self, other):
    """
    Determines how this hand compares to another hand, returns
    positive, negative, or zero depending on the comparison.

    :param self: The first hand to compare
    :param other: The second hand to compare
    :return: a negative number if self is worth LESS than other, zero
    if they are worth the SAME, and a positive number if self is
worth
    MORE than other
    """
```

- Other methods???

Refactor your main function (in the **main.py** module) so that it uses **Card**, **Deck**, and **PokerHand** objects to play the game described above.

Required refactoring

In addition to satisfying the program requirements, you must also refactor your code from Project 1 for use in Project 2 (make a copy into Project 2 so you can always go back to the version you had in Project 1).

If your Project 1 did not effectively use information hiding to divide the program into separate python files, you should **refactor** so it does.

Any poor variable and method/function names should also be fixed.

Unit Testing

Your **compare_to** method must be thoroughly unit tested, using the approaches we've discussed in class and lab. Recall that a test case is:

- Some code to run (i.e. **compare_to**)
- Some parameters to pass it (i.e. two hands)
- The expected result (i.e. is hand_a <, >, or == to hand_b)

Leave test functions in your code so that I can see them and run them (make sure they don't get run when I import your modules -- i.e. use `if __name__ == '__main__'`...

Your test functions should print information about the test being conducted along with "PASS" or "FAIL" to indicate whether **compare_to** has computed the correct answer. You can use the test suite code on the course website.

Some guidance

- **THINK FIRST**. **compare_to** might sound like an easy method to write, but if you just jump right in without some careful planning, it will get out of control quickly. So, take **some time to plan**.
- Make **incremental** progress. Start with simple comparisons first, and then get more sophisticated. E.G. you could make it work for the cases where the hands are of different kinds before tackling the more difficult cases.
- Use python's naming conventions for 'private' methods, so you can make helper methods in one class that are only visible in that one module.
- Remember to make your methods and functions take parameters instead of having your functions access variables defined outside the methods and functions.
- **Avoid hacking**. Especially when you are stuck on a bug:
    - write out pseudocode on paper,
    - draw diagrams of what's going on in memory, and
    - trace through examples by hand.
- Don't just make rapid changes to your code hoping that something will fix the problem. In other words, *program on purpose*.
- **CONTINUE** following the good habits I've been stressing in class:
    - **Use an iterative approach** of *code a little, test a little*.
    - Start with the things you know how to do, and build from there.
    - Be willing to refactor. And start early enough to have time for refactoring.
    - Aim for a modular design. Make helper functions/methods and use them to make your code readable.
    - Use good names for variables and functions.

- Test as you go.

Turn it in

Turn in the project by submitting the python files (only) to gradescope. The autograder will test your `compare_to()` method and award up to 20 points. The remaining points will be determined by the quality of your code -- whether it is readable, maintainable, understandable, and following information hiding.