Project 1: That's odd

Overview

In this project, you will write a program to help someone figure out how likely certain kinds of hands are in (a simplified version of) the game of poker.

Objectives

- Practice modularizing, using helper functions
- Practice choosing good variable and function names
- Re-acquaint yourself with Python, including looping constructs
- Practice testing and debugging
- Learn about and practice information hiding
- Practice refactoring

Poker

You might have played a version of Poker, a card game in which players bet on whether their hands are better than their opponents' hands, without knowing for certain what cards their opponents hold. There are many variations of Poker, but in all of them players have or make **5-card hands**, and depending on the cards in the hand, the hand can be worth more or less (or the same) as another 5-card hand. See cardplayer.com for a list of possible hand kinds and their relative rankings.

Part of the game is understanding the odds that your hand is of higher value than your opponents' hands. A *royal flush*, for example, is worth more than a hand with *four of a kind* because it is more rare. But a four of a kind is worth more than a hand with a *pair* because the four of a kind is more rare. So, if you have a *high value* hand, there are fewer hands that can beat it and therefore if you have a four of a kind, it's a pretty good bet (but not a sure thing) that your opponent's hand is not worth more.

A simplification

In this project, we will deal with just a small subset of the possible Poker hands. We'll only deal with hands of the following kinds:

- Flush
- Two pair (note: since our version of the game doesn't have 4-of-a-kind, the two pairs might be of the same rank)

- Pair
- High card (i.e. anything else)

Project Requirements

Your program's goal is to determine how likely hands of the different kinds are. It should model a deck of cards that can be shuffled and from which cards can be dealt. Using that modeled deck your program should start by shuffling the deck and then:

1. repeatedly create poker hands by dealing cards from that deck.
    - if the deck is used up, gather up the cards and shuffle them again.
2. count the number of such hands that are of the different kinds (pair, two pair, flush).

Your program should produce output like the following (with the x's replaced with real values you compute):

```
# of hands     pairs    %     2 pairs     %      flushes    %      high card    %
    10,000        xx  xx.xx         xx  xx.xx          xx  xx.xx           xx  xx.xx
    20,000        xx  xx.xx         xx  xx.xx          xx  xx.xx           xx  xx.xx
    30,000       xxx  xx.xx        xxx  xx.xx         xxx  xx.xx          xxx  xx.xx
    40,000       xxx  xx.xx        xxx  xx.xx         xxx  xx.xx          xxx  xx.xx
    50,000      xxxx  xx.xx       xxxx  xx.xx        xxxx  xx.xx         xxxx  xx.xx
    60,000     xxxxx  xx.xx      xxxxx  xx.xx       xxxxx  xx.xx        xxxxx  xx.xx
    70,000     xxxxx  xx.xx      xxxxx  xx.xx       xxxxx  xx.xx        xxxxx  xx.xx
    80,000     xxxxx  xx.xx      xxxxx  xx.xx       xxxxx  xx.xx        xxxxx  xx.xx
    90,000     xxxxx  xx.xx      xxxxx  xx.xx       xxxxx  xx.xx        xxxxx  xx.xx
   100,000     xxxxx  xx.xx      xxxxx  xx.xx       xxxxx  xx.xx        xxxxx  xx.xx
```

Each row in the output represents a summary of analysis of a large group of hands. The first row summarizes 10,000 hands, while the last row summarizes 100,000 hands. Within each row, for each of the kinds of poker hands (pair, 2 pair, flush, high card), there are two columns in the output. The first column is the raw count of the number of the hands that matched that kind of hand; the second column is the percentage of the hands that match. For example, if 25,000 of 50,000 hands dealt are pair hands, the pairs columns would be 25000 and 50.00.

Some notes on the output format:

- To receive full credit, your output must match the sample output.
- The output will be checked by an autograder program, with strict requirements. If the following aren't completely adhered to, the autograder is likely to fail:
    - The column headings should be **exactly** as shown in the sample output.

    - The columns should be in **exactly** the same order as in the sample output.
    - Your program should give **no other output** than the row of headings and the the 10 rows of data.
    - There should be **at least 2** blank spaces between adjacent columns.
- The percentages should be displayed with exactly 2 digits before the decimal point and exactly 2 digits after the decimal point. If the percentage is less than 10 percent, you'll need a leading 0 (e.g. 02.53 instead of 2.53).
- The raw count columns should be **right aligned**. Note that you should **not** interpret my sample output as a hint to the number of pairs, 2 pairs, etc... you should find. So, just because I show xx in the first row, that doesn't mean that you should expect between 9 and 100 pairs, 2 pairs, etc... for 10,000 hands.
- The total number of hands analyzed should be displayed with a comma thousands separator. The other columns should not be displayed with a thousands separator.
- While the alignment must match my output, the column widths and space between columns does not need to match my output exactly, but try to get close.
- **Note**: see pyformat.info for help with formatting the output nicely without a lot of fancy calculations on your part.

You must have a **poker_sim.py** module, within which there is a **play_rounds()** function I can call to produce the desired output. The autograder will import your poker_sim module and call the play_rounds() function, which should cause your program to compute all the counts and print the table of output.

Let's look in a bit more detail at the requirements:

Model a deck of cards

You can model a deck of cards with a *list* of strings, where each string represents a single card. One possible representation is to use the first part of the string to represent the rank of the card, and the last part of the string to represent the suit of the card. You can use a **single character for the suit**, so it's easy to determine which part of the string represents the rank and which the suit. Note: you can represent cards in other ways if you'd prefer (e.g. a card could be represented as a list, a tuple, a dictionary).

You should have functions that create a deck of cards, shuffle a deck of cards, and deal from a deck of cards. A deck of cards is 52 cards, with ranks Ace through King and four suits: Clubs, Diamonds, Hearts, Spades. Note: Aces are more valuable than all other ranks, but that's not important for this project.

Create poker hands

A poker hand can be a list of cards (i.e. a list of the same kinds of strings your deck list has).

You should have functions to add cards to a hand.

Count the hand kinds

You'll need to repeatedly use the above-mentioned functions to create a deck, shuffle it, and deal hands. Once you have hands, you'll need to determine if they match the different hand kinds.

Some guidance

- **Use an iterative approach** of “*code a little, test a little”* like I've shown in class. Each of the functions you write should be tested in-the-small before you use it in the larger program.
- Start with the things you know how to do, and build from there. For example, if you don't yet know how you'll implement a deck of cards, you can start with figuring out a single hand of cards. Or if you don't know how to get the formatting to match the sample output, make an initial implementation where you don't try to match the output format exactly. Once you know you are getting the right answers, you can focus on figuring out the formatting.
- **Be willing to refactor**. Make a chunk of code into a function, change how you shuffle the deck, change your internal representation. **Start the project early** so you have time to make your solution better.
- Aim for a modular design. Make helper functions and use them to make your code readable.
- Use good names for variables and functions. You want someone who reads your code to know what it does without having to think too much. Also, use the accepted Python naming conventions, in particular this:
  Function names [and variable names] should be lowercase, with words separated by underscores as necessary to improve readability.
- Test as you go. Leave your test cases in the code.

Turning it in TWICE

This project has two due dates:

1. At the first due date, you should turn in a **working** project, which the autograder will run to see if it works. You’ll get feedback on the correctness of the program's output *without* regard for what the code looks like (I probably won't even look at your code). The correctness at this point will be worth 15% of the project grade.

2. For the second due date, **refactor** to make your working program better: more efficient, better organized, more correct, more readable, more maintainable, etc... In particular, you should make sure you are following ***information hiding***. This second submission will also be tested for correctness, but the vast majority of the available points will be for the quality of your code.

For both submissions, you'll submit by uploading your .py files to gradescope. **Note that gradescope allows multiple submissions** --- you can <u>**submit as many times as you want**</u> until the due date/time and the last submission in the one you'll be graded on.