Assignment 12

CS161

For each of the problems below be sure to design your solution before you write any Python code. Follow the software development stages:

1. Analyze the problem
2. Determine the specifications for your solution
3. Create a design for your solution in pseudocode
4. Implement your design in Python
5. Test/debug your solution

Each Python module you turn in must begin with a block, called a *docstring*, that looks like the example below (see PEP 257 for more details). The module docstring must follow the shebang and coding statements in your module; i.e. it must be the first actual line of code in the module.

In your module docstring show the work you did following the software development stages above. Make notes about the problem as you analyze it; write specifications for what your program must do to find a solution; write your program out in pseudocode *before* you start to write Python; and design a set of test data that you can use to prove your program is producing correct results. Only after you've done all of this should you try to translate your pseudocode to Python.

```
#! /usr/bin/env python3
# coding=utf-8

"""
Brief, prescriptive, one sentence description of what the module does.

A more detailed explanation of what the module does. Use complete sentences and
proper grammar. Treat this as if it is documentation someone else will be reading
(because it is). Include any notes from your problem analysis as well as your
program specifications here.

Pseudo code, to be written before any actual Python code.

Assignment or Lab #

Firstname Lastname
Firstname Lastname of partner (if applicable)
"""
<two blank lines>
<your code>
```

Every function or method you write must also include its own docstring. That header will also use triple quotes and begin with a prescriptive, one sentence description of what the function or method does. If the function or method takes arguments, they should be described as well as any non-obvious return value(s). See examples below or PEP 257 for more details).

```
def print_greeting():
    """Print a greeting to the user."""
    print(“Hello out there in userland!”)


def hypotenuse(side_a, side_b):
    """
    Calculates the length of the hypotenuse of a right triangle using
    the formula c^2 = a^2 + b^2.

    side_a – the length of side A of the triangle
    side_b – the length of side B of the triangle
    """
    c = a**2 + b**2
    return math.sqrt(c)
```

Pair Programming

In this assignment we are going to use **pair programming**.

> *Pair programming is an agile software development technique in which two programmers work together at one workstation. One, the **driver**, writes code while the other, the **observer** or **navigator**, reviews each line of code as it is typed in. The two programmers switch roles frequently.*
>
> *While reviewing, the observer also considers the "strategic" direction of the work, coming up with ideas for improvements and likely future problems to address. This is intended to free the driver to focus all of their attention on the "tactical" aspects of completing the current task, using the observer as a safety net and guide.*

Choose one partner in the lab to work with. You can pair program in one of two ways:

1. Two people can work at one computer, occasionally switching the driver and observer roles. It is *critical* that both members be engaged in the work. You should take turns being in the driver and observer roles with only the driver typing and the observer making comments or driving the direction of the code.
2. Two people can work at separate machines using Visual Studio Live Share. Live Share, available as a free plug-in for VS Code and already installed on the school’s Surface devices, enables you and your partner to collaborate on the same codebase without the need any difficult configuration. When you share a collaborative session, your partner sees the context of the workspace in their editor. This means they can read the code you shared without having to clone a repo or install any dependencies your code relies on. They can use rich language features to navigate within the code; not only just opening other files as text but using semantic analysis-based navigation like Go to Definition or Peek. Again, you should take turns being in the driver and observer roles with only the driver typing and the observer making comments or driving the direction of the code.

Assignment

For this assignment you *must* implement your own `Card`, `Deck`, `Hand`, `Game`, and `Simulation` classes. You *may not* use any pre-defined modules that implement the required functionality.

Part 1

Create a `Card` class that represents a single card from a standard deck of playing cards. Each card should know its suit (Clubs, Diamonds, Hearts, Spades) and its rank (2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace, and maybe Joker). Each `Card` must also have the following *magic methods* defined on it: `__init__`, `__str__`, `__eq__`, `__ne__`, `__lt__`, `__gt__`, `__le__`, and `__ge__`. The `__init__` method will take the `Card`'s suit and rank as arguments. The comparison methods need only compare cards based on their rank. Remember to include module- and method-level docstrings to document your code.

Save your code in a file named `card.py`.

Part 2

Using the `Card` class from Part 1, create `Deck` class. Each `Deck` will contain the 52 or 54 cards commonly found in a standard deck of cards (one of each rank for each suit). Each `Deck` musk include an `__init__` method to instantiate the Deck, a `__str__` method that lists all the cards currently in the deck in order, and a `shuffle` method that randomized the `Deck`. The `__init__` method should take a Boolean parameter, `jokers`, which indicates whether two jokers should be included in the `Deck` and which has a default value of `False`. Remember to include module- and method-level docstrings to document your code.

Save your code in a file named `deck.py`.

Part 3

Using the `Deck` class from Part 2, create a `Hand` class that acts as a container for some number of cards. The number of cards in a `Hand` is specified as an argument to the `Hand`'s `__init__` method, called `size`, which has a default value of 5. A `Hand` should always sort and store its cards in ascending order. The Hand class should implement a `__str__` magic method as well as the following additional methods:

- `is_royal_flush` which returns a Boolean indicating whether the `Hand` is a royal flush; i.e. it contains 10, Jack, Queen, King, Ace of the same suit.
- `is_four_of_a_kind` which returns a Boolean indicating whether the `Hand` is a four of a kind; i.e. it has four cards of the same rank.
- `is_full_house` which returns a Boolean indicating whether the `Hand` is a full house; i.e. it contains a three of a kind with a pair.

Remember to include module- and method-level docstrings to document your code.

Save your code in a file named `hand.py`.

Part 4

Using the `Deck` and `Hand` classes from above, implement a `Game` class that instantiates an instance of a `Deck`, with no jokers, and a `Hand` with five cards. In addition to an `__init__` method, the Game class should implement a method, `new_deal`, which returns all the cards in play to the Deck, shuffles it, and deals a new `Hand`. Remember to include module- and method-level docstrings to document your code.

Save your code in a file named `game.py`.

Part 5

Using the `Game` class from above, write a class called `Simulation` that will instantiate a `Game`. `Simulation` should then prompt the user for a type of hand to check for: royal flush, four of a kind, or full house. Run ten million Monte Carlo simulations to approximate the odds of being dealt the indicated type of hand. Prompt the user asking if they would like to run another round of simulations or if they would like to quit. Remember to include module- and method-level docstrings to document your code.

Save your code in a file named `simulation.py`.

Submission

Submit all your Python script files, and any other necessary files, in the appropriate folder in Google Drive as demonstrated by your instructor.