Week 5 Assignment – Head-to-Head Blackjack

Overview

In this assignment, you adapt your code from Week 4's assignment using an object-oriented approach to implement the logic, and enhance your code to include the rules of game play to see which strategies work against each other.

Please follow the steps outlined below.

Preparation

Copy **blackjack2.py** from Week 4 to a new file called **blackjack3.py**; also copy your test cases from **test_blackjack2.py** to **test_blackjack3.py**. Adjust **test_blackjack3.py** so it uses the code defined in **blackjack3.py**.

Step 1: Hand class

In **blackjack3.py** implement a class called **Hand** that encapsulates some of the logic developed in the previous two assignments. The **Hand** class represents a single hand of blackjack and should have these attributes:

- **cards** – A list of integers that represents the cards in the hand. This is similar to the cards list used in the previous assignments.
- **total** – An integer that represents the current score of the hand.
- **soft_ace_count** – An integer that represents the number of "soft aces" (i.e., aces that count as 11).

The **Hand** class should have at least the methods shown in the screenshot below:

```
class Hand:
    """Class that encapsulates a blackjack hand."""
    def __init__(self, cards=None):

    def __str__(self):

    def add_card(self):

    def is_blackjack(self):

    def is_bust(self):

    def score(self):
```

Consider these notes as you implement the Hand class:

- **__init__** – Constructor should initialize the attributes of the class. The default **cards** argument will be useful when testing, and it should be used to pass in a list of card integers. If it is not provided (i.e., it is **None**), the cards attribute of the class should be initialized with an empty list. If the cards list is passed in, the score method should be called so the **total** and **soft_ace_count** attributes are maintained properly.

doesthis add a card regarless:

- **__str__** – Should return a string representing the hand. Useful for debugging.
- **add_card** – Should add a randomly selected card (using the same technique as the last two assignments), and call the **score** method.
- **is_blackjack** – Should return true if the Hand represents "Blackjack" (i.e., a soft ace and a 10, Jack, Queen, or King).
- **is_bust** – Should return true if the Hand total is greater than 21.
- **score** – This method should utilize the logic developed in the previous assignment for scoring a hand (i.e., loop across the cards and compute the proper values for **total** and **soft_ace_count**).

Adapt the tests in **test_blackjack3.py** to test all the methods of Hand. Since the **add_card** method adds a random card, the default argument of the constructor can be used in the test cases to set up a Hand object with a specific set of cards, thereby making it possible to test the **score** method.

Step 2: Strategy class

In **blackjack3.py** implement a class called **Strategy** that has two attributes, **stand_on_value** and **stand_on_soft**. This class will encapsulate the logic from the previous assignments related to standing and playing a hand. The class should be structured as shown in the following screenshot:

```
class Strategy:
    def __init__(self, stand_on_value, stand_on_soft):

    def __repr__(self):

    def __str__(self):

    def stand(self, hand):

    def play(self):
```

Consider these notes as you implement the Strategy class:

- **__init__** – This method should initialize the attributes based on the values passed in.
- **__repr__** – This method should return the "canonical" string representation of the object.
- **__str__** – This method should return a shorthand string representation of the Strategy that starts with either "H" or "S" (depending on the value of **stand_on_soft**) and the stand-on value. For example, a "stand on hard 16" strategy would be represented by "H16".
- **stand** – This method encapsulates the logic previously developed to determine if the strategy requires "hit" (returns False) or "stand" (returns True) for the Hand object that is passed in.
- **play** – This method should play a single hand of blackjack by instantiating a Hand object and using the methods of Hand and the **stand** method of the strategy object to play out the hand. This method should return the Hand object that was created.

Adjust the test cases in **test_blackjack3.py** as necessary to test **Strategy.stand** and make sure all existing test cases pass.

Step 3: Refactor main

In **blackjack3.py** refactor the functionality of the main method as described below. The program should still take a single command line argument that is the number of simulations to run per strategy.

```
> python3 blackjack3.py 100000
```

In this version of the program, the logic should be implemented to represent both a *player* hand and a *dealer* hand. The overall output of the program is shown in the table below, where the percentage of scenarios won by the *player* is contained in the table cells.

The rules of blackjack dictate the following:

- The *player* goes first and plays out their hand according to a strategy. If they go bust, they lose immediately without the *dealer* hand playing.
- The *dealer* goes second and plays out their hand according to a strategy. If they go bust, the *player* wins.
- If either the *dealer* or *player* has blackjack and the other does not, the one with blackjack wins.
- If the *player* and *dealer* have the same score, it is a tie (called a "push" in blackjack lingo); then the scenario should not be counted as a win or loss for either player (basically it is thrown out).
- Otherwise, the *player* or *dealer* with the higher total wins.

Your code should loop across all strategies from stand on 13 to stand on 20 with both hard and soft options, for both the *player* and *dealer* and report the percentage of non-tie wins the *player* has. Your code should produce output using the **csv.writer** object that corresponds to table shown below. The rows correspond to the different *player* strategies (stand on hard 13, stand on soft 13, etc.). The columns correspond to the different *dealer* strategies. Your code should not have zeros for all the cells, and values should be formatted to have digits after the decimal point.

P-Strategy	D-H13	D-S13	D-H14	D-S14	D-H15	D-S15	D-H16	D-S16	D-H17	D-S17	D-H18	D-S18	D-H19	D-S19	D-H20	D-S20
P-H13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-S13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-H14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-S14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-H15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-S15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-H16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-S16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-H17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-S17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-H18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-S18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-H19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-S19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-H20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P-S20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Upload

Please combine **blackjack3.py** and **test_blackjack3.py** into a single ZIP file.