CS5710 – Coursework

February 9, 2024

You will implement a simplified version of a popular casino game Three Card Poker (see `https://en.wikipedia.org/wiki/Three_Card_Poker`). Your implementation will rely on a full set of the Python language features including object-oriented programming, and mutable compound data types (such as lists and dictionaries). It will also involve implementing moderately complex control flows to support various parts of the game logic.

Before you start

1. This assignment is a **summative** coursework.
2. Your submissions will be both graded, and receive a feedback.
3. Collaboration is **NOT** permitted. In particular, you should **neither** copy any code from other students or online sources (plagiarism), **nor** let other students see any parts of your solution (collusion).
4. All submissions will be checked for plagiarism and collusion against both the other student submissions and the online sources.
5. There are five questions. All questions are mandatory, and have percentage weights specified next to them.
6. You will submit three files as per the instructions in the "What to submit" section at the end of the brief.
7. Submission **deadline** is **10:00am, 21 February 2024**.
8. Submit your solutions on Moodle by clicking on Coursework 1 under the "Assignments" heading.
9. Your submissions will be checked by automated testing scripts.

10. Make sure you carefully follow the instructions. Do **NOT** leave any testing print-outs or calls to `input` (except in the `if __name__ == '__main__':` sections) as they may cause the automated tester to fail.

11. You must use the functions from the Python's `random` module whenever you need to generate a random quantity or perform an operation that depends on randomness (such as the card deck shuffle).

12. Do **NOT** include any calls to `random.seed()` as this will cause your programs to fail all the tests.

Question 1: Implementing Basic Classes (40%)

Download a Python source file `Card.py` from Moodle. The file contains incomplete implementations of the following three classes representing entities found in any card game:

- `Card`: implements an abstraction of a playing card using two data attributes: `rank` and `suit`. The rank is an integer between `0` and `12`, and is mapped to a string representation using the `ranks` list. The suit is an integer between `0` and `3`, and is mapped to a string via the `suits` list. Note that both `ranks` and `suits` are *class variables*, which means they are available to all instances of the `Card` class as `Card.ranks` and `Card.suits` (see the code of the `__str__()` method for an example).

- `Deck`: implements an abstraction of the standard 52 (13 ranks $\times$ 4 suits) deck of playing cards. The deck is represented as a list `cards` of `Card` objects. It includes the implementation of the constructor (the `__init__()` method), the `__str__()` method, and a number of place-holders for other methods whose functionality is described via their doc strings.

- `Hand`: implements an abstraction of a hand in a card game. The `Hand` class is a subclass of `Deck`, which means that it inherits its `cards` attribute, and all its methods. It further refines the `Deck` class by adding the `name` attribute, which is a string holding the name of the person to whom the hand was dealt.

What to do: Replace the `pass` place-holders in the bodies of the following methods with the code implementing the functionality described in their docstrings:

- `Card` class:
  - `get_suit()`, a getter method for the `suit` attribute of the `Card` class. Hint: use the code of the `get_rank()` method as an example.
- `Deck` class:
  - `shuffle()`: randomly shuffles the list stored in `self.cards` in place using the `random.shuffle()` method of the Python's `random` library.
  - `pop_cards()`: takes a positive integer `n` as argument and removes the last `n` Card objects from the end of the `self.cards` list. Returns a new list instance comprised of the removed `Card` objects.
  - `add_cards()`: takes a list `cards` of `Card` objects as argument, and extends `self.cards` with a deep copy of the `cards` parameter. You can use the `copy.deepcopy()` method of the Python's `copy` library to generate a deep copy of a list. Note that we need a deep rather than a shallow copy since the list elements are by themselves mutable objects.
  - `clear_cards()`: clears the content of `self.cards`. Be careful to **NOT** remove the entire `self.cards` attribute, which will happen e.g., if you use `del self.cards`.
  - `move_cards()`: takes an instance `hand` of the `Hand` object and an integer `n` as arguments, and uses the `add_cards()` and `pop_cards()` methods to move n cards from `self` to `hand`.
- `Hand` class:
  - `get_name()`: a getter method for the `name` attribute of the `Hand` class.

Testing (not assessed): Add code to the `if __name__ == '__main__:'` section of the `Card.py` to test your implementation. You will find that some testing code is already there. Use the provided testing code as an example to create your own test cases.

Question 2: Ranking a Three-Card Poker Hand (20%)

Download a Python source `PokerHand.py` from Moodle. The file contains incomplete implementations of the following two classes representing a deck and a hand in the Three-Card Poker game:

- `ThreeCardPokerDeck`: implements an abstraction of a Three-Card Poker Deck. It is a subclass of the `Deck` class, which means that it inherits all of its data and method attributes, such as `cards`, `shuffle()`, etc. In addition, it implements a method `deal_hand()` that takes the hand name as optional argument, deals three cards from `self.cards`, and returns a new `Hand` object initialized with the cards that have been dealt.

- `ThreeCardPokerHand`: implements an abstraction of a Three-Card Poker hand. It is a subclass of `Hand`, and thus, inherits all attributes from both `Deck` and `Hand`. The provided implementation of the constructor `__init__()` takes a list of cards and an optional hand name as arguments, calls the constructor of the `Hand` class to initialize the parent object, and initializes the `cards` attribute with a copy of the card list argument.

  Since, as is discussed below, the Poker hand ranking is computed by considering the card ranks and the card suits in isolation, the code in the constructor extracts them into two separate list attributes: `ranks` and `suits`. The `ranks` list is then sorted in a reverse order of ranks to facilitate the rank comparison for Question 3. Finally, the `_compute_rank()` method is invoked to initialize the hand ranking.

What to do: Write a code for the method `_compute_rank()` replacing the `pass` keyword. The method should use the lists `self.ranks` (the card ranks sorted in the reverse order), and `self.suits` (the card suits) created by the constructor to compute the hand ranking. The result must be an integer between 0 (the lowest) and 5 (the highest). It must be stored in the `self.rank` attribute, and computed according to the rules below:

1. **Straight Flush** (`self.rank = 5`): either three cards of sequential rank, or Ace, 2, and 3, all of the same suit.

2. **Three of a Kind** (`self.rank = 4`): three cards of the same rank.

3. **Straight** (`self.rank = 3`): either three cards of sequential rank or Ace, 2, and 3, **not** all of the same suit.

4. **Flush** (`self.rank = 2`): three cards of the same suit, not sequential rank.

5. **Pair** (`self.rank = 1`): two cards of the same rank, no other matches.

6. **Nothing** (`self.rank = 0`): everything else. The convention is to refer to a Nothing hand by the rank of its highest card. For example, a Nothing hand in which 10 of Spades is the highest ranked card will be called "10-High". This is reflected in the implementation of the provided `__str__()` method.

Testing (not assessed): Add code to the `if __name__ == '__main__:'` section of the `ThreeCardPokerHand.py` to test your implementation. Deal a few Three-Card Poker hands as shown in the provided testing code, and print them out to reveal their content and the rank. You can also access the value of the rank attribute directly (not recommended), or via the provided getter method `get_rank()`. Try a few sample hands of your own to make sure the rank is computed correctly.

Question 3: Comparing the Hands (20%)

Below are the rules for comparing two hands in the game of Three-Hand Poker:

1. If the hand rankings are not equal, then the hand with a higher ranking wins.

2. If the hand rankings are the same, then the following tie-breaking procedure is applied:

   (a) If the hands are either both Straight Flush or both Straight, then the highest rank of any card in one hand is compared to the highest rank of any card in the other hand. For the Straight Flush hands that include an Ace, the highest rank is 3 (not Ace!). Otherwise, the highest rank is the rank of a card with the highest rank. If the highest ranks are distinct, then the hand with a higher ranked card wins. Otherwise, it is a tie.

Examples: "9 of Diamonds, 10 of Diamonds, Jack of Diamonds" beats "7 of Spades, 8 of Spades, 9 of Spades", "Queen of Hearts, King of Diamonds, Ace of Clubs" beats "Ace of Hearts, 2 of Diamonds, 3 of Spades", and "7 of Spades, 8 of Spades, 9 of Spades" and "7 of Clubs, 8 of Clubs, 9 of Clubs" is a tie.

(b) If the hands are both Three of a Kind, then the rank of an arbitrary card in one hand is compared to the rank of an arbitrary card in the other hand, and the hand with a higher ranked card wins. Note that a tie is impossible in this case.

(c) If the hands are either both Flush or both Nothing, then each hand is sorted in the reverse order of their ranks, and the outcome is computed by comparing the resulting rank triples to each other in the lexicographical order. That is, the highest ranks in both hands are first compared to each other, and if they are distinct, then the hand with a higher ranked card wins. Otherwise, the second highest ranks are compared to each other, and so force. If all respective ranks in both hands are equal, it is a tie.

Examples: "Ace of Diamonds", "10 of Diamonds", "2 of Diamonds" beats "King of Spades", "Jack of Spades", "10 of Spades".

(d) If both hands are Pairs, then the respective ranks of the paired cards are first compared to each other. If they are distinct, then the hand with a higher ranked card wins. Otherwise, the outcome is determined by comparing the ranks of the highest ranked cards in both hands.

Examples: "3 of Diamonds, 3 of Clubs, 2 of Spades" beats "2 of Clubs, 2 of Hearts, Ace of Spades", "2 of Clubs, 2 of Hearts, Ace of Spades" beats "2 of Diamonds, 2 of Spades, King of Hearts", "3 of Diamonds, 3 of Clubs, 2 of Spades" and "3 of Clubs, 3 of Spades, 2 of Hearts" is a tie.

What to do:

1. Write a code for the method `_compare()` of the class `ThreeCardPokerHand` found in `PokerHand.py`. The method takes another `Hand` object `other` as argument, and returns `-1`, `0`, and `1` if, according to the rules above, `other` is a winning hand, `self` and `other` are tied, and `self` is a winning hand respectively.

2. Use the `_compare()` method to write a code for the following three special methods: `__gt__()` ("greater than"), `__ge__()` ("greater or

equal than"), `__eq__()` ("equal"), `__ne__()` ("not equal"). Use the provided implementations of the `__lt__()` ("less than") and `__le__()` ("less or equal than") methods as examples. The resulting set of six comparison methods will allow the hands to be compared directly by using the standard comparison operators `<`, `<=`, `>`, `>=`, `==`, and `!=`.

Testing (not assessed): Add a code to the `if __name__ == '__main__':` section of the `ThreeCardPokerHand.py` to test your implementation.

Question 4: Estimating Hand Probabilities (10%)

You will write a code to estimate the probabilities of the six possible hand rankings in the Three-Card Poker game discussed in Question 2 using a Monte-Carlo simulation.

What to do: Download a file `Poker.py` from Moodle. Write a code for the function `make_dist()` that takes a number of trials n as argument, generates n random Three-Card Poker hands, and returns a Python dictionary mapping a string label representing the rank of each generated hand to its probability expressed as a *float percentage*. The possible hand labels are stored in the `all_labels` class variable of `ThreeCardPokerHand`:

```
all_labels = ['Nothing', 'Pair', 'Flush', 'Straight', 'Three of a Kind'
                  'Straight Flush']
```

You can use the provided `get_label()` method of `ThreeCardPokerHand` to retrieve the label for a given hand. For example, the probabilities generated by the model solution after 10000 trials were as follows:

```
{'Pair': 16.34, 'Nothing': 74.82, 'Flush': 5.19,
'Straight': 3.11, 'Straight Flush': 0.31, 'Three of a Kind': 0.23}
```

These are roughly consistent with the analytical probabilities listed in `https://en.wikipedia.org/wiki/Three_Card_Poker`.

Question 5: Playing the Game (10%)

You will implement a simplified variant of Three-Card Poker in which a single player is playing against the dealer, and no bonus bets or bonus payoffs are used. The game proceeds in a series of *rounds*. At the beginning of

each round the player and the dealer are dealt three card each face-down. The player then places an *ante* bet without looking into his/her cards. After placing the ante bet, the player consults his/her hand, and decides whether to *play* or *fold*.

If the player folds, the ante bet is forfeited, and the game proceeds to the next round

If the player plays, he/she places an additional *play bet*, which is equal to the ante bet. Both hands are then revealed. The dealer's hand must be at least *Queen-High* to qualify for playing. If the dealer's hand does not qualify, the player is paid 1:1 on the ante bet, and nothing on the play bet, which is then returned to the player. If the dealer's hand qualifies, then the hands are compared to each other according to the rules listed in Question 3. If the player's hand wins, the player is paid 1:1 on both the ante and the play bets. If the player's hand loses, then the player forfeits both the ante and the play bets. If it is a tie, both ante and play bets are returned to the player without any extra pay. A good online resource to try an interactive game play is `https://www.table-games-online.com/3-card-poker/` (make sure the pair-plus bet is set to 0).

What to do: Write a code for the function `play_round()` in `Poker.py` that plays a single round of Three-Card Poker. It takes the following as arguments:

- `dealer_hand`: an instance of `ThreeCardPokerHand` representing the dealer's hand dealt at the beginning of the round.
- `player_hand`: an instance of `ThreeCardPokerHand` representing the player's hand dealt at the beginning of the round.
- `cash`: a positive integer holding the amount of cash available to the player at the beginning of the round.
- `get_ante`: a function to be called to obtain the player's ante bet. It takes `cash` as argument and returns a positive integer.
- `is_playing`: a function to be called to determine whether the player plays or folds. It takes `player_hand` as argument and returns `True` if the player plays and `False` otherwise.

`play_round()` starts by invoking `get_ante()` to get the player's ante. It then calls `is_playing()` to determine if the player plays or folds. If the

player folds, no further steps are taken, and the function returns. If the player plays, the dealer's hand is compared against the minimum playing hand (available in `min_dealer_hand`) to determine if the dealer's hand qualifies for playing. If it does, the player's hand is compared against the dealer's to determine the winner, or if it is a tie.

`play_round()` returns a tuple `(ante, outcome)` where the `ante` is the amount returned by `get_ante()`, and `outcome` is `-1` if the player folds, `1` if the player plays and the dealer does not qualify, `2` if the dealer qualifies, and the player plays and wins, and `-2` if the dealer qualifies, and the player plays and loses.

Testing and experimentation (not assessed): Use the provided `play()` function, and sample `get_ante()` and `is_playing()` to test your implementation. In particular, the `play()` function allows you to simulate a series of games for the given initial stake and the final goal, and outputs an empirical probability of reaching the goal. Customize `get_ante()` and `is_playing()` to produce different playing strategies, and try them out to see if you can come up with a winning one. Obviously, you will not be able to (or otherwise, all casinos in the world will go bust), but you can improve your chances to up to about $40 - 45\%$. The key is in setting the ante carefully. For example, compare a strategy with a fixed ante, and one where it is doubled every round. If you are interested to learn more about the subject, a good starting point is to check out the Wikipedia page on the Gambler's Ruin problem (see `https://en.wikipedia.org/wiki/Gambler%27s_ruin`).

What to submit

1. Submit the Python source files `Card.py`, `PokerHand.py`, and `Poker.py` including your solution code in all relevant places as explained above.

2. Be sure to use these file names as otherwise your programs may fail the automated testing.

3. Make sure your code does not include any testing print-outs or calls to the `input` function anywhere but in the `if __name__ == '__main__'` sections as otherwise, your submission may fail the automated testing.

4. Submit the three files above by clicking on the relevant link under the 'Assignments' heading on Moodle.