CMPUT 274/5 – Tangible Computing Fall 2019
Weekly Exercise #4

Deck of Cards

In this exercise, you will create a new class to represent a deck of playing cards. You will create an instance of this class, populating the deck with cards that you read in from a text file. You will then use that deck object to play a simple game of *High Card Draw*.

Task #1: Create Deck class

You are given the skeleton code for a new Deck class in deck.py. This class has one attribute, which is a list of all of the cards in the deck. Note that the **first element in this list (i.e. at index 0) is the card that is on the top of the deck**, while the last element in the list is the card on the bottom of the deck.

1. Complete the `__init__()` method. Initialize the private attribute `deck` so that it is a copy (not an alias) of the list of two-character strings passed in as a parameter. One way to create a copy of a list is to use the familiar `list()` conversion function. Try a new approach: use a function from the copy module. Read the documentation for this module and decide whether you can use the `copy()` function or if you have to use the `deepcopy()` function in this case: https://docs.python.org/3/library/copy.html You may also want to check out this tutorial on copying Python objects: https://realpython.com/copying-python-objects/

2. Complete the `deal()` method. This method should remove the top card in the deck, and return it. If there are no cards in the deck, this method should instead return False.

3. Complete the `validate()` method. This method returns a tuple of two values. The first value is a Boolean value indicating whether the deck is valid (True) or not (False). If the deck is valid, then the second value will be an empty string. If the deck is not valid, then the second value will be one of the following strings describing why it is not valid:

    - `"Card x is not a valid card"`, where x is the first invalid card found, going from the top of the deck to the bottom
    - `"Incomplete deck"`, applies when there are less than 52 valid cards in the deck
    - `"Deck contains duplicate cards"`, applies when there are at least 52 valid cards in the deck and at least one card is repeated

Note that a valid deck contains exactly 52 unique cards, where each card is represented by a two-character string. The first character represents the rank of the card; either a digit `"2"-"9"`, or a capital letter as follows: `"T"` for 10, `"J"` for Jack, `"Q"` for Queen, `"K"` for King, and `"A"` for Ace. The second character represents the suit of the card – a capital letter as follows: `"S"` for Spade, `"D"` for Diamond, `"C"` for Club, `"H"` for Heart. So, for example, `"JS"` is the Jack of Spades.

4. Complete the `__str__()` method. This method should return a string representation of the deck. Each card should be displayed as its two-character string, from the top of the deck to the bottom, with a hyphen in between each card.

   Example: if the deck contains the King of Spades (top card), 6 of Hearts (middle card), and 2 of Diamonds (bottom card), the string representation of the deck would be: `"KS-6H-2D"`

Task #2: Use deck object in game of *High Card Draw*

1. Take the name of an input file called `<input_file_name>` as a command line argument. For example, your program might be called using:

   ```
   python3 deck.py example.txt
   ```

   The input file will contain one or more two-character sequences, each of which will represent a single card. You can assume that only one card will exist on each line of the input file, but there may also be leading and/or trailing whitespace on the same line which should be removed. The card on the first line of the input file will represent the card on the top of the deck, followed by the rest of the cards, in order, so that the last line of the input file will represent the card on the bottom of the deck. Use the contents of the input file (two-character card representations) to create an instance of the Deck class.

   You can assume that the input text file (if provided via the command line) will exist. The file may store each card code as either uppercase or lowercase letters, but you should populate your deck so that it only contains cards with uppercase letters. e.g. `"td"` may be present in the input file to represent the ten of diamonds, but it should be converted to `"TD"` before being stored in the deck. The input file may also contain two-character strings that are not associated with a valid card, but you do not have to validate each card while creating the deck. e.g. `"1$"` is not a valid card, but it can be added to the deck when it is first created.

2. Invoke the `validate()` method to check if the deck that you created in step 1 is valid. If it is not valid, the returned message describing why it is not valid should be displayed on the screen, and the program should terminate immediately. If it is valid, nothing needs to be displayed – simply continue with the game as described in step 3. The

`shuffledDeck.txt` file that you have been provided with contains a valid deck so that you may test your code.

3. Using your valid deck, deal the top card to the opposing player. Then deal the next top card to yourself (the dealer). Compare cards to determine who has the highest ranking card. Assume that suits don't matter – just compare the ranks. The lowest rank is 2, followed by 3, then 4, then 5, then 6, then 7, then 8, then 9, then Ten, then Jack, then Queen, then King, and Ace is highest. Display an appropriate message to indicate who has the highest ranking card: either `"Round 1: Dealer wins!"`, `"Round 1: Player wins!"`, or `"Round 1: Tie!"`

4. Continue to play *High Card Draw* (i.e. repeat step 3) until there are no more cards in the deck, printing the result for each round on the screen. (*Hint*: 26 rounds should be played.) Use the value returned from the deck's `deal()` method to decide when to stop playing rather than hardcoding a loop to repeat 26 times.

Requirements:

In your final submission:

1. We expect you to produce modular, well-designed code. This means creating one function to do each job (there are multiple jobs in this assignment, so you should not use only one function!). A general breakdown of the problem has been suggested in this assignment description, but you may feel you need to create additional functions for Task #2. A correct solution that does not use any functions (i.e. all code is in the global scope) will receive 0 marks for code design.

2. We expect that when the program is called from the command line as specified above, your program will run the *High Card Draw* game. This means that what you include under if `__name__ == "__main__"` is important.

Submission Guidelines:

Submit all of the following files as a compressed archive called `deck.tar.gz` or `deck.zip`:

- The file `deck.py` containing your complete implementation of the weekly exercise (completed `Deck` class and the *High Card Draw* game).
- Your README, following the Code Submission Guidelines.