Week 7 Assignment - DATASCI200 Introduction to Data Science Programming, UC Berkeley MIDS

Write code in this Jupyter Notebook to solve the following problems. Please upload this **Notebook** with your solutions to your GitHub repository and gradescope.

Assignment due date: 11:59PM PT the night before the Week 8 Live Session.

Objectives:

- Demonstrate how to define classes
- Design and implement class objects and class interactions
- Understand how to call methods from both inside and outside of classes
- Understand how to set internal attribute within a class

General Guidelines:

- All calculations need to be done in the classes (that includes any formatting of the output)
- Name your classes exactly as written in the problem statement
- Do NOT make separate input() statements. The classes will be passed the input as shown in the examples
- The examples are using the '>>>' as the command entered into a Jupyter coding cell with the example output shown below it.
- The examples given are samples of how we will test/grade your code. Please ensure your classes output the same information
- Inputs to classes and methods do need to be validated or checked where the problem or examples specifically state to check for inputs. Otherwise, you can assume that the correct input will be sent as shown in the examples.
- Docstrings and comments in your code are strongly suggested but won't be graded
- In each code block, do NOT delete the ### comment at the top of a cell (it's needed for the auto-grading!)
  - Do NOT print or put other statements in the grading cells. The autograder will fail - if this happens please delete those statments and re-submit
  - You will get 80 points from the autograder for this assignment and 20 points will be hidden. That is, passing all of the visible tests will give you 80 points. Make sure you are meeting the requirements of the problem to get the other 20 points.
  - You may upload and run the autograder as many times as needed in your time window to get full points
  - The assignment needs to be named HW_Unit_07.ipynb to be graded from the autograder!

- The examples given are samples of how we will test/grade your code. Please ensure your code outputs the same information.
  - In addition to the given example, the autograder will test other examples
  - Each autograder test tells you what input it is using
- Once complete, the autograder will show each tests, if that test is passed or failed, and your total score
- The autograder fails for a couple of reasons:
  - Your code crashes with that input (for example: `Test Failed: string index out of range`)
  - Your code output does not match the 'correct' output (for example: `Test Failed: '1 2 3 2 1' != '1 4 6 4 1'`)

7-1 Deck of Cards (50 points)

Please design two classes in this notebook as follows:

1. Please create a class called **PlayingCard**. (20 points)
This class should have:

- An attribute, "rank" that takes a value of "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", or "A"
- An attribute, "suit" that takes a value of "♠" "♥" "♦" or "♣". (If you don't know how to make these characters you can cut and paste from this block)
- An **init** method that:
  - Accepts as parameters a specific rank (as a string) and suit (as a string).
  - **Raise a TypeError** with "Invalid rank!" or "Invalid suit!" when a rank or suit is not valid.
- The `__str__` and `__repr__` methods to display the cards correctly (as shown below in the examples)

2. Please create a class called **Deck**. (30 points)
This class should have:

- An attribute, "cards", that holds a list of PlayingCard objects.
- An **init** method that:
  - By default stores a full deck of 52 playing card (with proper numbers and suits) in the "cards" list. Each cards will be of the class PlayingCard above
  - Allows the user to specify a specific suit of the 4 ("♠" "♥" "♦" or "♣"). In this case, the program should only populate the deck with the 13 cards of that suit.
  - After the cards object is initialized, call the "shuffle_deck()" function (below).
- A "shuffle_deck()" method that randomly changes the order of cards in the deck.
  - Import the random library to 'shuffle' the deck: https://docs.python.org/3.9/library/random.html#random.shuffle

  - Please import it at the top of your block instead of inside the class / methods.
- A "deal_card(card_count)" method that:
  - **Removes** the first `card_count` cards from the deck and **returns** them as a **list**.
  - If the deck doesnt have the `card_count` number of cards left to deal, **return** the message `Cannot deal <x> cards. The deck only has <y> cards left!` (do not raise an exception or print inside the method).

Example:

```
>>> card1 = PlayingCard("A", "♠")
>>> print(card1)
A of ♠

>>> card2 = PlayingCard("15", "♠")
< error stack >
TypeError: Invalid rank!

>>> card2 = PlayingCard("10", "bunnies")
< error stack >
TypeError: Invalid suit!

>>> deck1 = Deck()
>>> print(deck1.cards)
[K of ♠, A of ♥, 6 of ♣, 7 of ♠, J of ♦, 6 of ♠, Q of ♦, 5 of
♣, 10 of ♦, 2 of ♥, 8 of ♣, 8 of ♦, 4 of ♦, 7 of ♦, 3 of ♣, K
of ♣, 9 of ♠, 4 of ♥, 10 of ♥, 10 of ♣, A of ♠, 9 of ♥, 7 of ♥,
9 of ♣, 7 of ♣, 5 of ♠, 3 of ♦, 10 of ♠, Q of ♥, J of ♣, 5 of
♥, K of ♥, K of ♦, 2 of ♠, 8 of ♠, Q of ♣, 3 of ♠, 6 of ♥, 6 of
♦, A of ♣, A of ♦, 3 of ♥, J of ♠, 4 of ♣, 5 of ♦, 2 of ♦, 4 of
♠, 2 of ♣, Q of ♠, J of ♥, 8 of ♥, 9 of ♦]

>>> deck2 = Deck('♠')
>>> deck2.shuffle_deck()
>>> deck2.cards
[A of ♠, 10 of ♠, 3 of ♠, 7 of ♠, 5 of ♠, 4 of ♠, 8 of ♠, J of
♠, 9 of ♠, Q of ♠, 6 of ♠, 2 of ♠, K of ♠]

>>> hand = deck2.deal_card(7)
>>> hand
[A of ♠, 10 of ♠, 3 of ♠, 7 of ♠, 5 of ♠, 4 of ♠, 8 of ♠]

>>> deck2.deal_card(7)
'Cannot deal 7 cards. The deck only has 6 cards left!'
```

In [1]:
```
# Q7-1 Grading Tag:

import random
from random import shuffle

ranks = ["2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A"]
suits = ["♠", "♥", "♦", "♣"]
```

```
class PlayingCard:

    """
    Class to represent a playing card with a suit and rank
    Validated against a static list of ranks and suits as shown below
    """

    def __init__(self, rank, suit):

        """
        I have kept things simple with a single function to initiatialize the c
        Decided not to complicate matters by using setter/getter methods or dec

        """

        # initialize the two attribute values (rank and suit)

        self.rank = rank
        self.suit = suit


        # Validate for conditions and throw a TypeError if any or both conditio
        # if none of the error occur, then we can ssume the card object was cre

        if self.rank not in ranks:
            raise TypeError("Invalid rank!")
        if self.suit not in suits:
            raise TypeError("Invalid suit!")


    # using  __str__ method to print

    def __str__(self):
        return "%s of %s" %(self.rank, self.suit)


    # using __repr__ to print and make this readable

    def __repr__(self):
        return "%s of %s" %(self.rank, self.suit)



class Deck:

    """
    Class to create a deck of cards; either a full one (52 cards) or by suit (i
    has a couple methods (shuffle_deck) to randomnly shuffle the cards
    and deal_card (which takes a user-defined input and deals those cards; say
    """

    def __init__(self, chosen_suit = ["♠", "♥", "♦", "♣"]):
        self.cards = []
        self.chosen_suit = chosen_suit
        self.cards = [PlayingCard(rank, suit) for rank in ranks for suit in cho
        print(self.cards)

    def shuffle_deck(self):
        random.shuffle(self.cards)
        print(self.cards)

    def deal_card(self, card_count):
        self.popped_list = []
        self.card_count = card_count
        if card_count > len(self.cards):
            return "Cannot deal " + str(self.card_count) + " cards. The deck or
        else:
            self.popped_list = [self.cards.pop() for i in range(card_count)]
            return self.popped_list
```

7-2 Sorting Marbles (50 points)

In a particular board game, there is exactly one row and it comprises N spaces, numbered 0 through N - 1 from left to right. There are also N marbles, numbered 0 through N - 1, initially placed in some arbitrary order. After that, there are two moves available that only can be done one at a time:

- Switch: Switch the marbles in positions 0 and 1.
- Rotate: Move the marble in position 0 to position N - 1, and move all other marbles one space to the left (one index lower).

The objective is to arrange the marbles in order, with each marble i in position i.

1. Write a class, **MarblesBoard**, to represent the game above. (25 points)

- Write an `__init__` function that takes a starting sequence of marbles (the number of each marble listed in the positions from 0 to N - 1). (Notice in the sequence all the marbles are different numbers and are sequential numbered but not in order!)
- Next, write `switch()` and `rotate()` methods to simulate the player's moves as described above.
- Write a method, `is_solved()`, that returns True if the marbles are in the correct order or False otherwise.
- Additionally, write `__str__` and `__repr__` methods to display the current state of the board.

Your class should behave like the following example:

```
>>> board = MarblesBoard((3,6,7,4,1,0,8,2,5))
>>> board
3 6 7 4 1 0 8 2 5
>>> board.switch()
>>> board
6 3 7 4 1 0 8 2 5
>>> board.rotate()
>>> board
3 7 4 1 0 8 2 5 6
>>> board.switch()
>>> board
7 3 4 1 0 8 2 5 6
```

2. Write a second class, **Solver**, that actually plays the MarblesGame. (25 points)

- Write an `__init__` method that takes a MarblesBoard class in its initializer and stores it in an attribute: `board`.
- Write a `solve()` method:
  - Which repeatedly calls the switch() or the rotate() method of the given MarblesBoard until the game is solved.
  - Before the first switch or rotate, make a **list** of **tuples** with the starting tuple of `('start', <board starting state>)`
  - After each step ('switch' or 'rotate'), append to the above **list** a tuple of:
    - What step ('switch' or 'rotate') was performed. Remember, you can only do one switch or one rotate per step!
    - The state of the board
  - Return the above list as output to this method.
  - You are to come up with your own algorithm for solving the marbles game. Before you write your solve() method, you may want to practice solving some small versions of the marbles game yourself.
  - Your Solver should strive to make the algorithm reasonably efficient and strive to be the fastest runtime. (10 points are awarded based on algorithm efficiency)

Below is an example:

```
>>> board2 = MarblesBoard((1,3,0,2))
>>> solver = Solver(board2)
>>> solver.solve()
[('start', 1 3 0 2),
 ('rotate', 3 0 2 1),
 ('rotate', 0 2 1 3),
 ('rotate', 2 1 3 0),
 ('switch', 1 2 3 0),
 ('rotate', 2 3 0 1),
 ('rotate', 3 0 1 2),
 ('rotate', 0 1 2 3)]
```

You may be interested to know that your program is a variation of a well-known sorting algorithm called bubble sort. Bubble sort would normally be used on a list of items, not on a rotating track, but adapting your algorithm to this setting could be straight-forward.

In [2]:
```
# Q7-2 Grading Tag:

import random

class MarblesBoard:
    """creates a marble board with number marbles in specific spots"""
    def __init__(self, marble_sequence):
        self.board = [x for x in marble_sequence]

    def __str__(self):
        return " ".join(map(str, self.board))
    def __repr__(self):
        return "%r " % (self.board)

    def switch(self):
        """switch the marbles in position 0 and 1"""
        self.board[0], self.board[1] = self.board[1], self.board[0]
        return self.board

    def rotate(self):
#          """
#          Rotates item in position o to position N-1. All remaning items are mo
#          """
        self.board = self.board[1:] + [self.board[0]]
        return self.board

    def is_solved(self):
        return self.board == sorted(self.board)

    def should_rotate(self):
        # used a random generator for True/False -- used to select Switch or Ro
        return random.choice([True, False])

class Solver:
    """solves the marble sorting game when given a marble board as input"""

    def __init__(self, marbles_board):
        self.marbles_board = marbles_board
#         myboard = self.marbles_board

    def solve(self):
        steps = 0
        mylist = []
        myboard = self.marbles_board
        mylist.append(('start', str(myboard)))
#         mylist.append(('start', (myboard)))

#         print(mylist)


        while not self.marbles_board.is_solved():
#             if steps == 200:break
            if self.marbles_board.should_rotate():
                self.marbles_board.rotate()
#                 print('rotate', self.marbles_board)
#                 mylist.append(('rotate ' + str(self.marbles_board)))
                mylist.append(('rotate', str(self.marbles_board)))
#                 mylist.append(('rotate', (self.marbles_board)))
#                 print(mylist)

            else:
                self.marbles_board.switch()
#                 print('switch', self.marbles_board)
#                 mylist.append(('rotate ' + str(self.marbles_board)))
                mylist.append(('switch', str(self.marbles_board)))
#                 mylist.append(('switch', (self.marbles_board)))
#                 print(mylist)

            steps += 1
        return mylist
```

```
#        print(f"Number of steps: {steps}")
#        print(self.marbles_board)
```

In [ ]:

```
```