STAT 29000 Project 12

Topics: Python, classes

Motivation: We'd be remiss spending almost an entire semester solving data driven problems in python without covering the basics of classes. Whether or not you will ever choose to use this feature in your work, it is best to at least understand some of the basics so you can navigate libraries and other code that does use it.

Context: We've spent nearly the entire semester solving data driven problems in python, and now we are going to learn about one of the primary features in python: classes. Python is an object oriented programming language, and as such, much of python, and the libraries you use in python are objects which have properties and methods. In this project we will explore some of the terminology and syntax relating to classes.

Scope: Python, classes, and some other concepts.

You can find useful examples that walk you through relevant material here or on scholar:

`/class/datamine/data/spring2020/stat29000project12examples.ipynb`.

It is highly recommended to read through these to help solve problems.

Use the template found here or on scholar:

`/class/datamine/data/spring2020/stat29000project12template.ipynb`

to submit your solutions.

After each problem, we've provided you with a list of keywords. These keywords could be package names, functions, or important terms that will point you in the right direction when trying to solve the problem, and give you accurate terminology that you can further look into online. You are not required to utilize all the given keywords. You will receive full points as long as your code gives the correct result.

Question 1: card

1a. *(1 pt)* Create a `Card` class with a `number` and a `suit`. The `number` should be any number between 2-10 or any of a 'J', 'Q', 'K', or 'A' (for Jack, Queen, King, or Ace). The `suit` should be any of: "Clubs", "Hearts", "Spades", or "Diamonds". You should initialize a `Card` by first providing the `number` then the `suit`. Make sure that any provided `number` is one of our valid values, if it isn't, throw an exception (that is, stop the function and return a message):

```
raise Exception("Suit wasn't one of: clubs, hearts, spades, or diamonds.")
```

Here are some examples to test:

```
my_card = Card(11, "Hearts") # Exception: Number wasn't 2-10 or J, Q, K, or A.
my_card = Card(10, "Stars") # Suit wasn't one of: clubs, hearts, spades, or diamonds.
my_card = Card("10", "Spades")
my_card = Card("2", "clubs")
my_card = Card("2", "club") # Suit wasn't one of: clubs, hearts, spades, or diamonds.
```

Important note: *Accept both upper and lowercase variants for both* ***`suit`*** *and* ***`number`****. To do this, convert any input to lowercase prior to processing/saving. For* ***`number`****, you can do* ***`str(num).lower()`****.*

Item(s) to submit:

- A cell in the Jupyter notebook containing the python code used to create your `Card` class.
- One cell in the Jupyter notebook for each of the examples to test (and their output, total of 5 cells).

1b. *(.5 pts)* Usually when we talk about a particular card, we say it is a "Four of Spades" or "King of Hearts", etc. Right now, if you `print(my_card)` you will get something like `<__main__.Card object at 0x7fccd0523208>`. Not very useful. We have a dunder method for that!

Implement the `__str__` dunder method to work like this:

```
print(Card("10", "Spades")) # 10 of spades
print(Card("2", "clubs")) # 2 of clubs
```

Another, closely related dunder method is called `__repr__` – short for representation. This is similar to `__str__` in that it should print the code used to create the object being printed. So for our examples:

```
print(repr(Card("10", "Spades"))) # Card(str(10), "spades")
print(repr(Card("2", "clubs"))) # Card(str(2), "clubs")
```

Implement both dunder methods to function as exemplified.

Item(s) to submit:

- A cell in the Jupyter notebook containing your updated `Card` class.
- A cell in the Jupyter notebook containing the examples (both the "print" and "repr" examples) to test (and their output).

1c. *(1 pt)* It is natural that we should be able to compare cards, after all thats necessary to play nearly any game. Typically there are two ways to "sort" cards. Ace high, or ace low. Ace high is when the Ace represents the highest card.

Implement the following dunder methods to enable comparison of cards where ace is high:

`__eq__ __lt__ __gt__`

Make sure the following examples work:

```
card1 = Card(2, "spades")
card2 = Card(3, "hearts")
card3 = Card(3, "diamonds")
card4 = Card(3, "Hearts")
card5 = Card("A", "Spades")
card6 = Card("A", "Hearts")
card7 = Card("K", "Diamonds")

print(card1 < card2) # True
print(card1 < card3) # True
print(card2 == card3) # True
print(card2 == card4) # True
print(card3 < card4) # False
print(card4 < card3) # False
print(card5 > card4) # True
print(card5 > card6) # False
print(card5 == card6) # True
print(card7 < card5) # True
print(card7 > card1) # True
```

Important note: *Two cards are deemed equal if they have the same number, regardless of their suits.*

Hint: *There are many ways to deal with comparing the "JKQA" against other numbers. One possibility is to have a dict that maps the value of the card to it's numeric value.*

Item(s) to submit:
- A cell in the Jupyter notebook containing your updated `Card` class.
- A cell in the Jupyter notebook containing the example(s) to test (and their output).

Question 2: deck

We now have cards that we can create, print, and compare. Great! Let's make a deck now.

2a. *(.5 pts)* Create a `Deck` class that contains the standard 52 cards. You will create new deck like this:

```
my_deck = Deck()
my_deck.cards
# [Card(str(2), "clubs"),
# Card(str(3), "clubs"),
# ...
# ]
```

Item(s) to submit:
- A cell in the Jupyter notebook containing your `Deck` class.
- A cell in the Jupyter notebook containing the example(s) to test (and their output).

2b. *(.5 pts)* Find the length of the deck using the `len` function. What? You get an error? Implement the `__len__` dunder method and try again.

```
my_deck = Deck()
len(my_deck) # should be 52
```

Item(s) to submit:
- A cell in the Jupyter notebook containing your updated `Deck` class.
- A cell in the Jupyter notebook containing the example(s) to test (and their output).

2c. *(.5 pts)* When using the `Deck` class it makes sense to do something like `my_deck[9]`. If you try it now, you'll get a TypeError and message saying `Deck` does not support indexing. Let's fix that. Implement the `__getitem__` dunder method and try again.

```
my_deck = Deck()
my_deck[4]
# Card(str(6), "clubs")
```

Item(s) to submit:
- A cell in the Jupyter notebook containing your updated `Deck` class.
- A cell in the Jupyter notebook containing the example(s) to test (and their output).

2d. *(.5 pts)* What is one of the main thing someone does with a deck of cards? Shuffle! Try and use the `shuffle` function from the `random` library on a `Deck`. What you will get is a TypeError that says `Deck` doesn't support item assignment. This is easy to fix. Implement the `__setitem__` dunder method and try again.

```
my_deck = Deck()
from random import shuffle
shuffle(my_deck)
my_deck[:3] # excellent
```

Hint: *It is important to note that you do not need to use* ***`random.shuffle`*** *anywhere inside of your* ***`__setitem__`*** *implementation.* ***`__setitem__`*** *only allows* ***`random.shuffle`*** *to properly assign (or set) values inside your class. By implementing* ***`__setitem__`****,* ***`random.shuffle`*** *will understand how to shuffle your deck.*

Hint: *The* ***`__setitem__`*** *implementation is very short and simple and is only a single expression.*

Item(s) to submit:
- A cell in the Jupyter notebook containing your updated `Deck` class.
- A cell in the Jupyter notebook containing the example(s) to test (and their output).

Question 3: war!

We've now built a `Card` class and a `Deck` class. Let's put these classes to use. War is one of the most simple card games. Your colleague provided you with an additional couple of classes: `Player` and `War`.

```
import queue
from random import shuffle


class Player:

    def __init__(self, name):
        self.name = name
        self.hand = queue.Queue(52)


class War:

    def __init__(self, p1, p2):
        self.player1 = p1
        self.player2 = p2
        self.deck = Deck()

    def deal(self):
        shuffle(self.deck)

        for card in self.deck[:26]:
            self.player1.hand.put(card)

        for card in self.deck[26:]:
            self.player2.hand.put(card)

    def score(self):
        return f'Player 1: {self.player1.hand.qsize()}\nPlayer 2: {self.player2.hand.qsize()}'

    def play(self):

        # each player draws a card
        p1card = self.player1.hand.get()
        p2card = self.player2.hand.get()

        print(f'(Player 1) {p1card} x (Player 2) {p2card}')

        if p1card == p2card:
```

```
            print(f'War!')

            # in war, each player places 1 card face down
            face_down_p1 = self.player1.hand.get()
            face_down_p2 = self.player2.hand.get()

            print(f'Players put 1 card facedown.')

            # then the players play again
            winner = self.play()

            # the winner gets the face down cards
            # and the cards that caused a war
            winner.hand.put(face_down_p1)
            winner.hand.put(face_down_p2)
            winner.hand.put(p1card)
            winner.hand.put(p2card)

            return winner

        elif p1card > p2card:
            print(f'Player 1 wins!')

            # player 1 wins and gets both cards
            self.player1.hand.put(p1card)
            self.player1.hand.put(p2card)

            return self.player1

        else:
            print(f'Player 2 wins!')

            # player 2 wins and gets both cards
            self.player2.hand.put(p2card)
            self.player2.hand.put(p1card)

            return self.player2

    def winner(self):
        if self.player1.hand.empty():
            return self.player2.name
        elif self.player2.hand.empty():
            return self.player1.name
        else:
            return None
```

Here is a resource to better understand a `Queue`. A `Queue` is a data structure that is "first in first out" (FIFO, as opposed to another data structure called a `Stack` which is "last in first out" (LIFO)). `Queue` has two primary methods: `get` and `put`. `put` adds an object of some sort to the "end of the line", and `get` goes to the "front of the line" and removes and returns an object. Here are some examples:

```
import queue

# create a fresh queue that can hold 5 objects
```

```
my_queue = queue.Queue(5)

# add a number to the queue
my_queue.put(1)
print(list(my_queue.queue))

# add a number to the queue
my_queue.put(2)
print(list(my_queue.queue))

# add a number to the queue
my_queue.put(3)
print(list(my_queue.queue))

# as you can see 1 is at the front of the queue, then 2, then 3

# get the object in the "front of the line"
value = my_queue.get()
print(value)
print(list(my_queue.queue))

# add a value to the back of the line
my_queue.put(4)
print(list(my_queue.queue))

# as you can see when we `put` the value/object goes to the back of the line
# when we `get` we get whatever value/object that is in the front of the line
```

Read through the code and answer the following questions.

3a. *(1.5 pts)* Create 2 players: `player1` and `player2`. Create a game of war called `war` between `player1` and `player2`. Deal the cards, play a single hand, and get the score. Place the code you used to do this in a cell in your notebook.

Create a new cell right below the cell with the code you used to do this. Copy and past the part of the code used to play a single hand. Re-run this cell until "War!" occurs.

Read the code again. Can you see a problem with the code if we wanted to continue to use `play`?

Hint: *"War!" means that the two players have cards of the same rank. As the process is random, the number of times that you need to run the cell is not fixed. You may find it convenient to use the shortcut key for running the selected code cell repeatedly: Ctrl + Enter for Windows/Scholar, and Command + Enter for MacOS.*

Important note: *Up until this point, we have really only been adding dunder methods to our classes. Now, we will be writing and modifying methods that change the attributes of our class when run. For example, the* `deal` *method from the* `War` *class modifies the player1 and player2 objects* `hand` *attribute.*

Item(s) to submit:

- A cell in the Jupyter notebook containing the code you used to: create `player1`, `player2`, and `war`. In addition, the code you used to deal, play a single hand, and get the score.
- A markdown cell in the Jupyter notebook with a statement explaining the problem with the provided code OR simply saying you don't understand yet what the problem is (you'll get full credit for either answer).

3b. *(2 pts)* In war, the game ends as soon as one player has all of the cards or in the situation where one

player doesn't have enough cards to continue playing (in a "war!" situation). As you can see from the code, if we were to continue to `war.play()`, eventually we would have an error as there is no code that stops the game and declares a winner. Modify the `War` class to gracefully end a game.

Hints:

1. Create a `game_over` method that checks if either player's hand is empty. If so, it returns `True` (and the game is over), otherwise, it returns `False`. You should use a while loop to play one game of War until the end, at which time the user can call the `winner` method to find the winner. You will use the `game_over` method in the while loop to accomplish this. The only other instance that this strategy doesn't cover is when a war happens and a player may not have enough cards to continue on (and therefore loses).
2. Add a check at the beginning of the chunk of code where there is a tie (war). In order to go into war, players must each have a minimum of two more cards after the initial draw. If a player doesn't have 2+ cards going into war, they forfeit and transfer all cards to the opponent. Check to see if player 1 has < 2 cards, if so, transfer all cards to player 2, and return the winner (`self.player2`). In addition, check to see if player 2 has < 2 cards, if so, transfer all cards to player 1, and return the winner (`self.player1`).
3. You should modify the `deal` method to reset each players' hand before giving each player half of the cards. Assigning each players' hand to a fresh `Queue` as shown in the `Player` class's `__init__` function would be an effective way to do this.

Item(s) to submit:
- A cell with the modified `War` class.

3c. *(2 pts)* Great! You may now see how – depending on the task at hand – classes can make code both more concise and readable. Use our new class and simulate 100 games of war. Create a bar plot using `plotly` to show the wins by player.

Item(s) to submit:
- A cell with the code used to simulate 100 games of war.
- A `plotly` bar plot showing the number of wins for each player.

3d. *(optional, 0 pts)* Modify the `deal` method to accept an argument, n, where n cards are dealt to the first player, and the remaining cards are dealt to the second player. Use the new `deal` method to give an advantage to player1, simulate 100 games of war. Create a bar plot using `plotly` to show the wins by player. Rinse and repeat with different advantages to see the effect of uneven splitting of the deck.

Project Submission:

Submit your solutions for the project at this URL: https://classroom.github.com/a/xDe0oclj using the instructions found in the GitHub Classroom instructions folder on Blackboard.

Important note: Make sure you submit your solutions in both .ipynb and .pdf formats. We've updated our instructions to include multiple ways to convert your .ipynb file to a .pdf on scholar. You can find a copy of the instructions on scholar as well: `/class/datamine/data/spring2020/jupyter.pdf`. If for some reason the script does not work, just submit the .ipynb.