CS106 Lab 10: I guess you realize this means War!
Dr. Victor Norman

Goals: To practice...
• designing classes
• implementing classes
• unit testing classes

In today's most excellent lab, you will design and implement classes to represent a playing card and a deck of cards.

Card Class

First we need to design a Card class to represent a playing card. This should be easy: a playing card has only 2 important properties. What are they?

Here’s the answer: each card is one of 13 denominations (2-10, J, Q, K, and A) in 4 suits (spades, hearts, clubs, and diamonds).

Remember that a class is like a recipe or a template. It defines what that thing will look like and act like, but it does not actually make one of those things. Also, when designing the class you need to decide what attributes (state or values) each object will need to maintain, and what methods (or functions or operations) need to be implemented.

So, for a Card instance, the card will have 2 states or properties -- its suit and its denomination. And, the actual values for these two really should be determined at card creation time -- i.e., at construction or instantiation. After a card has been created, there is no need to provide a way to change a card’s denomination or suit, right? (However, one could implement at third state for a card -- "face-up" or "face-down". One would want to be able to change this state.)

One of the early decisions we have to make is how to store the 2 attributes -- i.e., what type will each be? One could store the denomination as an integer from 2 to 14, or as a one-character string (e.g., "2", "3", ..., "J", "Q", "K", "A"). The suit could be stored as a one-character string (e.g., "S", "D", "H", and "C") or as a longer string ("spades", "diamonds", etc.). How does one best decide which type to use for an attribute?

One of the ways to decide is to figure out what one will do with an attribute. One of the things we will want to do with cards is compare 2 of them to see which one has a higher denomination. Comparing two cards’ denominations would be quite trivial if we stored the denomination attribute as an integer, right? However, we also want to be able to print out a card in a nice readable fashion, such as "J of Hearts", "3 of clubs", "Ace of spades", or perhaps just "J H", "3 C", or "A S". But, if we store the denomination as an integer, then printing out the face cards with names like "Ace" or "King" is more difficult as we are not
storing "Ace", "A", "King", or "K", but instead are storing 14 (for Ace) and 13 (for King).

So, what to do, what to do? One solution is to store the denomination both ways -- as an integer and as a string representation of that denomination. It is usually not a good idea to store one piece of information in two ways: you have to make sure you keep both things in sync with each other, and you use twice as much memory space. Another solution is to have some code to convert the integer representation (e.g., 14) into an appropriate string representation (e.g., "Ace"). That's the solution we'll use here.

So, in conclusion, we'll store the denomination as an integer, and the suit as a string -- either a short one-character string or a longer string -- you decide. Thus, with that decision in mind, we can now create our constructor, and we know that it will take self as the first argument (always), a denomination argument (which must be an integer), and a suit argument (which must be a string).

Exercises:
1. Start up Canopy and make a new folder lab10 / in myCS106 / as usual. Create a file in the lab10/ directory called card.py.
2. Create a class called Card and its constructor method. The constructor should take as parameters the two properties of the card, and should store those properties as attributes of self. That's it. It just gets the suit and denomination it is passed, and stores those values in self, as attributes. Remember to put an underscore at the beginning of the attribute names. See your notes or page 206 of the text book for a reminder of what a constructor looks like.
3. At the bottom of card.py (outside of the Card class), write a few lines of code to see if you can successfully create some Card instances, and print out the attributes in these instances.

Now, wouldn't it be nice if we could just call

```
print card1      # where card1 is a Card instance
```

Python provides a slick way to print out a class object. Whether you know it or not, integers, strings, floats, lists, tuples, etc., are all Python classes, and when you call print on an (integer, float, tuple, etc.) object, print calls a class-specific method that returns (not prints) a string representation of that object. We can do the same for Cards, so that we can create a card instance, card1, and then just call print card1.

The output from this print call should include all the distinguishing characteristics of the instance. To get print to work in this way, we need to define a method in the Card class called __str__. (Note: that is two underscores before and after str.) The method must return a string (not print it) that represents that card instance. Note that when you call str(card1) that will also call __str__().
Exercises:
4. Create a method/function \texttt{__str__} in your \texttt{Card} class that returns a string representing the important characteristics of your \texttt{Card} instance. The denomination of the card (i.e., the integer 2 through 14) should be printed out as "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", or "A". The suit of the card may be printed out as "S", "C", "D", or "H", or "Spades", "Clubs", etc. Remember that this is a method of \texttt{Card}, so you must define this method inside the \texttt{Card} class, it must take only the single \texttt{self} argument, and it must return a string, such as "2 of Clubs", "A of Spades", "10 C", or "J H", e.g. To do this, you may have to have a big if-elif-elif block of code to handle the jack, queen, king, or ace situations. E.g.,

\begin{verbatim}
if self._denom == 14:    # ace
denom_str = "A"
elif self._denom == 13:  # king
denom_str = "K"
... etc...
\end{verbatim}

Unit testing Card

When you run the file \texttt{card.py} from Canopy all the code in the \texttt{if \_\_name\_ \_ == \"\_\_main\_\_\"\:} clause will be executed (2 underscores before and after "name" and "main"). The code inside this if clause is called a unit test. This code typically instantiates the class multiple times with various parameters (if it is useful) and then executes all methods in the class at least once. It is a good idea to compare the results of each method call to the expected results, using assert statements. E.g.,

\begin{verbatim}
if \_\_name\_ \_ == \"\_\_main\_\_\":
    card = Card(10, "S")
    card2 = Card(14, "C")
    # Unit test the __str__() method.
    assert str(card) == "10 S"    # tests __str__ implementation
    assert str(card2) == "A C"    # handles facecards correctly?
\end{verbatim}

I like to put a print statement at the end of my unit tests, stating that the unit tests are complete:

\begin{verbatim}
print "End of unit tsts. If no errors seen, then all tests passed."
\end{verbatim}

Exercises:
5. Create some unit tests in \texttt{card.py} that test instantiating a \texttt{Card} object. Then, add code to exercise all the methods you’ve defined for a \texttt{Card}. Your file should have only the definition of the \texttt{Card} class in it, and then code that is in the \texttt{if \_\_name\_ \_ …} clause. (There shouldn’t be any code outside of the \texttt{if \_\_name\_ \_ clause.}
6. What should happen if you try to create a bad card? E.g., what happens if you try to create a card, with denomination -1 and suit “Q”? Think about this... don’t worry about how to handle it, though.

One of the operations you will have to do with a Card is to compare one instance to another, using <, >, or ==. E.g., if you would like to be able to write this code:

```python
if card < card2:  # where card and card2 are instances of Card
etc
```

But, what does it mean that card is less than card2? For us, we’ll make it mean that card’s denomination (number) is less than card2’s denomination. But, how do we write code that gets called when you use < on two cards? Or > between 2 cards? Or ==?

One can easily implement all of this in python by overriding the built-in method __cmp__.

(The way this happens under the hood is complex and tricky, but fortunately we don’t have to understand it all. So, we just need to implement __cmp__.)

The signature of __cmp__ (in our Card class) must be:

```python
def __cmp__(self, other) --> integer  # i.e., it returns int
```

Note that self and other are instances of Card. In the example above, self would refer to card, and other would be card2. To work correctly, the function __cmp__ must return -1 if self < other, 0 if self == other, and 1 if self > other. ("cmp" stands for "compare"). Again, remember that if we say self < other, we means self's denomination is less than other's denomination.

**Exercises:**

7. Implement and unit test __cmp__ for Cards. Make the code return 0 if self's denomination == other's denomination, -1 if self's denom < other's denom, and 1 if self's denom > other's denom. Note again, that __cmp__ must be defined "within" the Card class, so make sure you indent it correctly. For our purposes only the denomination (not the suit) of the card matters. Thus, these assertions should be true,

```python
assert Card(10, "S") == Card(10, "D")
assert Card(4, "H") < Card(13, "S")
assert Card(14, "D") > Card(13, "C")
```

**A Deck of Cards**

A Card object is not very interesting, as once you create one, there is not much one can do with it on its own. However, if you have a deck of cards, then things start to get interesting.
For our purposes a deck of cards is just a collection/stack of cards. (That is, a deck does **not** have to mean a full deck.) A standard full deck of cards contains one of each kind of card – 52 cards in all. I recommend you store cards in the deck in a **list**.

**Exercises:**

8. Open a new file called deck.py in the same folder as card.py. In this file, create a Deck class and its constructor, which has one parameter: `self` (as required). In the constructor, initialize the list of cards that represents the cards in “this” deck to be empty: create `self._cards`, initialized to an empty list. That’s all the code in the constructor.

9. Create a method in Deck called `addAllCards(self)` that adds the 52 cards of a standard deck to this deck instance. Think about how you do this: you need to make a "2 C", "3 C", "4 C", ... "A C", then "2 S", "3 S", "4 S", ... up to "A S", then "2 D" to "A D" and then "2 H" up to "A H". So, for each of 2 through 14, and each of ("C", "S", "D", "H"), create a card, and append it to `self._cards`. `addAllCards()` does not need to return anything, as everything it has done is stored within `self`.

Test this code with a unit test. (This unit test is kind of hard to do. One could simply see if `self._cards` has 52 cards in it. Or one could print out the deck and make sure each card is in there... You decide.)

10. Add a `_str_` method to Deck to return a string representation of a Deck. Recall that if you define this `_str_` method, then you can print a whole deck of cards with 1 print call. The code will need to iterate through the list of cards, converting each to a string, and appending that string onto a resulting string. Then that resulting string is returned.

11. Add a method to Deck to shuffle the deck. There is an easy way to shuffle a list in the random module.

12. (Note that in my implementation, I decided the top of the deck was `self._cards[0]` and the bottom of the deck was `self._cards[-1].`)

13. Add these methods to your Deck class: (in my implementation, I think every one of these methods was 1 line of code.)

   removeTopCard(self) --> Card: remove the top card and return it.
   isEmpty(self) --> boolean: return True if the deck has no cards in it. (and False otherwise – duh!)
   addCardsToBottom(self, cardsSeq): cardsSeq is a sequence (a list or tuple) of Card objects. Add them to the end of the _cards list. Returns nothing.
   numCards(self) --> int: returns the number of cards in the deck.
   cardOnBottom(self) --> Card: returns the card on the bottom, without removing it from the deck.
   getCards(self) --> list: returns the list of cards in this deck.

Unit test all of these methods. If you can’t figure out what these methods are supposed to do, ask.
Submission:

Submit your `card.py` and `deck.py` classes by copying them to

```
/home/cs/106/current/<youruserid>/lab10/
```

(after creating the `lab10` directory).

Grading Rubric:
8 points for correctness;
7 points for cleanliness (good and correct variable, function, and attribute names, good docstrings, etc.);
5 points for complete tests.
Total: 20 points.