Lecture 17

Week 7 Friday

Miles Chen, PhD

Taken from Chapter 1 of Fluent Python by Luciano Ramalho

I highly recommend this text if you are interested in more advanced Python programming.

While you are connected to the UCLA network, you can access the book from your browser here:

https://proquest.safaribooksonline.com/book/programming/python/9781491946237 (https://proquest.safaribooksonline.com/book/programming/python/9781491946237)

Another Card Deck Class

We have already seen a class defined to create a deck.

The following will be another class definition for another card deck.

What's notable about the following class definition is that we will implement two special "double under" methods: `__getitem__` and `__len__`

Named Tuples

Named tuples are like a shortcut for defining a very simple class.

For example, in an earlier lecture, we defined a class Point:

In [1]:

```python
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

In [2]:

```python
p = Point()
```

In [3]:

```python
print(p)
```

```
(0, 0)
```

We can create a similar class very quickly using `namedtuple` which is found in the module `collections`.

To use named tuple, you provide the name of the class, and then you provide a list of attributes which will be stored as a tuple.

In [4]:

```
import collections
Point = collections.namedtuple('Point', ['x','y'])
```

In [5]:

```
print(Point) # Point is a class
```

```
<class '__main__.Point'>
```

In [6]:

```
# we can create objects of class Point as before
p = Point(1, 2)
```

In [7]:

```
# when we print, it prints out in the 'named tuple' form
print(p)
```

```
Point(x=1, y=2)
```

In [8]:

```
# you can access the attributes like before:
p.x
```

Out[8]:

```
1
```

In [9]:

```
# however, you cannot set attributes in a named tuple like you would a class
p.x = 3
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-9-8b0fd7dc14e1> in <module>
      1 # however, you cannot set attributes in a named tuple like you would a
class
----> 2 p.x = 3

AttributeError: can't set attribute
```

If you need to make the class more complicated by adding more methods, you can create a new class that inherits from the namedtuple.

```
class Point_more(Point):
    # more stuff
    pass
```

For our deck, we'll use namedtuple to create a class for our cards:

```
In [10]: # remember to import collections first
         Card = collections.namedtuple('Card', ['rank', 'suit'])
```

```
In [11]: test_card = Card("7", "diamonds")
```

```
In [12]: test_card
```

```
Out[12]: Card(rank='7', suit='diamonds')
```

Now that we have defined a class for cards, we can create a class for a standard 52-card deck, also called a French Deck.

In [13]:

```
class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades hearts diamonds clubs'.split()
    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]
    def __len__(self):
        return len(self._cards)
    def __getitem__(self, position):
        return self._cards[position]
```

Within this class definition, we have a few things going on.

`ranks = [str(n) for n in range(2, 11)] + list('JQKA')` uses a list comprehension to create a list of `['2', '3', ... , '10', 'J', 'Q', 'K', 'A']`

`'spades hearts diamonds clubs'.split()` splits the string into a list of strings, so suits is equal to `['spades', 'hearts', 'diamonds', 'clubs']`

The `__init__` method uses a list comprehension to iterate through all ranks and all suits to create a list of 52 Card class objects. It names the list `_cards`

The special method `__len__` will return the length of the list `_cards`

The special method `__getitem__` will return the card object from the list `_cards` at the index `position`

The `__getitem__` method provides us a way to retrieve items with an index.

```
In [14]: deck = FrenchDeck()
```

```
In [15]: len(deck)
```

```
Out[15]: 52
```

```
In [16]: # first item in the deck
         deck[0]
```

```
Out[16]: Card(rank='2', suit='spades')
```

```
In [17]: # last item in the deck
         deck[-1]
```

```
Out[17]: Card(rank='A', suit='clubs')
```

Should we create a method to pick a random card? No need. Python already has a function to get a random item from a sequence: `random.choice`. We can just use it on a deck instance:

```
In [18]: from random import choice
         choice(deck)
```

```
Out[18]: Card(rank='9', suit='spades')
```

```
In [19]: choice(deck)
```

```
Out[19]: Card(rank='8', suit='hearts')
```

```
In [20]: choice(deck)
```

```
Out[20]: Card(rank='A', suit='spades')
```

Because our `__getitem__` delegates to the `[]` operator of `self._cards`, our deck automatically supports slicing.

In [21]:

```
deck[:3]
```

Out[21]:

```
[Card(rank='2', suit='spades'),
 Card(rank='3', suit='spades'),
 Card(rank='4', suit='spades')]
```

In [22]:

```
deck[0:13:2]
```

Out[22]:

```
[Card(rank='2', suit='spades'),
 Card(rank='4', suit='spades'),
 Card(rank='6', suit='spades'),
 Card(rank='8', suit='spades'),
 Card(rank='10', suit='spades'),
 Card(rank='Q', suit='spades'),
 Card(rank='A', suit='spades')]
```

In [23]:

```
deck[12::13] # pick the A and every 13th card after that
```

Out[23]:

```
[Card(rank='A', suit='spades'),
 Card(rank='A', suit='hearts'),
 Card(rank='A', suit='diamonds'),
 Card(rank='A', suit='clubs')]
```

Just by implementing the `__getitem__` special method, our deck is also iterable:

In [24]:

```python
for card in deck:
    print(card)
```

```
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
Card(rank='4', suit='spades')
Card(rank='5', suit='spades')
Card(rank='6', suit='spades')
Card(rank='7', suit='spades')
Card(rank='8', suit='spades')
Card(rank='9', suit='spades')
Card(rank='10', suit='spades')
Card(rank='J', suit='spades')
Card(rank='Q', suit='spades')
Card(rank='K', suit='spades')
Card(rank='A', suit='spades')
Card(rank='2', suit='hearts')
Card(rank='3', suit='hearts')
Card(rank='4', suit='hearts')
Card(rank='5', suit='hearts')
Card(rank='6', suit='hearts')
Card(rank='7', suit='hearts')
Card(rank='8', suit='hearts')
Card(rank='9', suit='hearts')
Card(rank='10', suit='hearts')
Card(rank='J', suit='hearts')
Card(rank='Q', suit='hearts')
Card(rank='K', suit='hearts')
Card(rank='A', suit='hearts')
Card(rank='2', suit='diamonds')
Card(rank='3', suit='diamonds')
```

In [25]:

```
for card in reversed(deck):
    print(card)
```

```
Card(rank='A', suit='clubs')
Card(rank='K', suit='clubs')
Card(rank='Q', suit='clubs')
Card(rank='J', suit='clubs')
Card(rank='10', suit='clubs')
Card(rank='9', suit='clubs')
Card(rank='8', suit='clubs')
Card(rank='7', suit='clubs')
Card(rank='6', suit='clubs')
Card(rank='5', suit='clubs')
Card(rank='4', suit='clubs')
Card(rank='3', suit='clubs')
Card(rank='2', suit='clubs')
Card(rank='A', suit='diamonds')
Card(rank='K', suit='diamonds')
Card(rank='Q', suit='diamonds')
Card(rank='J', suit='diamonds')
Card(rank='10', suit='diamonds')
Card(rank='9', suit='diamonds')
Card(rank='8', suit='diamonds')
Card(rank='7', suit='diamonds')
Card(rank='6', suit='diamonds')
Card(rank='5', suit='diamonds')
Card(rank='4', suit='diamonds')
Card(rank='3', suit='diamonds')
Card(rank='2', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='K', suit='hearts')
Card(rank='Q', suit='hearts')
Card(rank='J', suit='hearts')
Card(rank='10', suit='hearts')
Card(rank='9', suit='hearts')
Card(rank='8', suit='hearts')
Card(rank='7', suit='hearts')
Card(rank='6', suit='hearts')
Card(rank='5', suit='hearts')
Card(rank='4', suit='hearts')
Card(rank='3', suit='hearts')
```

Iteration is often implicit. If a collection has no `__contains__` method, the `in` operator does a sequential scan. Case in point: `in` works with our `FrenchDeck` class because it is iterable.

```
In [26]: Card('Q', 'hearts') in deck

Out[26]: True
```

```
In [27]: Card('7', 'beasts') in deck

Out[27]: False
```

How about sorting? A common system of ranking cards is by rank (with aces being highest), then by suit in the order of spades (highest), then hearts, diamonds, and clubs (lowest). Here is a function that ranks cards by that rule, returning 0 for the 2 of clubs and 51 for the ace of spades:

In [28]:

```
suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)
```

In [29]:

```
def spades_high(card):
    # a function to return a value 0 for 2 of clubs, 51 for ace of spades
    rank_value = FrenchDeck.ranks.index(card.rank)
    return rank_value * len(suit_values) + suit_values[card.suit]
```

In [30]:

```python
# we can then print using the sorting key
for card in sorted(deck, key=spades_high):
    print(card)
```

```
Card(rank='2', suit='clubs')
Card(rank='2', suit='diamonds')
Card(rank='2', suit='hearts')
Card(rank='2', suit='spades')
Card(rank='3', suit='clubs')
Card(rank='3', suit='diamonds')
Card(rank='3', suit='hearts')
Card(rank='3', suit='spades')
Card(rank='4', suit='clubs')
Card(rank='4', suit='diamonds')
Card(rank='4', suit='hearts')
Card(rank='4', suit='spades')
Card(rank='5', suit='clubs')
Card(rank='5', suit='diamonds')
Card(rank='5', suit='hearts')
Card(rank='5', suit='spades')
Card(rank='6', suit='clubs')
Card(rank='6', suit='diamonds')
Card(rank='6', suit='hearts')
Card(rank='6', suit='spades')
Card(rank='7', suit='clubs')
Card(rank='7', suit='diamonds')
Card(rank='7', suit='hearts')
Card(rank='7', suit='spades')
Card(rank='8', suit='clubs')
Card(rank='8', suit='diamonds')
Card(rank='8', suit='hearts')
Card(rank='8', suit='spades')
Card(rank='9', suit='clubs')
Card(rank='9', suit='diamonds')
Card(rank='9', suit='hearts')
Card(rank='9', suit='spades')
Card(rank='10', suit='clubs')
Card(rank='10', suit='diamonds')
Card(rank='10', suit='hearts')
Card(rank='10', suit='spades')
```

By implementing the special methods `__len__` and `__getitem__`, our FrenchDeck behaves like a standard Python sequence, allowing it to benefit from core language features (e.g., iteration and slicing) and from the standard library, as shown by the examples using `random.choice`, `reversed`, and `sorted.`

As implemented so far, the `FrenchDeck` cannot be shuffled, because it is immutable: the cards and their positions cannot be changed, unless we handle the `_cards` attribute directly, which violates the principle of encapsulation.

We can fix this by implementing a special method called `__setitem__` which allows for items in the class to be mutable. See **Fluent Python** Chapter 11.

```
In [31]: from random import shuffle
```

```
In [32]: deck = FrenchDeck()
```

```
In [33]: deck[:5]
```

```
Out[33]: [Card(rank='2', suit='spades'),
          Card(rank='3', suit='spades'),
          Card(rank='4', suit='spades'),
          Card(rank='5', suit='spades'),
          Card(rank='6', suit='spades')]
```

In [34]:

```
shuffle(deck)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-34-f43911d87fe3> in <module>
----> 1 shuffle(deck)

C:\ProgramData\Anaconda3\lib\random.py in shuffle(self, x, random)
    275                 # pick an element in x[:i+1] with which to exchange x
[i]
    276                 j = randbelow(i+1)
--> 277                 x[i], x[j] = x[j], x[i]
    278         else:
    279             _int = int

TypeError: 'FrenchDeck' object does not support item assignment
```

In [35]:

```
x = list(range(10))
x
```

Out[35]:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [37]:

```
shuffle(x)
x
```

Out[37]:

```
[8, 4, 2, 5, 3, 6, 9, 7, 0, 1]
```

In [38]:

```
class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades hearts diamonds clubs'.split()
    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]
    def __len__(self):
        return len(self._cards)
    def __getitem__(self, position):
        return self._cards[position]
    def __setitem__(self, key, value):
        self._cards[key] = value
```

```
In [39]: deck = FrenchDeck()
```

```
In [40]: deck[:5]
```

```
Out[40]: [Card(rank='2', suit='spades'),
          Card(rank='3', suit='spades'),
          Card(rank='4', suit='spades'),
          Card(rank='5', suit='spades'),
          Card(rank='6', suit='spades')]
```

```
In [45]: shuffle(deck)
```

```
In [46]: deck[:5]
```

```
Out[46]: [Card(rank='2', suit='hearts'),
          Card(rank='3', suit='hearts'),
          Card(rank='Q', suit='spades'),
          Card(rank='5', suit='spades'),
          Card(rank='J', suit='clubs')]
```

The special method, `__setitem__` uses takes two additonal arguments to self: `key` and `value`.

When we call `shuffle`, shuffle implements this assignment system to alter the values in `_cards`

You can see more special methods that are used with container types.

https://docs.python.org/3/reference/datamodel.html#emulating-container-types (https://docs.python.org/3/reference/datamodel.html#emulating-container-types)

```
In [ ]: # quiz answers
```

```
In [58]: letters = list('abcde')
         choice(letters)

Out[58]: 'a'
```

```
In [59]: choice(letters)

Out[59]: 'd'
```

```
In [ ]:
```