Lesson 6 - Thomas Deneuville

October 13, 2017

1 `collections` module

We will cover a few of the objects from the `collections` module. Specifically, we will cover:

- `Counter` object
- `defaultdict` object
- `OrderedDict` object
- `namedtuple` object

1.1 `Counter` object

A very powerful object in Python. It accepts an iterable or a dictionary and it converts it into an object that counts the frequency of each unique value / instance happening.

```
In [1]: from collections import Counter

        A = [1, 2, 3, 3, 4, -1, 0, 2, 2, 2, 3]
        C = Counter(A) # Creates a Counter object that counts how many times
                       # you have seen each value
        print(C)
```

```
Counter({2: 4, 3: 3, 0: 1, 1: 1, 4: 1, -1: 1})
```

Think of this as a "dictionary" where the keys are unique values in an iterable (list, set, etc.) and the value is the number of times you have seen that value. This also works for strings or anything that you can iterate over. For strings, `Counter` is case-sensitive so words with different capitalization or spelling count as a separate instance.

```
In [3]: B = ["Hello", "Hello", "how", "are", "you", "you", "today?", "hello"]
        D = Counter(B)
        print(D)
```

```
Counter({'you': 2, 'Hello': 2, 'how': 1, 'are': 1, 'today?': 1, 'hello': 1})
```

```
In [4]: E = 'aaabbbccdefadsfsvas'
        F = Counter(E)
        print(F)
```

```
Counter({'a': 5, 's': 3, 'b': 3, 'f': 2, 'c': 2, 'd': 2, 'e': 1, 'v': 1})
```

We can access the individual counts of each unique value in our `Counter`. Use like a normal dictionary. If we haven't seen a value before, we return 0.

```
In [7]: print(F['a'])
        print(F['s'])
        print(F['testtest'])
```

```
5
3
0
```

We can merge 2 `Counters` together so that one of them is the combined version of the 2. We can initialize a `Counter` multiple ways. You can provide a dictionary, or key-value pairs.

```
In [14]: X = Counter(hello=4, how=5, a=6)
         Y = Counter({1: 2, 2: 5, 'hi':6, 'a':7, 'how':6})
         print(X)
         print(Y)
```

```
Counter({'a': 6, 'how': 5, 'hello': 4})
Counter({'a': 7, 'hi': 6, 'how': 6, 2: 5, 1: 2})
```

We can add two `Counters` together by using the `update` method.

```
In [15]: X.update(Y) # This updates X with the contents of Y
         print(X)
```

```
Counter({'a': 13, 'how': 11, 'hi': 6, 2: 5, 'hello': 4, 1: 2})
```

We can also **subtract** two `Counters` together. Use the `subtract` method

```
In [16]: print(Y)
         X.subtract(Y)
         print(X)
```

```
Counter({'a': 7, 'hi': 6, 'how': 6, 2: 5, 1: 2})
Counter({'a': 6, 'how': 5, 'hello': 4, 'hi': 0, 2: 0, 1: 0})
```

You can remove values from the `Counter`. Just use `del`

```
In [17]: del X['a']
         print(X)
```

```
Counter({'how': 5, 'hello': 4, 'hi': 0, 2: 0, 1: 0})
```

You can use `most_common` and specify the value `n` that tells you the top `n` most occurring values in your `Counter`.

```
In [18]: print(X.most_common(2)) # Specify the 2 highest occurring elements in the input
```

```
[('how', 5), ('hello', 4)]
```

What is returned is a list of tuples where the first element is the value encountered and the second element is how many times we have seen that value.

```
In [19]: X.update(Counter({'are': 8, 'you': 5}))
```

```
In [26]: print(X)
         print(X.most_common(5))
```

```
Counter({'are': 8, 'how': 5, 'you': 5, 'hello': 4, 'hi': 0, 2: 0, 1: 0})
[('are', 8), ('how', 5), ('you', 5), ('hello', 4), ('hi', 0)]
```

The frequencies of occurrence are sorted in descending order, and we return the `n` beginning resulting elements of the sorted results. If we have counts that are the same, we don't know which one will be returned but when we run this multiple times, that same one will be returned.

It may be useful to determine what the actual values are, not the counts. You would use the `elements` method to display those. We display each "key" for as many times as we have seen it. Any values that are 0 do not get displayed.

```
In [29]: print(list(X.elements())) # The values 2 and 1 are not displayed because their counts
```

```
['how', 'how', 'how', 'how', 'how', 'hello', 'hello', 'hello', 'hello', 'you', 'you', 'you', '
```

2 defaultdict

Very much like a dictionary but we can control what happens when you specify a key that does not exist in the dictionary. When you create a `defaultdict` you specify a function or object that would get returned if you specify an invalid key. If we don't specify this behaviour, it behaves like a normal dictionary which will produce a `KeyError` if a non-existing key is trying to be accessed.

```
In [34]: from collections import defaultdict

         # Specify an anonymous function WITH NO INPUTS
         # such that any key that comes in, you return the value 0
         d = defaultdict(lambda: 0)

         d['key1'] = [1, 2, 3, 4]
         d['key2'] = (1, 2, 3)
         d['hello'] = 'string'

         print(d)
         print(d['testastsdtat'])

         # We can also change the behaviour so the output is an
         # empty list
         e = defaultdict(list)
         e['hello'] = 1
         print(e)
         print(e['teasdtsdt'])

         f = defaultdict(lambda: 'Error')
         f['test'] = 1
         print(f)
         print(f['adwawetwe'])
```

```
defaultdict(<function <lambda> at 0x00000000056DB8C8>, {'key1': [1, 2, 3, 4], 'key2': (1, 2, 3
0
defaultdict(<class 'list'>, {'hello': 1})
[]
defaultdict(<function <lambda> at 0x00000000056DBA60>, {'test': 1})
Error
```

3 OrderedDict

Like a dictionary but it remembers the order of how you have inserted into it. The difference between this and a dictionary in terms of performance is that the `OrderedDict` will be slower because it has to remember the order of insertion. With dictionaries in general, the order is random on purpose due to performance (using Bloom filters).

```
In [40]: from collections import OrderedDict

         A = OrderedDict()
         A['key'] = 'yes'
         A[1] = 78
         A['hello'] = 0
         A[-1] = -90
         A[(1, 2, 3)] = 'hello'
         print(A)

         for key in A:
             print(key)`
```

```
OrderedDict([('key', 'yes'), (1, 78), ('hello', 0), (-1, -90), ((1, 2, 3), 'hello')])
key
```

```
1
hello
-1
(1, 2, 3)
```

4 namedtuple

Very much like a `tuple`, but we can also assign names to the values in the tuple on top of accessing by index. To create this, you first specify what the name for the `tuple` is and you specify a list of variables / fields that you want the `tuple` to contain.

```
In [46]: from collections import namedtuple

         # Create a tuple of type Point that contains two fields
         # x and y
         H = namedtuple('Point', ['x', 'y'])

         test = H(9, 8)
         print(test)
         test2 = H(y=10, x=-1)
         print(test2)

         # To access elements, you can use either the
         # variable name or you can use the position
         # of the tuple element
         print(test.x)
         print(test[0])

         print(test2.y)
         print(test2[1])
```

```
Point(x=9, y=8)
Point(x=-1, y=10)
9
9
10
10
```

To access elements in your `namedtuple`, you can either use the position in the `tuple` or you can use the actual name by ensuring that you place a dot (.) after the `namedtuple` instance and then the variable you want to access. The list of variables you put into `namedtuple` the order is maintained, so each position in your tuple corresponds to the same order as the list of variables put inside the creation of `namedtuple`. We also cannot change any contents of the tuple once it's created so the benefits of a tuple are also embodied.

```
In [47]: test2.y = -10000
```

```
    ---------------------------------------------------------------------------

    AttributeError                            Traceback (most recent call last)

    <ipython-input-47-3711e9d02612> in <module>()
----> 1 test2.y = -10000


    AttributeError: can't set attribute
```