Collections Module

The collections module is a built-in module that implements specialized container data types providing alternatives to Python's general purpose built-in containers. We've already gone over the basics: dict, list, set, and tuple.

Now we'll learn about the alternatives that the collections module provides.

Counter

Counter is a *dict* subclass which helps count hash-able objects. Inside of it elements are stored as dictionary keys and the counts of the objects are stored as the value.

Lets see how it can be used:

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

Counter() with lists

```
In [2]: l = [1,2,2,2,2,3,3,3,1,2,1,12,3,2,32,1,21,1,223,1]

        Counter(l)
```

```
Out[2]: Counter({1: 6, 2: 6, 3: 4, 32: 1, 12: 1, 21: 1, 223: 1})
```

Counter with strings

```
In [3]: Counter('aabsbsbsbhshhbbsbs')
```

```
Out[3]: Counter({'b': 7, 's': 6, 'h': 3, 'a': 2})
```

Counter with words in a sentence

```
In [4]: s = 'How many times does each word show up in this sentence word times each ea
        ch word'

        words = s.split()

        Counter(words)
```

```
Out[4]: Counter({'word': 3, 'each': 3, 'times': 2, 'show': 1, 'this': 1, 'many': 1,
        'in': 1, 'up': 1, 'How': 1, 'does': 1, 'sentence': 1})
```

In [5]:
```
# Methods with Counter()
c = Counter(words)

c.most_common(2)
```

Out[5]:
```
[('word', 3), ('each', 3)]
```

Common patterns when using the Counter() object

In [ ]:
```
sum(c.values())                 # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular dictionary
c.items()                       # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]       # n least common elements
c += Counter()                  # remove zero and negative counts
```

defaultdict

defaultdict is a dictionary like object which provides all methods provided by dictionary but takes first argument (default_factory) as default data type for the dictionary. Using defaultdict is faster than doing the same using dict.set_default method.

A defaultdict will never raise a KeyError. Any key that does not exist gets the value returned by the default factory.

In [12]:
```
from collections import defaultdict
```

In [14]:
```
d = {}
```

In [22]:
```
d['one']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-22-07706fc5dc20> in <module>()
----> 1 d['one']

KeyError: 'one'
```

In [23]:
```
d  = defaultdict(object)
```

In [24]:
```
d['one']
```

Out[24]:
```
<object at 0x1002c3a50>
```

In [26]:
```
for item in d:
    print item
```

```
one
```

Can also initialize with default values:

In [27]:
```
d = defaultdict(lambda: 0)
```

In [28]:
```
d['one']
```

Out[28]:
```
0
```

OrderedDict

An OrderedDict is a dictionary subclass that remembers the order in which its contents are added.

Fro example a normal dictionary:

In [32]:
```
print 'Normal dictionary:'

d = {}

d['a'] = 'A'
d['b'] = 'B'
d['c'] = 'C'
d['d'] = 'D'
d['e'] = 'E'

for k, v in d.items():
    print k, v
```

```
Normal dictionary:
a A
c C
b B
e E
d D
```

An Ordered Dictionary:

In [33]:
```
print 'OrderedDict:'

d = collections.OrderedDict()

d['a'] = 'A'
d['b'] = 'B'
d['c'] = 'C'
d['d'] = 'D'
d['e'] = 'E'

for k, v in d.items():
    print k, v
```

```
OrderedDict:
a A
b B
c C
d D
e E
```

Equality with an Ordered Dictionary

A regular dict looks at its contents when testing for equality. An OrderedDict also considers the order the items were added.

A normal Dictionary:

In [36]:
```
print 'Dictionaries are equal? '

d1 = {}
d1['a'] = 'A'
d1['b'] = 'B'

d2 = {}
d2['b'] = 'B'
d2['a'] = 'A'

print d1 == d2
```

```
Dictionaries are equal?
True
```

An Ordered Dictionary:

In [37]:
```
print 'Dictionaries are equal? '

d1 = collections.OrderedDict()
d1['a'] = 'A'
d1['b'] = 'B'


d2 = collections.OrderedDict()

d2['b'] = 'B'
d2['a'] = 'A'

print d1 == d2
```

```
Dictionaries are equal?
False
```

namedtuple

The standard tuple uses numerical indexes to access its members, for example:

In [38]:
```
t = (12,13,14)
```

In [39]:
```
t[0]
```

Out[39]: 12

For simple use cases, this is usually enough. On the other hand, remembering which index should be used for each value can lead to errors, especially if the tuple has a lot of fields and is constructed far from where it is used. A namedtuple assigns names, as well as the numerical index, to each member.

Each kind of namedtuple is represented by its own class, created by using the namedtuple() factory function. The arguments are the name of the new class and a string containing the names of the elements.

You can basically think of namedtuples as a very quick way of creating a new object/class type with some attribute fields. For example:

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

In [47]:
```
Dog = namedtuple('Dog','age breed name')

sam = Dog(age=2,breed='Lab',name='Sammy')

frank = Dog(age=2,breed='Shepard',name="Frankie")
```

We construct the namedtuple by first passing the object type name (Dog) and then passing a string with the variety of fields as a string with spaces between the field names. We can then call on the various attributes:

```
In [42]: sam
```

```
Out[42]: Dog(age=2, breed='Lab', name='Sammy')
```

```
In [43]: sam.age
```

```
Out[43]: 2
```

```
In [44]: sam.breed
```

```
Out[44]: 'Lab'
```

```
In [45]: sam[0]
```

```
Out[45]: 2
```

Conclusion

Hopefully you now see how incredibly useful the collections module is in Python and it should be your go-to module for a variety of common tasks!