The `collections` module

- the **`collections`** module contains a bunch of useful types which are derived from (read: inherited from) some of the built-in types we're already familiar with

Ordered Dictionaries

- ordered dictionaries are dictionaries which retain their insertion order, i.e., the order in which you insert the items is in the order in which you iterate through them

In [1]:

```
%%python2
# dictionaries did not retain insertion order prior to Python 3.6
d = {}
d['one'] = 3
d['two'] = 6
d['three'] = 0
print(d)
```

```
{'three': 0, 'two': 6, 'one': 3}
```

In [2]:

```
%%python2
from collections import OrderedDict
d = OrderedDict() # {}
d['one'] = 3
d['two'] = 6
d['three'] = 0
print(d)
```

```
OrderedDict([('one', 3), ('two', 6), ('three', 0)])
```

In [3]:

```
%%python2
from collections import OrderedDict
d = OrderedDict()
d['one'] = 3
d['two'] = 6
d['three'] = 0
```

```
for k, v in d.items():
    print('%s => %s' % (k, v))
```

```
one => 3
two => 6
three => 0
```

In [4]:

```
# Python 3.6 dicts retain insertion order by default
# see https://mail.python.org/pipermail/python-dev/2016-September/146327.html
d = {}
d['one'] = 3
d['two'] = 6
d['three'] = 0
print(d)
```

```
{'one': 3, 'two': 6, 'three': 0}
```

In [5]:

```
# OrderedDict less useful in Python 3.6, but it does have a
# new method...
from collections import OrderedDict
d = OrderedDict()
d['one'] = 3
d['two'] = 6
d['three'] = 0
print(d)
```

```
OrderedDict([('one', 3), ('two', 6), ('three', 0)])
```

In [6]:

```
d.move_to_end('one')
d
```

Out[6]:

```
OrderedDict([('two', 6), ('three', 0), ('one', 3)])
```

In [7]:

```
d.move_to_end('one', False)
d
```

Out[7]:

```
OrderedDict([('one', 3), ('two', 6), ('three', 0)])
```

The `collections` module: Default Dictionaries

Default Dictionaries

- suppose we need a default value for any key which does not exist in the dictionary
  - we can use the **`get()`** function, or **`setdefault()`** (or the **`in`** operator), or we can use a `Default Dictionary`

In [8]:

```python
# what we did before...

def count_letters(word):
    '''Returns a dict of letters and how many times the letter
    appeared in the word passed in'''
    count = {}
    for ltr in word:
        #count[ltr] = count.setdefault(ltr, 0) + 1
        count[ltr] = count.get(ltr, 0) + 1
    return count

count_letters('antidisestablishmentarianism')
```

Out[8]:

```
{'a': 4,
 'n': 3,
 't': 3,
 'i': 5,
 'd': 1,
 's': 4,
 'e': 2,
 'b': 1,
 'l': 1,
 'h': 1,
 'm': 2,
 'r': 1}
```

In [9]:

```python
from collections import defaultdict

def count_letters(word):
    '''Returns a dict of letters and how many times the letter
    appeared in the string passed in.'''
    # When creating a defaultdict,
    # the passed argument dictates what the
    # default value will be (int = 0, str = "", list = [])
    count = defaultdict(int)
    for ltr in word:
        count[ltr] += 1
    return count
```

```
count_letters('one two three four two three three')
```

Out[9]:

```
defaultdict(int,
            {'o': 4,
             'n': 1,
             'e': 7,
             ' ': 6,
             't': 5,
             'w': 2,
             'h': 3,
             'r': 4,
             'f': 1,
             'u': 1})
```

Lab: Default Dictionaries

- read from a file where each line is a word followed by a count, e.g.,

  ```
  apple 2
  pear 3
  cherry 5
  apple 3
  pear 6
  apple 1
  ```

  (as shown above, words may be duplicated)
- generate a **`defaultdict`** where the keys are the words and the value are a *list* of all the counts for that word, e.g., `defaultdict(<class 'list'>, {'apple': ['2', '3', '1'], 'pear': ['3', '6'], 'cherry': ['5']})`

Now, for more fun, let's implement a default dictionary without using the `collections` module

- In other words, make your own class (e.g., MyDefaultDict)
- What class or classes should it inherit from?
- You will need to create the method `__getitem__(self, key)__` which is what Python uses under the hood to retrieve an item from a dictionary
  - if the key in question is not currenty in the dict, what should you return?

The `collections` module: Deque

Deque

- double ended queue
- pronounced "deck"

In [10]:

```
from collections import deque
dq = deque(range(10), maxlen=10) # maxlen is optional
print(dq)
```

```
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
```

In [11]:

```
dq.rotate(3) # +n takes items from right, prepends to left, vice versa for -n
print(dq)
```

```
deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)
```

In [12]:

```
dq.rotate(-4)
print(dq)
```

```
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], maxlen=10)
```

In [13]:

```
dq.appendleft('a') # appending to full deque discards item(s) from other end
print(dq)
```

```
deque(['a', 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
```

In [14]:

```
dq.extend('bcd')
print(dq)
```

```
deque([3, 4, 5, 6, 7, 8, 9, 'b', 'c', 'd'], maxlen=10)
```

In [15]:

```
dq.extendleft((-1, -2, -3))
print(dq)
```

```
deque([-3, -2, -1, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
```

In [16]:

```
dq.pop() # same as list
```

Out[16]:

```
9
```

In [17]:

```
dq.popleft() # specific to deque, as is rotate()
```

Out[17]:

```
-3
```

In [18]:

```
print(dq)
dq.remove(3) # same as list
print(dq)
```

```
deque([-2, -1, 3, 4, 5, 6, 7, 8], maxlen=10)
deque([-2, -1, 4, 5, 6, 7, 8], maxlen=10)
```

In [19]:

```
dq.reverse()
print(dq)
```

```
deque([8, 7, 6, 5, 4, -1, -2], maxlen=10)
```

In [20]:

```
dq.append(0)
dq
```

Out[20]:

```
deque([8, 7, 6, 5, 4, -1, -2, 0])
```

Lab: Deque

- use a deque to print the last *n* lines of file, much like **`tail`** in Linux
- remember that you can iterate through a file a line at a time

The `collections` module: Named Tuples

Named Tuples

- tuples are quite handy, but they are missing a key feature when using them as records–sometimes we want to name the fields
    - more efficient (i.e., less memory) than dictionaries because instances don't need to contain the keys themselves, as dictionaries do, just the values
- **`namedtuple()`** returns not an individual object but a new class, customized for the given names

In [21]:

```python
from collections import namedtuple
Point = namedtuple('Point', 'x y')
# first argument is the name of the tuple class itself
# second argument is attribute names as an iterable of strings or a
# single space/comma-delimited string
point1 = Point(1, 3)
print(point1, type(point1))
```

```
Point(x=1, y=3) <class '__main__.Point'>
```

In [22]:

```python
point2 = Point(-3, -2)
print(point2)
print(point1[0], point2[1]) # what we would do if just a tuple
```

```
Point(x=-3, y=-2)
1 -2
```

In [23]:

```python
print(point1.x, point1.y) # much nicer, because fields are named
```

```
1 3
```

In [24]:

```python
from collections import namedtuple
City = namedtuple('City', 'name country population coordinates')
tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
print(tokyo)
```

```
City(name='Tokyo', country='JP', population=36.933, coordinates=(35.689722, 139.691667))
```

In [25]:

```python
print(tokyo.population) # Prefer to use attribute or field names
print(tokyo.coordinates)
print(tokyo[1]) # use indexing if I wish
```

```
36.933
(35.689722, 139.691667)
JP
```

In [26]:

```
type(City), type(tokyo)
```

Out[26]:

```
(type, __main__.City)
```

In [27]:

```
for field in City._fields: # tuple containing field names
    print(field)
```

```
name
country
population
coordinates
```

In [28]:

```
LatLong = namedtuple('LatLong', 'lat long')
delhi_data = ('Delhi NCR', 'IN', 21.935,
              LatLong(28.613889, 77.2098889)) # tuple
```

In [29]:

```
delhi = City._make(delhi_data)
delhi
```

Out[29]:

```
City(name='Delhi NCR', country='IN', population=21.935, coordinates=LatLong(lat=28.613889, long=77.2098889))
```

In [30]:

```
delhi2 = City(*delhi_data)
delhi2
```

Out[30]:

```
City(name='Delhi NCR', country='IN', population=21.935, coordinates=LatLong(lat=28.613889, long=77.2098889))
```

In [31]:

```
delhi == delhi2
```

Out[31]:

```
True
```

In [32]:

```
d = delhi._asdict() # returns an OrderedDict built from named tuple
print(d)
```

```
{'name': 'Delhi NCR', 'country': 'IN', 'population': 21.935, 'coordinates': LatLong(lat=28.613889, long=77.2098889)}
```

Lab: Named Tuples

1. Create a named tuple called **`Card`** (representing a playing card) which has two fields, **`rank`** and **`suit`**
2. Create a list of **`Card`** s, which, when initialized, contains all 52 cards in a deck
3. In other words, the list (or deck) should contain

`[Card(rank=2, suit='clubs'), Card(rank=3, suit='clubs'), Card(rank=4, suit='clubs'), ..., Card(rank='Q', suit='spades'), Card(rank='K', suit='spades'), Card(rank='A', suit='spades')]`

In [ ]:

```python

```

The `collections` module: Counters

Counters

- **`dict`** subclass for counting things
- unordered collection where things being counted are `dict` keys and the counts are `dict` values
- **`Counters`** can have negative values

In [33]:

```python
from collections import Counter
c = Counter()
c
```

Out[33]:

```
Counter()
```

In [34]:

```python
c = Counter('antidisestablishmentarianism')
c
```

Out[34]:

```
Counter({'a': 4,
         'n': 3,
         't': 3,
         'i': 5,
         'd': 1,
         's': 4,
         'e': 2,
         'b': 1,
         'l': 1,
         'h': 1,
         'm': 2,
         'r': 1})
```

In [35]:

```python
c.update('establish' * 10)
c
```

Out[35]:

```
Counter({'a': 14,
         'n': 3,
         't': 13,
         'i': 15,
         'd': 1,
         's': 24,
         'e': 12,
         'b': 11,
         'l': 11,
         'h': 11,
         'm': 2,
         'r': 1})
```

In [36]:

```python
c = Counter({'red': 5, 'blue': -1})
c
```

Out[36]:

```
Counter({'red': 5, 'blue': -1})
```

In [37]:

```python
c = Counter(foo=1, bar=2)
c
```

Out[37]:

```
Counter({'foo': 1, 'bar': 2})
```

In [38]:

```python
c = Counter(red=6, blue=5, green=3, pink=1,
            yellow=-3)
c.elements() # returns an iterator
```

Out[38]:

```
<itertools.chain at 0x7f3fb04d07f0>
```

In [39]:

```python
for thing in c.elements(): # cf. list(...)
    print(thing, end=' ')
```

```
red red red red red red blue blue blue blue blue green green green pink
```

In [40]:

```
c.most_common(3) # returns the n most common elements
```

Out[40]:

```
[('red', 6), ('blue', 5), ('green', 3)]
```

In [41]:

```
d = Counter(fuschia=3, pink=0, red=3, blue=5, green=2)
c.subtract(d) # preserves negative values
c
```

Out[41]:

```
Counter({'red': 3,
         'blue': 0,
         'green': 1,
         'pink': 1,
         'yellow': -3,
         'fuschia': -3})
```

In [42]:

```
c.items() # remember that under the hood, this is a dict
```

Out[42]:

```
dict_items([('red', 3), ('blue', 0), ('green', 1), ('pink', 1), ('yellow', -3), ('fuschia', -3)])
```

In [43]:

```
+c # generates new Counter, discarding 0s or negatives
```

Out[43]:

```
Counter({'red': 3, 'green': 1, 'pink': 1})
```

In [44]:

```
c = +c
c
```

Out[44]:

```
Counter({'red': 3, 'green': 1, 'pink': 1})
```

In [45]:

```
c = Counter(red=6, blue=-5, green=3, pink=1, yellow=-3)
c = -c # discard positives and multiply remaining negatives by -1
c
```

Out[45]:

```
Counter({'blue': 5, 'yellow': 3})
```

In [46]:

```
d = Counter(red=6, yellow=7, green=9)
c.update(d)
c
```

Out[46]:

```
Counter({'blue': 5, 'yellow': 10, 'red': 6, 'green': 9})
```

In [47]:

```
c = Counter(a=3, b=1, c=4)
d = Counter(a=1, b=2, c=5)
c + d
```

Out[47]:

```
Counter({'a': 4, 'b': 3, 'c': 9})
```

In [48]:

```
c - d
```

Out[48]:

```
Counter({'a': 2})
```

In [49]:

```
print(c, d, sep='\n')
```

```
Counter({'c': 4, 'a': 3, 'b': 1})
Counter({'c': 5, 'b': 2, 'a': 1})
```

Lab: Counters

- Use a `Counter` to count the words in a file
- That is, read in a file, separate it into words, and use a `Counter` to count the number of occurrences of each word in the file.
- Print out the 10 most common words in the file