Becoming a Master Collector

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). He is based in Chicago, where he also teaches a variety of Python courses.
dave@dabeaz.com

As a Python programmer, you know that lists, sets, and dictionaries are useful for collecting data. For example, you use a list whenever you want to store data and keep it in order:

```
>>> names = ['Dave', 'Paula', 'Thomas', 'Lewis']
>>>
```

If you simply want a collection of unique items and don't care about the order, you can make a set:

```
>>> colors = set(['red','blue','green','purple','yellow'])
>>>
```

You use a dictionary whenever you want to make key-value lookup tables:

```
>>> prices = { 'AAPL' : 613.20, 'ACME' : 71.23, 'IBM' : 174.11 }
>>> prices['AAPL']
613.20
>>>
```

Using just these three primitives, you can build just about any other data structure in the known universe. However, why would you? In this article, we reach into Python's collections library and look at some of the tools it provides for manipulating collections of data. If you're like me, these will quickly become a part of your day-to-day programming.

Tabulating Data

How many times have you ever needed to tabulate data or build a histogram? For example, suppose you want to tabulate and count all of the IP addresses that made requests on your Web site from a server log such as this:

```
78.192.56.97 - - [15/Mar/2012:01:50:37 -0500] "GET /ply/ HTTP/1.1" 200 11875
69.237.118.150 - - [15/Mar/2012:01:51:52 -0500] "GET /ply/ply.html HTTP/1.1"
200 107623
69.237.118.150 - - [15/Mar/2012:01:51:57 -0500] "GET /ply/example.html
HTTP/1.1" 200 2393
91.35.214.71 - - [15/Mar/2012:01:52:13 -0500] "GET /ply/ HTTP/1.1" 200 11875
91.35.214.71 - - [15/Mar/2012:01:52:13 -0500] "GET /favicon.ico HTTP/1.1" 404
369
```

You might be inclined to write a small fragment of code using a Python dictionary, like this:

```
hits_by_ipaddr = {}
for line in open("access-log"):
        fields = line.split()
        ipaddr = fields[0]
        if ipaddr in hits_by_ipaddr:
           hits_by_ipaddr[ipaddr] += 1
        else:
           hits_by_ipaddr[ipaddr] = 1
```

Although this code "works," it's also a bit clunky. For example, you have to add a special check for first initialization (otherwise the attempt to increment the count will fail with a `KeyError` on first access). On top of that, after you have populated the dictionary, you will probably want to do some further analysis. For example, maybe you want to print a table showing the 25 most common IP addresses in descending order:

```
popular_ips = sorted(hits_by_ipaddr,
        key=lambda x: hits_by_ipaddr[x],
        reverse=True)

for ipaddr in popular_ips[:25]:
        print("%5d: %s" % (hits_by_ipaddr[ipaddr],ipaddr))
```

As output, this will produce a table such as this:

```
 1096: 78.192.56.97
 1040: 206.15.64.54
  473: 212.85.154.246
  226: 89.215.101.39
  209: 212.85.154.254
  185: 82.226.112.70
  180: 78.192.56.101
...
```

Although this code is relatively easy to write, you still need to think about it a bit—especially the tricky sort with the `lambda`. However, you can avoid all of this if you simply use Counter objects from the collections module. Here is a much simplified version of the same code:

```
from collections import Counter

hits_by_ipaddr = Counter()
for line in open("access-log"):
        fields = line.split()
        ipaddr = fields[0]
        hits_by_ipaddr[ipaddr] += 1

for ipaddr, count in hits_by_ipaddr.most_common(25):
        print("%5d: %s" % (count, ipaddr))
```

First added to Python 2.7, Counter objects are perfectly suited for tabulation. They automatically take care of initializing elements on first access. Not only that, they provide useful methods such as most_common([n]) that return the *n* most common items. However, this is really only scratching the surface.

If you want, counters can be automatically initialized from iterables. For example, let's make letter counts from strings:

```
>>> a = Counter("Hello")
>>> b = Counter("World")
>>> a
Counter({'l': 2, 'H': 1, 'e': 1, 'o': 1})
>>> b
Counter({'d': 1, 'r': 1, 'o': 1, 'W': 1, 'l': 1})
>>>
```

Or, if you're inclined and a bit more sophisticated, you can populate a counter from a generator expression:

```
>>> f = open("access-log")
>>> hits_by_ipaddr = Counter(line.split()[0] for line in f)
>>> hits_by_ipaddr['78.192.56.97']
1096
>>>
```

You can also do math with counters:

```
>>> a + b   # Adds counts together
Counter({'l': 3, 'o': 2, 'e': 1, 'd': 1, 'H': 1, 'r': 1, 'W': 1})
>>> a - b   # Takes away counts in b
Counter({'H': 1, 'e': 1, 'l': 1})
>>> b - a   # Takes away counts in a
Counter({'r': 1, 'd': 1, 'W': 1})
>>> a & b   # Minimum counts
Counter({'l': 1, 'o': 1})
>>> a | b   # Maximum counts
Counter({'l': 2, 'e': 1, 'd': 1, 'H': 1, 'o': 1, 'r': 1, 'W': 1})
>>>
```

Adding and subtracting counts are also available in-place using update() and subtract methods, respectively. For example:

```
>>> a = Counter("Hello")
>>> a
Counter({'l': 2, 'H': 1, 'e': 1, 'o': 1})
>>> a.update("World")
>>> a
Counter({'l': 3, 'o': 2, 'e': 1, 'd': 1, 'H': 1, 'r': 1, 'W': 1})
>>>
```

Using some of these techniques, we can refine our script to process an entire directory of log files:

```
from collections import Counter
from glob import glob

hits_by_ipaddr = Counter()

logfiles = glob("*.log")
for filename in logfiles:
        f = open(filename)
```

```
        hits_by_ipaddr.update(line.split()[0] for line in f)
        f.close()

for ipaddr, count in hits_by_ipaddr.most_common(25):
        print("%5d: %s" % (count, ipaddr))
```

By now, hopefully, you've gotten the idea that Counter objects are the way to go for tabulation. Frankly, they're one of my favorite new additions to Python.

Dictionaries with Multiple Values

Normally, dictionaries map a single key to a single value. However, a common question that sometimes arises is how you map a key to multiple values. Naturally, the solution is to map a key to a list or set. For example, suppose you wanted to make a dictionary that mapped URLs to all of the unique IP addresses that accessed it. Here is some code that would do it:

```
url_to_ips = {}
for line in open("access-log"):
        fields = line.split()
        ipaddr = fields[0]
        url = fields[6]
        # Create a set on first access
        if url not in url_to_ips:
           url_to_ips[url] = set()
        url_to_ips[url].add(ipaddr)
```

Again, we are faced with the problem of creating the first entry for each URL (hence, the check that makes the set on first access). We can't use Counter objects here, but not to worry—the defaultdict class is built just for this case. Here is an alternative implementation:

```
from collections import defaultdict
url_to_ips = defaultdict(set)
for line in open("access-log"):
        fields = line.split()
        ipaddr = fields[0]
        url = fields[6]
        url_to_ips[url].add(ipaddr)
```

After running this code, you could do things like find out which IP addresses are likely to be robots:

```
>>> url_to_ips['/robots.txt']
set(['173.11.97.115', '107.20.104.146', '61.135.249.76', ...])
>>>
```

defaultdict is a special Python dictionary that allows you to supply a callable for creating the initial entry to be used on first access. In the above code, we've specified that a `set` be used. Here are some examples to try:

```
>>> from collections import defaultdict
>>> a = defaultdict(set)
>>> a
defaultdict(<type 'set'>, {})
>>> a['x'].add(2)
```

```
>>> a['y'].add(3)
>>> a['x'].add(4)
>>> a
defaultdict(<type 'set'>, {'y': set([3]), 'x': set([2, 4])})
>>>
```

In effect, the function provided to defaultdict is triggered to create the first value whenever a non-existent key is accessed. Here are more examples:

```
>>> a['q']
set()
>>> a['r']
set()
>>> a
defaultdict(<type 'set'>, {'y': set([3]), 'x': set([2, 4]), 'r': set([]), 'q':
set([])})
>>>
```

Notice how entries for 'q' and 'r' were added simply by being referenced.

Underneath the covers, defaultdict uses a little-known special method called __missing__(). It's called on a dictionary whenever you read from a missing key. For example:

```
>>> class mydict(dict):
...     def __missing__(self, key):
...     return 0   # Return the missing value
...
>>> d = mydict()
>>> d['x']
0
>>> d['y']
0
>>>
```

Counter objects are implemented using the __missing__() function shown above. defaultdict objects create the missing value using a user-supplied function.

Dictionaries, Views, and Sets

One of the more subtle improvements to Python over the years has been related to the relationship between dictionaries and sets. In many respects, a set is just a collection of dictionary keys with no values. In fact, the underlying implementation of sets and dictionaries is very similar and shares much of the same code.

Despite their similarities, dictionaries have not traditionally provided a natural way to interact with sets of keys or values. Instead, there are simple methods to return the keys, values, and items as a list:

```
>>> a = { 'x' : 2, 'y' : 3, 'z': 4 }
>>> a.keys()
['y', 'x', 'z']
>>> a.values()
[3, 2, 4]
>>> a.items()
[('y', 3), ('x', 2), ('z', 4)]
>>>
```

Starting with Python 2.7, it is possible to express the keys and values of a dictionary as a "view" (which is also the default behavior of the above methods in Python 3). Unlike a list, a view offers a direct window inside the dictionary implementation. Changes to the underlying dictionary directly change the view:

```
>>> k = a.viewkeys()
>>> k
dict_keys(['y', 'x', 'z'])
>>> v = a.viewvalues()
>>> v
dict_values([3, 2, 4])

>>> # Now change the dictionary and observe how the views change
>>> a['w'] = 5
>>> k
dict_keys(['y', 'x', 'z', 'w'])
>>> v
dict_values([3, 2, 4, 5])
>>>
```

At first glance, it might not be immediately obvious how views are useful. On a superficial level, they support iteration, allowing them to be useful in many of the same ways as having a list. However, one of their unique features is the ability to interact with sets and other sequences more elegantly. To illustrate, here are some simple examples you can try:

```
>>> a = { 'x' : 1, 'y': 2, 'z' : 3 }
>>> b = { 'x' : 4, 'y': 2 }
>>> # Find all keys in common
>>> a.viewkeys() & b.viewkeys()
set(['y', 'x'])

>>> # Iterate over all keys except 'z'
>>> for k in a.viewkeys() - ['z']:
...     print("%s = %s" % (k, a[k]))
...
y = 2
x = 1

>>> # Make a set of all key/value pairs
>>> a.viewitems() | b.viewitems()
set([('z', 3), ('y', 2), ('x', 4), ('x', 1)])
>>>
```

In more practical terms, understanding the nature of views can simplify your code. For example, if you wanted to find all of the IP addresses that accessed your site but didn't look at the robots.txt file, you could simply write this:

```
>>> nonrobots = hits_by_ipaddr.viewkeys() - url_to_ips['/robots.txt']
>>>
```

Other Goodies: Queues, Ring Buffers, and Ordered Dictionaries

The collections module has a variety of other data structures that are also worth a look. For instance, if you ever need to build a queue, use the deque object. A deque is like a list except that it's optimized for insertion and deletion operations on both

ends; in contrast, a list has O(n) performance for operations that insert or delete items from the front of the list:

```
>>> from collections import deque
>>> q = deque()
>>> q.appendleft(1)
>>> q.appendleft(2)
>>> q
deque([2, 1])
>>> q.append(3)
>>> q
deque([2, 1, 3])
>>> q.pop()
3
>>> q.popleft()
2
>>>
```

If you specify a maximum size, a deque turns into a ring-buffer or circular queue:

```
>>> q = deque(maxlen=3)
>>> q.extend([1,2,3])
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)
>>>
```

Last, but not least, there is an OrderedDict class. This is used if you want to store information in a dictionary while preserving its insertion order. This can be useful if you're reading data that you later want to output in the same order in which it was read. For example, suppose you had a file of parameters like this:

```
FILENAME foo.txt
DIRNAME  /users/beazley
MODE    a
```

You could read it into an OrderedDict like this:

```
>>> from collections import OrderedDict
>>> parms = OrderedDict()
>>> for line in open("parms.txt"):
...     name,value = line.split()
...     parms[name] = value
...
>>> p['DIRNAME'] '
/users/beazley'
>>> for p in parms.items():
...     print(p)
... ('FILENAME', 'foo.txt')
```

```
('DIRNAME', '/users/beazley')
('MODE', 'a')
>>>
```

Carefully observe how iterating over the dictionary contents preserves data in the same order as read.

Final Words

If you're using Python to manipulate data, the collections module is definitely worth a look. Even if you've been using Python for a while, the contents of this module have been expanded with each new Python release. In modern Python releases, you might be surprised at what you find.