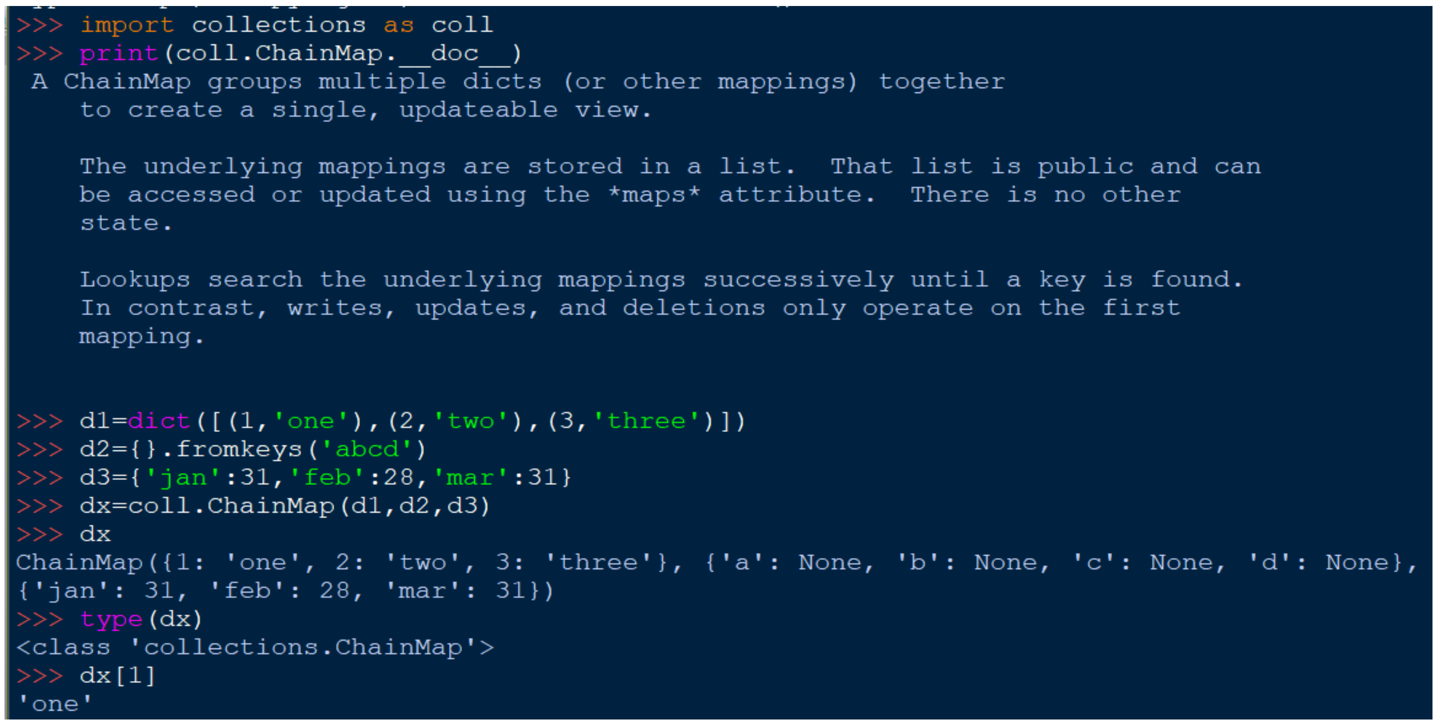
TOPIC: THE COLLECTIONS MODULE

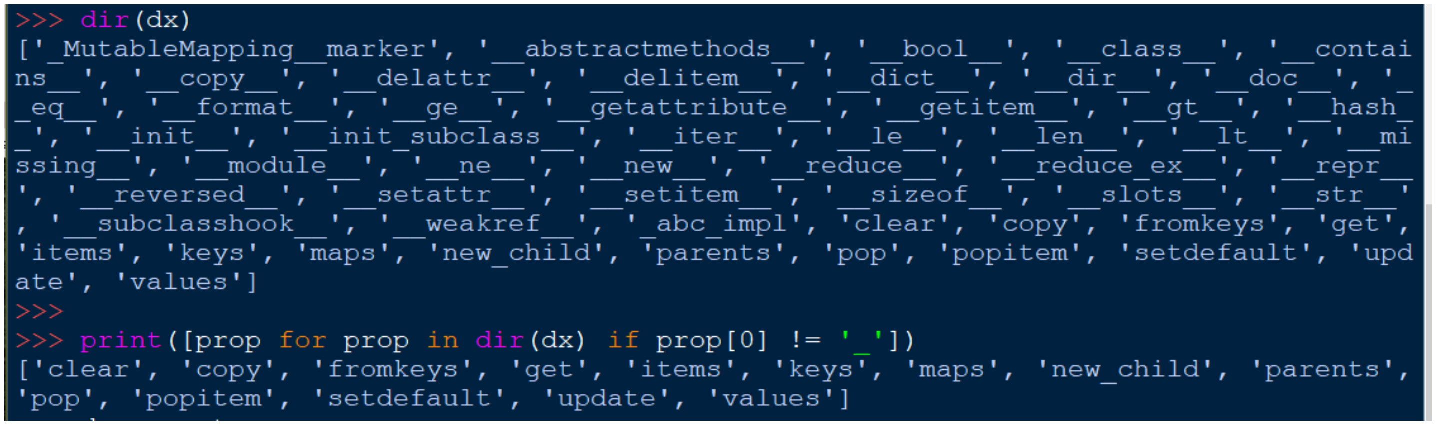
Python’s collections module has specialized container data types that can be used to replace Python’s general purpose containers (dict, tuple, list, and set).

ChainMap

A **ChainMap** is a class that provides the ability **to link multiple mappings(dict) together** such that they end up being a single unit. Similar to the ***zip builtin*** that links multiple ordered collections into a single collection.



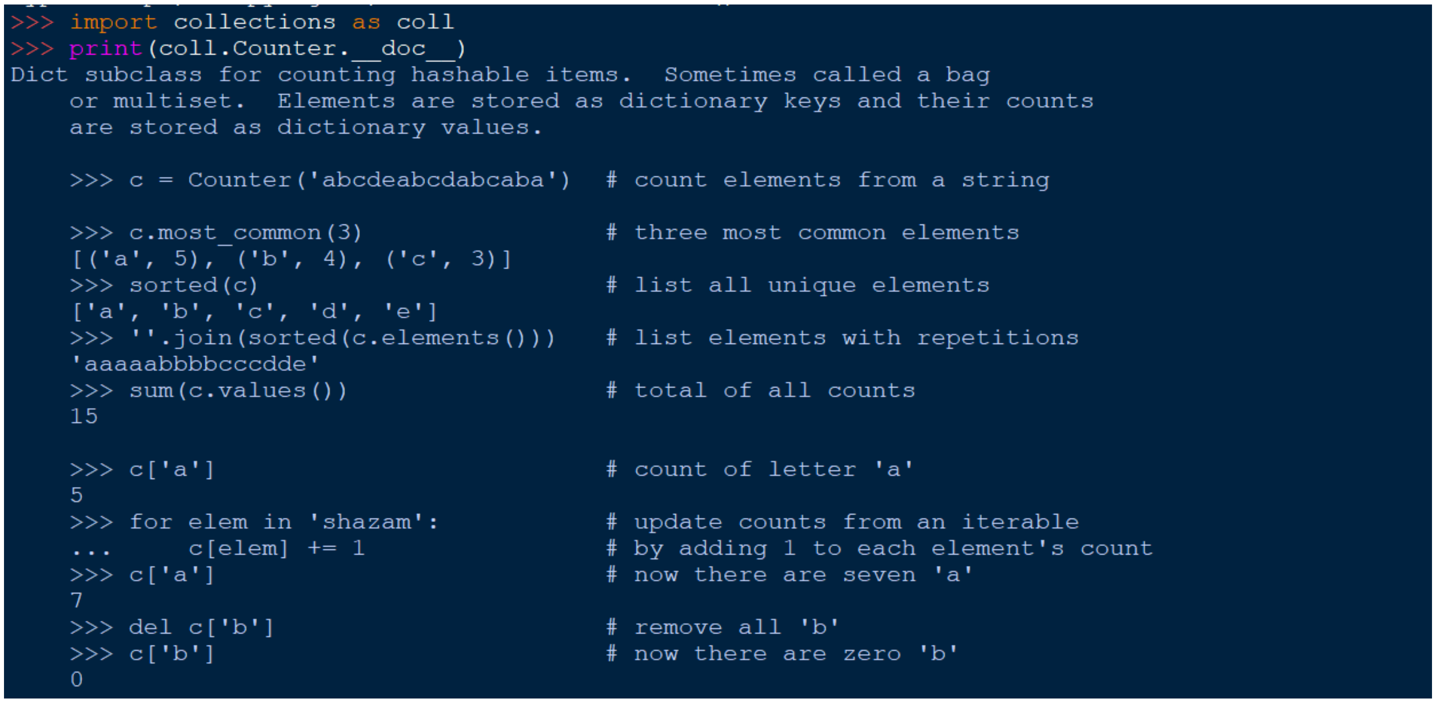
```
>>> import collections as coll
>>> print(coll.ChainMap.__doc__)
 A ChainMap groups multiple dicts (or other mappings) together
    to create a single, updateable view.

    The underlying mappings are stored in a list.  That list is public and can
    be accessed or updated using the *maps* attribute.  There is no other
    state.

    Lookups search the underlying mappings successively until a key is found.
    In contrast, writes, updates, and deletions only operate on the first
    mapping.


>>> d1=dict([(1,'one'),(2,'two'),(3,'three')])
>>> d2={}.fromkeys('abcd')
>>> d3={'jan':31,'feb':28,'mar':31}
>>> dx=coll.ChainMap(d1,d2,d3)
>>> dx
ChainMap({1: 'one', 2: 'two', 3: 'three'}, {'a': None, 'b': None, 'c': None, 'd': None},
{'jan': 31, 'feb': 28, 'mar': 31})
>>> type(dx)
<class 'collections.ChainMap'>
>>> dx[1]
'one'
```



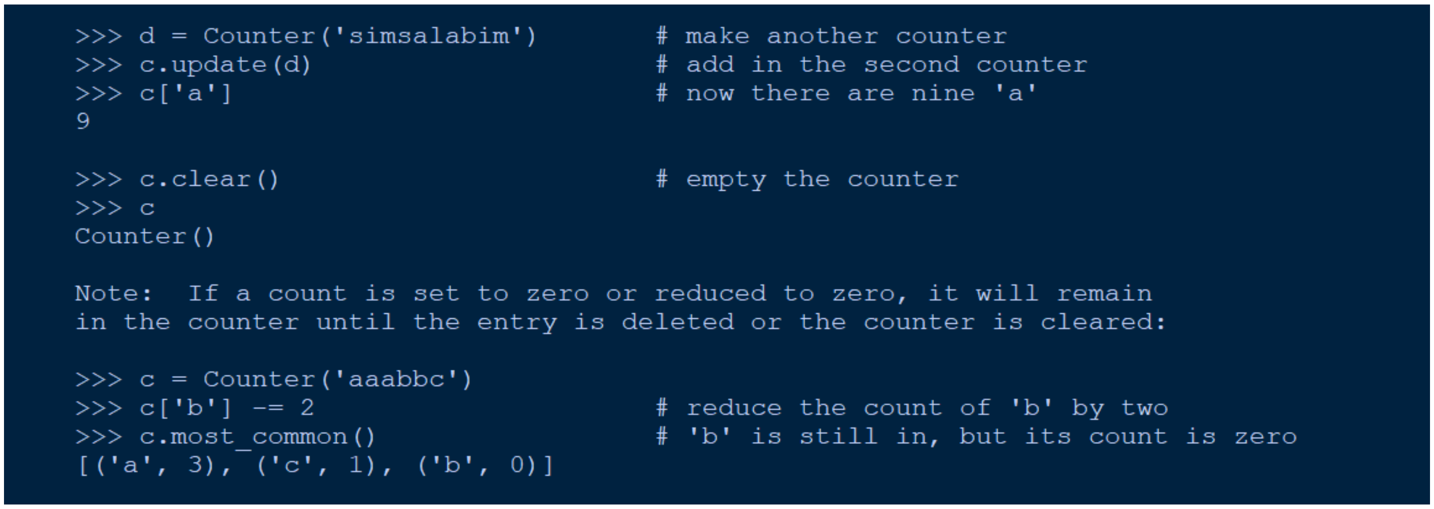
```
>>> dir(dx)
['_MutableMapping__marker', '__abstractmethods__', '__bool__', '__class__', '__contai
ns__', '__copy__', '__delattr__', '__delitem__', '__dict__', '__dir__', '__doc__', '_
_eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash_
_', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mi
ssing__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__
', '__reversed__', '__setattr__', '__setitem__', '__sizeof__', '__slots__', '__str__'
, '__subclasshook__', '__weakref__', '_abc_impl', 'clear', 'copy', 'fromkeys', 'get',
'items', 'keys', 'maps', 'new_child', 'parents', 'pop', 'popitem', 'setdefault', 'upd
ate', 'values']
>>>
>>> print([prop for prop in dir(dx) if prop[0] != '_'])
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'maps', 'new_child', 'parents',
'pop', 'popitem', 'setdefault', 'update', 'values']
```

Counter

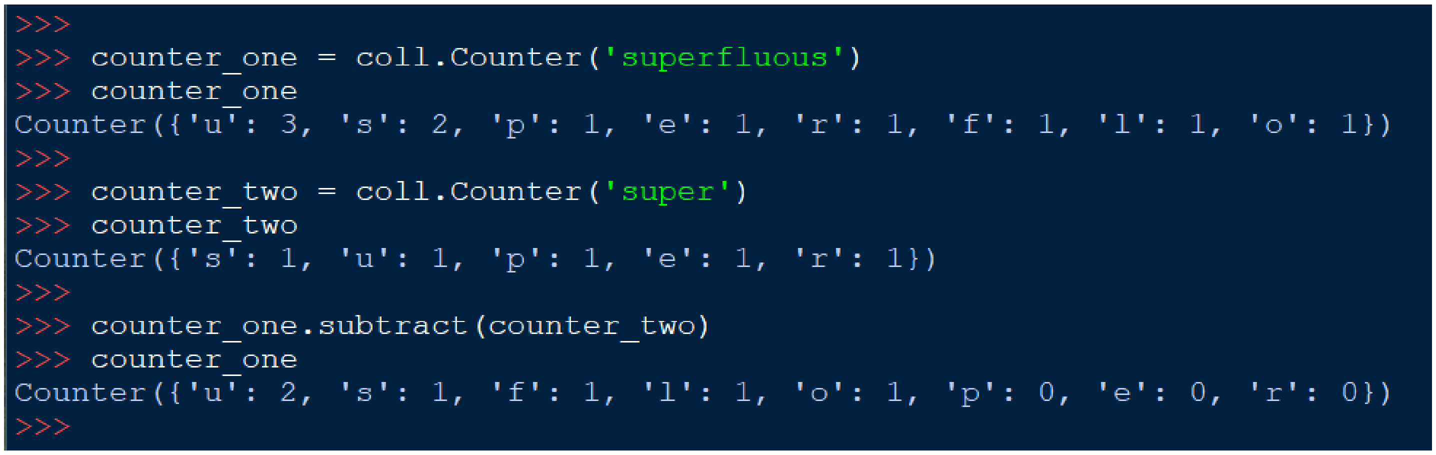
The collections module also provides us with a neat little tool that supports convenient and fast tallies. This tool is called **Counter**. You can run it against most iterables.



```
>>> import collections as coll
>>> print(coll.Counter.__doc__)
Dict subclass for counting hashable items.  Sometimes called a bag
    or multiset.  Elements are stored as dictionary keys and their counts
    are stored as dictionary values.

    >>> c = Counter('abcdeabcdabcaba')  # count elements from a string

    >>> c.most_common(3)                # three most common elements
    [('a', 5), ('b', 4), ('c', 3)]
    >>> sorted(c)                       # list all unique elements
    ['a', 'b', 'c', 'd', 'e']
    >>> ''.join(sorted(c.elements()))   # list elements with repetitions
    'aaaaabbbbcccdde'
    >>> sum(c.values())                 # total of all counts
    15

    >>> c['a']                          # count of letter 'a'
    5
    >>> for elem in 'shazam':           # update counts from an iterable
    ...     c[elem] += 1                # by adding 1 to each element's count
    >>> c['a']                          # now there are seven 'a'
    7
    >>> del c['b']                      # remove all 'b'
    >>> c['b']                          # now there are zero 'b'
    0
```



```
    >>> d = Counter('simsalabim')       # make another counter
    >>> c.update(d)                     # add in the second counter
    >>> c['a']                          # now there are nine 'a'
    9

    >>> c.clear()                       # empty the counter
    >>> c
    Counter()

    Note:  If a count is set to zero or reduced to zero, it will remain
    in the counter until the entry is deleted or the counter is cleared:

    >>> c = Counter('aaabbc')
    >>> c['b'] -= 2                     # reduce the count of 'b' by two
    >>> c.most_common()                 # 'b' is still in, but its count is zero
    [('a', 3), ('c', 1), ('b', 0)]
```

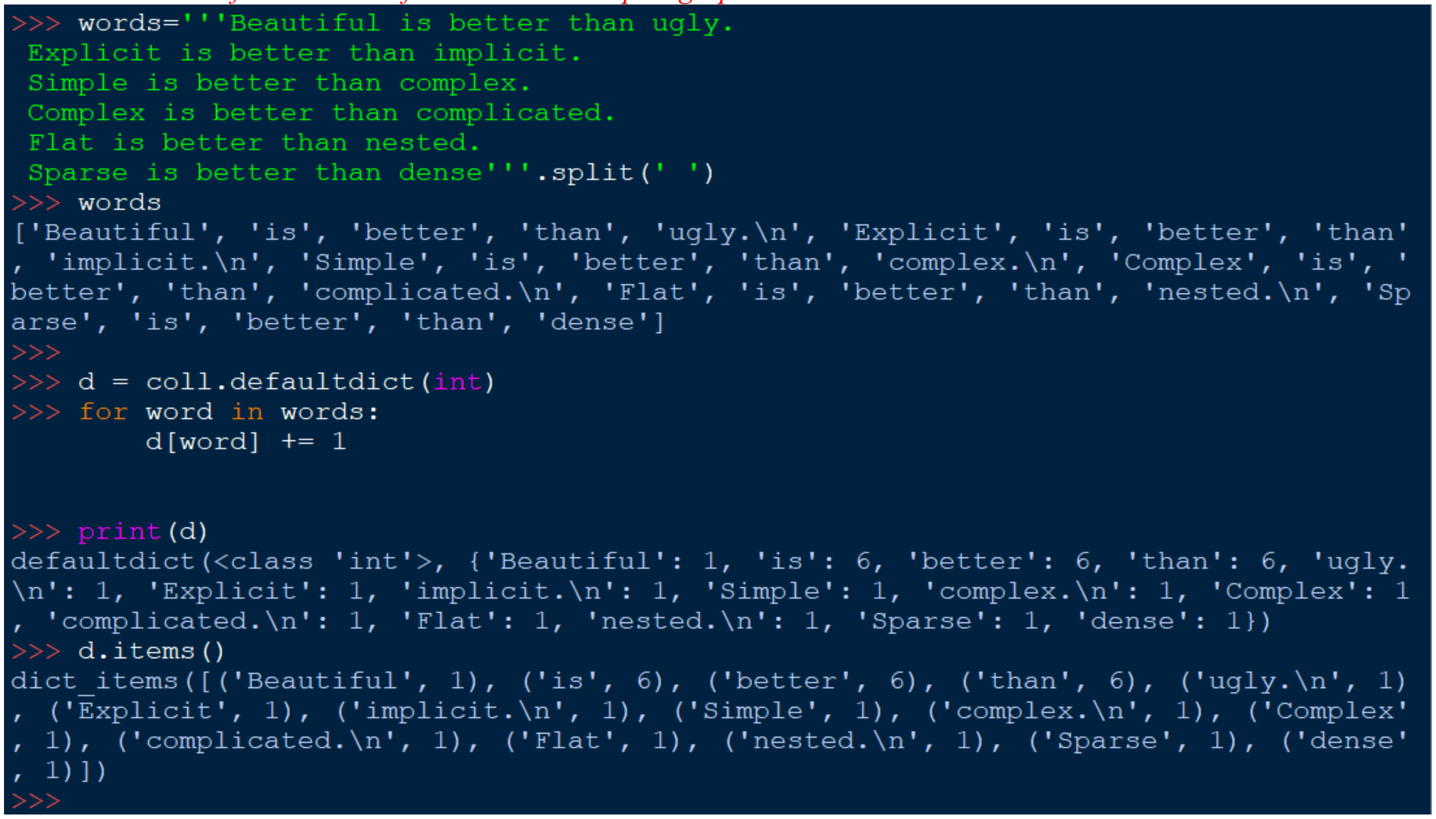


```
>>>
>>> counter_one = coll.Counter('superfluous')
>>> counter_one
Counter({'u': 3, 's': 2, 'p': 1, 'e': 1, 'r': 1, 'f': 1, 'l': 1, 'o': 1})
>>>
>>> counter_two = coll.Counter('super')
>>> counter_two
Counter({'s': 1, 'u': 1, 'p': 1, 'e': 1, 'r': 1})
>>>
>>> counter_one.subtract(counter_two)
>>> counter_one
Counter({'u': 2, 's': 1, 'f': 1, 'l': 1, 'o': 1, 'p': 0, 'e': 0, 'r': 0})
>>>
```

defaultdict

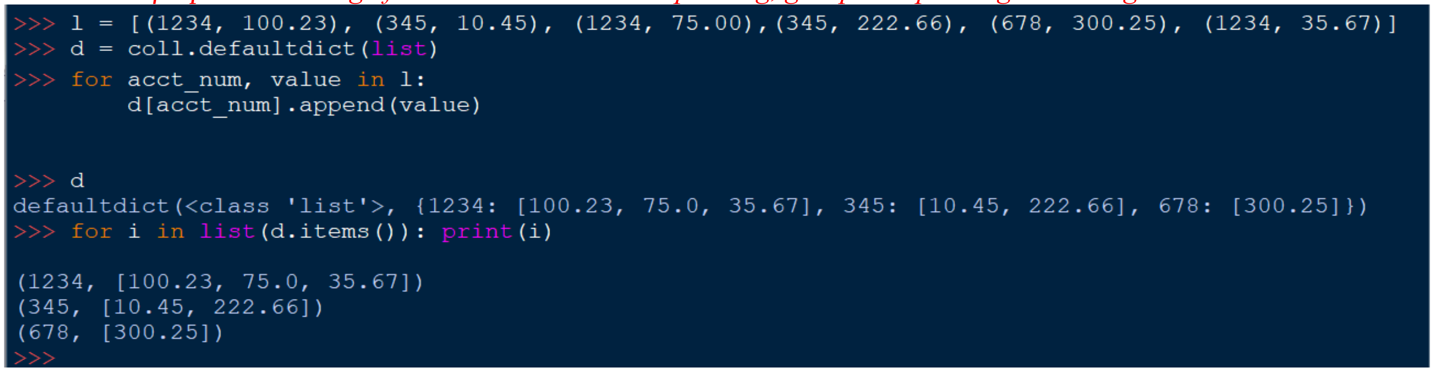
The defaultdict is a subclass of Python's **dict** that accepts a default_factory as its primary argument. The default_factory is usually a Python type, such as **int** or **list**, but you can also use a **function** or a **lambda** too.

Count the number of occurances of each word in the paragraph…….



```
>>> words='''Beautiful is better than ugly.
 Explicit is better than implicit.
 Simple is better than complex.
 Complex is better than complicated.
 Flat is better than nested.
 Sparse is better than dense'''.split(' ')
>>> words
['Beautiful', 'is', 'better', 'than', 'ugly.\n', 'Explicit', 'is', 'better', 'than'
, 'implicit.\n', 'Simple', 'is', 'better', 'than', 'complex.\n', 'Complex', 'is', '
better', 'than', 'complicated.\n', 'Flat', 'is', 'better', 'than', 'nested.\n', 'Sp
arse', 'is', 'better', 'than', 'dense']
>>>
>>> d = coll.defaultdict(int)
>>> for word in words:
        d[word] += 1


>>> print(d)
defaultdict(<class 'int'>, {'Beautiful': 1, 'is': 6, 'better': 6, 'than': 6, 'ugly.
\n': 1, 'Explicit': 1, 'implicit.\n': 1, 'Simple': 1, 'complex.\n': 1, 'Complex': 1
, 'complicated.\n': 1, 'Flat': 1, 'nested.\n': 1, 'Sparse': 1, 'dense': 1})
>>> d.items()
dict_items([('Beautiful', 1), ('is', 6), ('better', 6), ('than', 6), ('ugly.\n', 1)
, ('Explicit', 1), ('implicit.\n', 1), ('Simple', 1), ('complex.\n', 1), ('Complex'
, 1), ('complicated.\n', 1), ('Flat', 1), ('nested.\n', 1), ('Sparse', 1), ('dense'
, 1)])
>>>
```

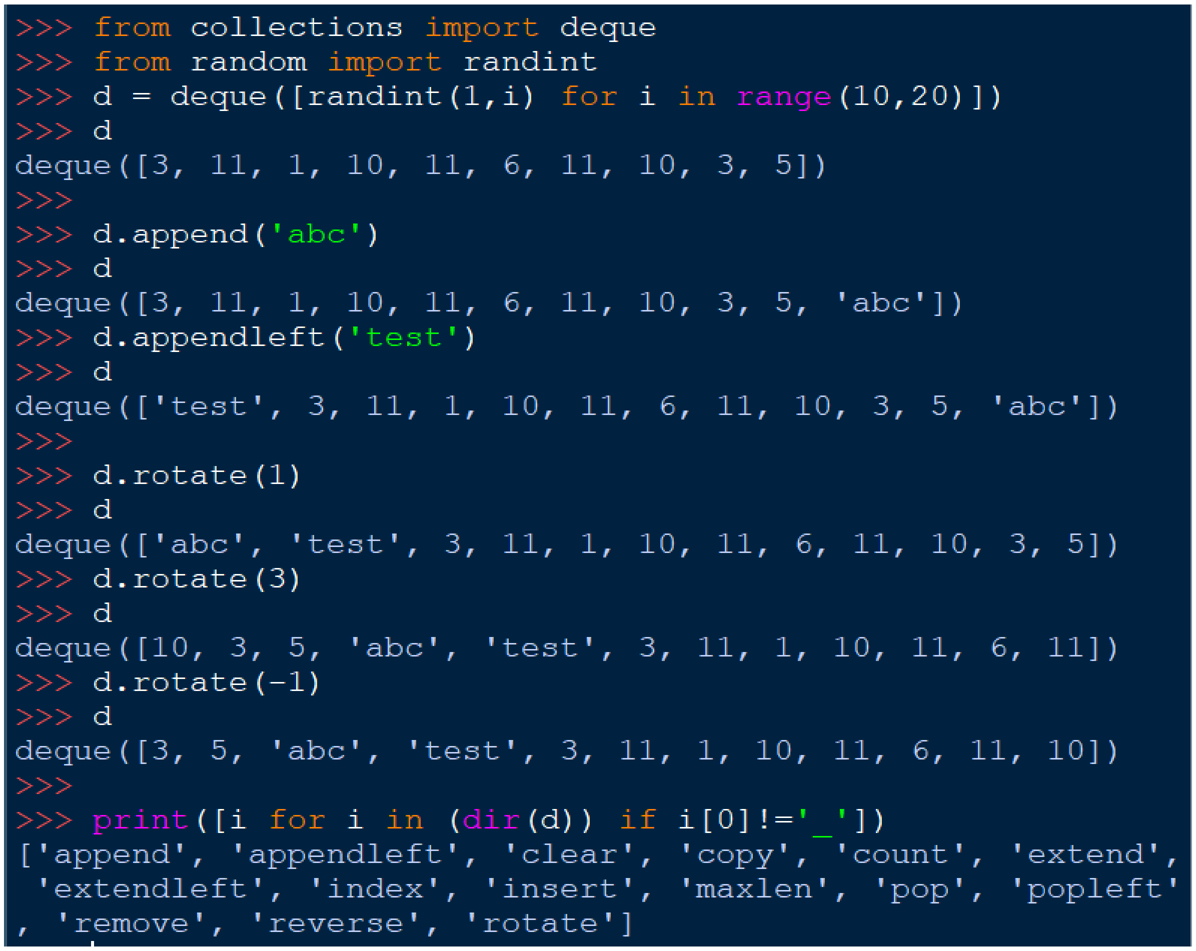
From a list of tuples consisting of Account numbers and spending, group the spending according to account numbers…



```
>>> l = [(1234, 100.23), (345, 10.45), (1234, 75.00),(345, 222.66), (678, 300.25), (1234, 35.67)]
>>> d = coll.defaultdict(list)
>>> for acct_num, value in l:
        d[acct_num].append(value)


>>> d
defaultdict(<class 'list'>, {1234: [100.23, 75.0, 35.67], 345: [10.45, 222.66], 678: [300.25]})
>>> for i in list(d.items()): print(i)

(1234, [100.23, 75.0, 35.67])
(345, [10.45, 222.66])
(678, [300.25])
>>>
```

deque

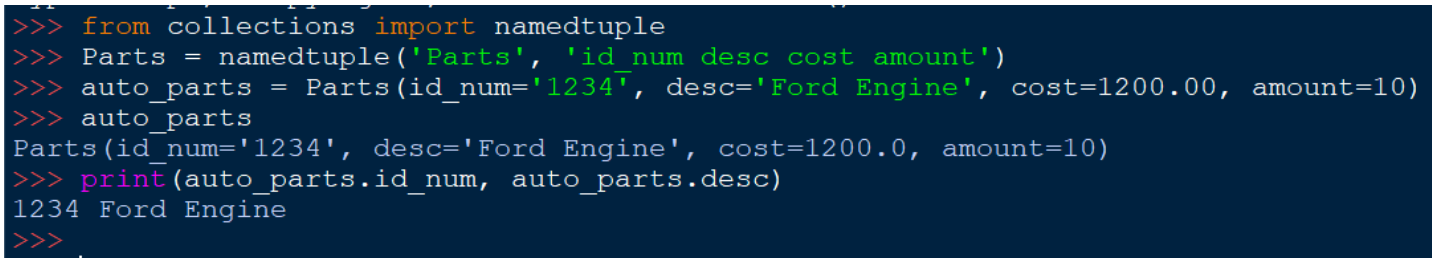
According to the Python documentation, **deques** “are a generalization of stacks and queues”. They are pronounced “deck” which is short for “double-ended queue”. They are a replacement container for the Python **list**. Deques are thread-safe and support memory efficient appends and pops from either side of the deque. A list is optimized for fast fixed-length operations. You can get all the details in the Python documentation. A deque accepts a **maxlen** argument which sets the bounds for the deque. Otherwise the deque will grow to an arbitrary size. When a bounded deque is full, any new items added will cause the same number of items to be popped off the other end.
As a general rule, if you need fast appends or fast pops, use a deque. If you need fast random access, use a list.



```
>>> from collections import deque
>>> from random import randint
>>> d = deque([randint(1,i) for i in range(10,20)])
>>> d
deque([3, 11, 1, 10, 11, 6, 11, 10, 3, 5])
>>>
>>> d.append('abc')
>>> d
deque([3, 11, 1, 10, 11, 6, 11, 10, 3, 5, 'abc'])
>>> d.appendleft('test')
>>> d
deque(['test', 3, 11, 1, 10, 11, 6, 11, 10, 3, 5, 'abc'])
>>>
>>> d.rotate(1)
>>> d
deque(['abc', 'test', 3, 11, 1, 10, 11, 6, 11, 10, 3, 5])
>>> d.rotate(3)
>>> d
deque([10, 3, 5, 'abc', 'test', 3, 11, 1, 10, 11, 6, 11])
>>> d.rotate(-1)
>>> d
deque([3, 5, 'abc', 'test', 3, 11, 1, 10, 11, 6, 11, 10])
>>>
>>> print([i for i in (dir(d)) if i[0]!='_'])
['append', 'appendleft', 'clear', 'copy', 'count', 'extend',
 'extendleft', 'index', 'insert', 'maxlen', 'pop', 'popleft'
, 'remove', 'reverse', 'rotate']
```

namedtuple

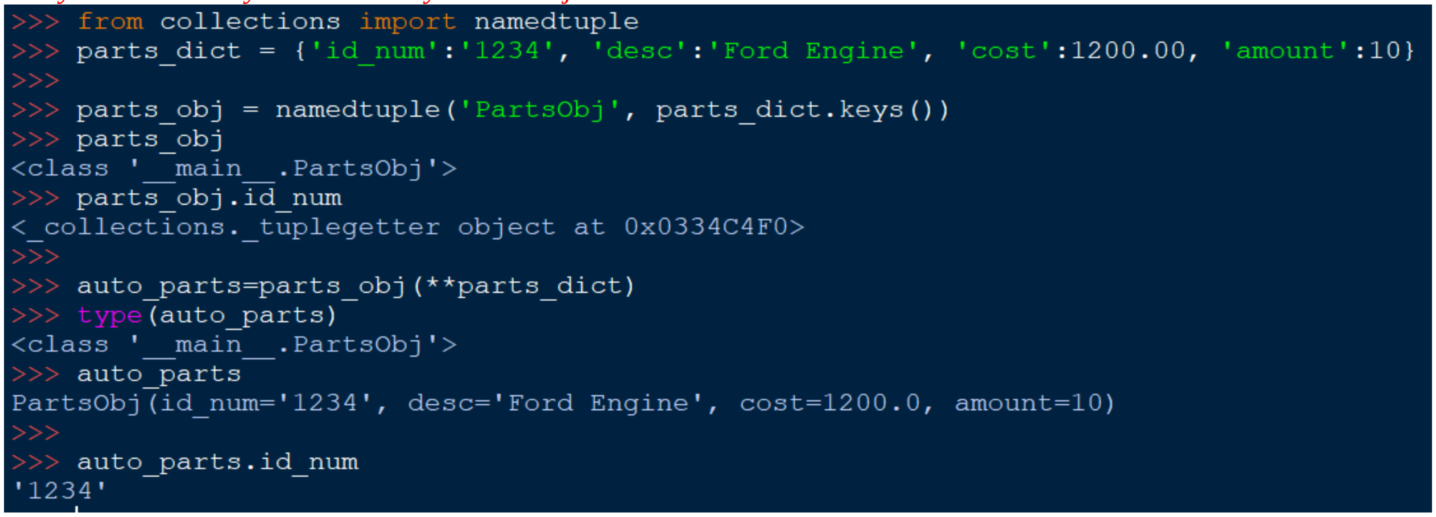
The **namedtuple** can be used to replace Python’s **tuple**. Of course, the namedtuple is not a drop-in replacement. It can be used like a **struct**. A *struct* is basically a complex data type that groups a list of variables under one name.



```
>>> from collections import namedtuple
>>> Parts = namedtuple('Parts', 'id_num desc cost amount')
>>> auto_parts = Parts(id_num='1234', desc='Ford Engine', cost=1200.00, amount=10)
>>> auto_parts
Parts(id_num='1234', desc='Ford Engine', cost=1200.0, amount=10)
>>> print(auto_parts.id_num, auto_parts.desc)
1234 Ford Engine
>>>
```

Here we import **namedtuple** from the **collections** module. Then we called namedtuple, which will return a new subclass of a tuple but with named fields. So basically we just created a new tuple class.
you will note that we have a strange string as our second argument. This is a space delimited list of properties that we want to create.
Now that we have our shiny new class, let’s create an instance of it! As you can see above, we do that as our very next step when we create the **auto_parts** object. Now we can access the various items in our auto_parts using dot notation because they are now properties of our Parts class.
One of the benefits of using a namedtuple over a regular tuple is that you no longer have to keep track of each item’s index because now each item is named and accessed via a class property.

A way to convert a Python dictionary into an object



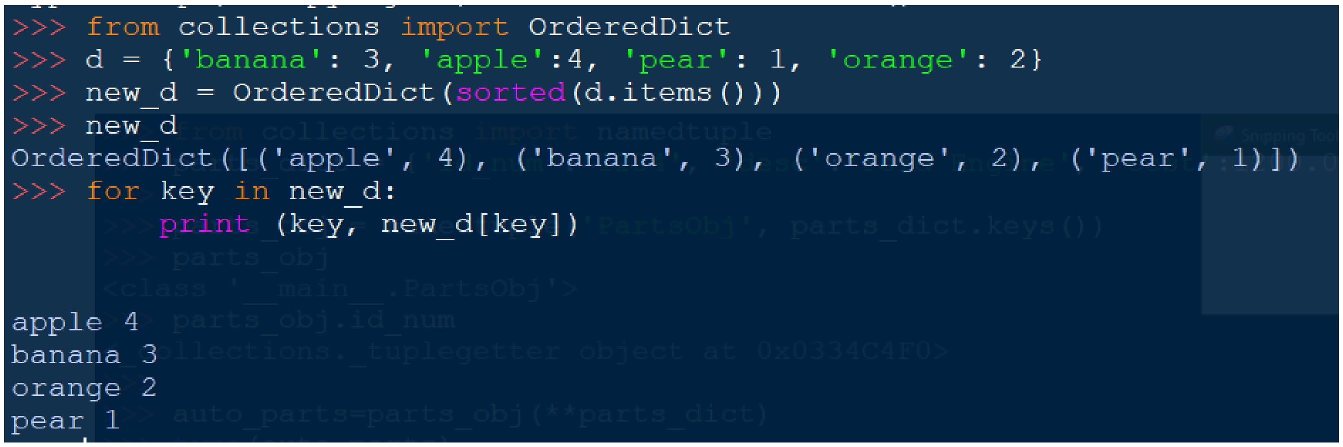
```
>>> from collections import namedtuple
>>> parts_dict = {'id_num':'1234', 'desc':'Ford Engine', 'cost':1200.00, 'amount':10}
>>>
>>> parts_obj = namedtuple('PartsObj', parts_dict.keys())
>>> parts_obj
<class '__main__.PartsObj'>
>>> parts_obj.id_num
<_collections._tuplegetter object at 0x0334C4F0>
>>>
>>> auto_parts=parts_obj(**parts_dict)
>>> type(auto_parts)
<class '__main__.PartsObj'>
>>> auto_parts
PartsObj(id_num='1234', desc='Ford Engine', cost=1200.0, amount=10)
>>>
>>> auto_parts.id_num
'1234'
```

OrderedDict

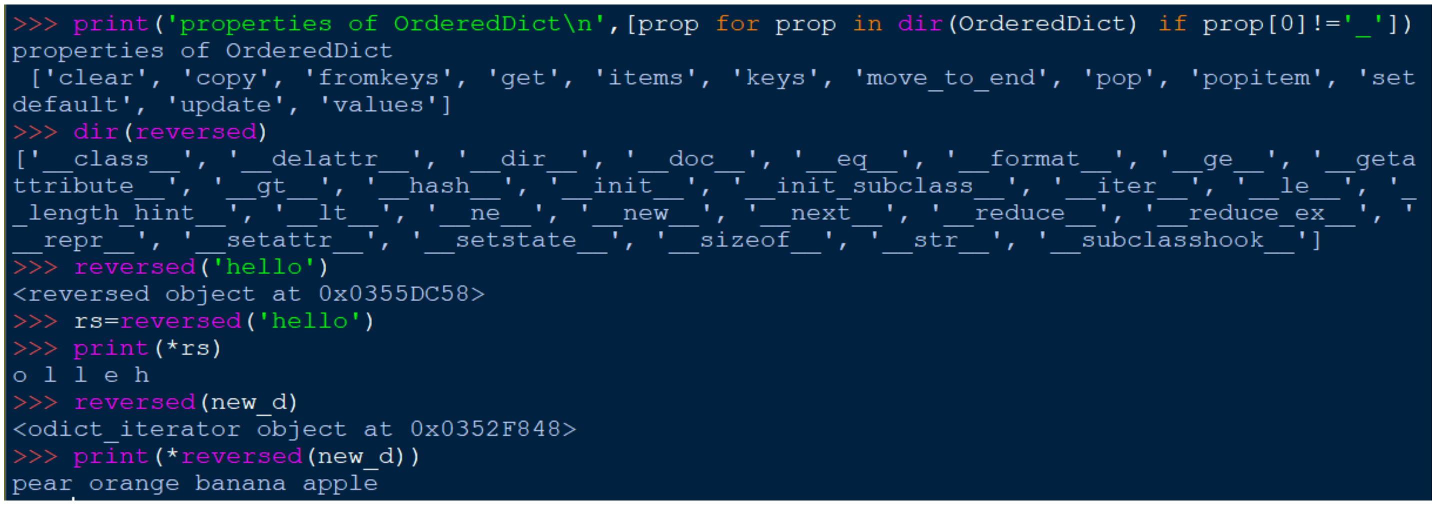
Python's collections module has another great subclass of **dict** known as **OrderedDict**. As the name implies, this dictionary keeps track of the order of the keys as they are added. If you create a regular **dict**, you will note that it is an unordered data collection. Every time you print it out, the order may be different. There are times when you will need to loop over the keys of your dictionary in a specific order. For example, I needed the keys sorted so I could loop over them in order. To do that, you can do the following:

```
>>> d = {'banana': 3, 'apple':4, 'pear': 1, 'orange': 2}
>>> d
{'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}
>>> for key in sorted(d.keys()):
            print (key, d[key])

apple 4
banana 3
orange 2
pear 1
>>>
```

Let's create an instance of an **OrderedDict** using our original **dict**, but during the creation, we'll sort the dictionary's keys:



```
>>> from collections import OrderedDict
>>> d = {'banana': 3, 'apple':4, 'pear': 1, 'orange': 2}
>>> new_d = OrderedDict(sorted(d.items()))
>>> new_d
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])
>>> for key in new_d:
        print (key, new_d[key])


apple 4
banana 3
orange 2
pear 1
```



```
>>> print('properties of OrderedDict\n',[prop for prop in dir(OrderedDict) if prop[0]!='_'])
properties of OrderedDict
 ['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'move_to_end', 'pop', 'popitem', 'set
default', 'update', 'values']
>>> dir(reversed)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__geta
ttribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '_
_length_hint__', '__lt__', '__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__', '
__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__']
>>> reversed('hello')
<reversed object at 0x0355DC58>
>>> rs=reversed('hello')
>>> print(*rs)
o l l e h
>>> reversed(new_d)
<odict_iterator object at 0x0352F848>
>>> print(*reversed(new_d))
pear orange banana apple
```