

Python 201

Intermediate Python



Python 201

Intermediate Python

Michael Driscoll

This book is for sale at <http://leanpub.com/python201>

This version was published on 2018-01-30



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2016 - 2018 Michael Driscoll

Tweet This Book!

Please help Michael Driscoll by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#python201](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#python201](#)

Also By **Michael Driscoll**

Python 101

Contents

Chapter 1 - An Intro to Argument Parsing using argparse	1
Getting Started	1
Adding Arguments	2
Short Options and Long Options	4
Options that Conflict	4
Wrapping Up	5
 Chapter 2 - The collections module	 6
ChainMap	6
Counter	8
defaultdict	9
deque	13
namedtuple	15
OrderedDict	16
Wrapping Up	18
 Chapter 3 - Context Managers	 19
Creating a Context Manager class	19
Creating a Context Manager using contextlib	20
contextlib.closing(thing)	21
contextlib.suppress(*exceptions)	22
contextlib.redirect_stdout / redirect_stderr	23
ExitStack	23
Reentrant Context Managers	24
Wrapping Up	25

Chapter 1 - An Intro to Argument Parsing using argparse

Have you ever wondered how to process command line arguments in Python? Yeah, there's a module for that. It's called argparse, which is a replacement for optparse. In this article, we'll be taking a whirlwind tour of this helpful module. Let's start with something simple!

Getting Started

I have always found the simplest way to explain a coding concept is to show some code. So that's what we're going to do. Here's a super simple example that doesn't do much of anything:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(
...     description="A simple argument parser",
...     epilog="This is where you might put example usage"
... )
...
>>> parser.print_help()
usage: _sandbox.py [-h]
```

A simple argument parser

optional arguments:

-h, --help show this help message and exit

This is where you might put example usage

Here we just import argparse and give it a description and set up a usage section. The idea here is that when you ask the program you are creating for help, it will tell you how to use it. In this case, it prints out a simple description, the default optional arguments ("-h" in this case) and example usage.

Now let's make this example a bit more concrete. You won't normally be parsing arguments from the command-line after all. So we'll move the code into a Python function inside of a Python file:

```

1  # arg_demo.py
2
3  import argparse
4
5
6  def get_args():
7      """
8      parser = argparse.ArgumentParser(
9          description="A simple argument parser",
10         epilog="This is where you might put example usage"
11     )
12     return parser.parse_args()
13
14 if __name__ == '__main__':
15     get_args()

```

Now let's call the script from the command line:

```
1 python arg_demo.py -h
```

This will print out the help text like we saw earlier. Now let's learn about how to add some of our own custom arguments.

Adding Arguments

Let's write some code that adds three new arguments that our parser can understand. We'll add an argument that is required and two that are not. We'll also look at adding a default and a required type. Here's the code:

```

1  # arg_demo2.py
2
3  import argparse
4
5
6  def get_args():
7      """
8      parser = argparse.ArgumentParser(
9          description="A simple argument parser",
10         epilog="This is where you might put example usage"
11     )

```

```

12
13     # required argument
14     parser.add_argument('-x', action="store", required=True,
15                         help='Help text for option X')
16     # optional arguments
17     parser.add_argument('-y', help='Help text for option Y', default=False)
18     parser.add_argument('-z', help='Help text for option Z', type=int)
19     print(parser.parse_args())
20
21 if __name__ == '__main__':
22     get_args()

```

Now let's run it a few times so you can see what happens:

```

1  mike@pc:~/py/argsparsing$ python arg_demo2.py
2  usage: arg_demo2.py [-h] -x X [-y Y] [-z Z]
3  arg_demo2.py: error: argument -x is required
4
5  mike@pc:~/py/argsparsing$ python arg_demo2.py -x something
6  Namespace(x='something', y=False, z=None)
7
8  mike@pc:~/py/argsparsing$ python arg_demo2.py -x something -y text
9  Namespace(x='something', y='text', z=None)
10
11 mike@pc:~/py/argsparsing$ python arg_demo2.py -x something -z text
12 usage: arg_demo2.py [-h] -x X [-y Y] [-z Z]
13 arg_demo2.py: error: argument -z: invalid int value: 'text'
14
15 mike@pc:~/py/argsparsing$ python arg_demo2.py -x something -z 10
16 Namespace(x='something', y=False, z=10)

```

As you can see, if you run the code without passing it any arguments, you will get an error. Next we pass it just the required argument so you can see what the defaults are for the other two. Then we try passing “text” to the ‘-y’ argument and that gets stored, so we know it doesn’t require a Boolean. The last two examples show what happens when you pass an invalid and a valid value to the ‘-z’ argument.

By the way, the argument names do not have to be one character in length. You can change those something more descriptive, like ‘arg1’ or ‘simulator’ or whatever you want.

Short Options and Long Options

Let's take a look at how we might use a short option versus a long a one. We actually are already using the short option here:

```
1 parser.add_argument('-x', action="store", required=True,
2                     help='Help text for option X')
```

If we wanted a long option, then we'd just need to add it right after the short one. Here's an example:

```
1 parser.add_argument('-x', '--execute', action="store", required=True,
2                     help='Help text for option X')
```

You will note that a long option is more than one character in length and that it must start with two dashes instead of one.

Options that Conflict

What do you do if you have options that conflict with each other? A common example would be running your application in verbose mode versus quiet mode. You can run it in either mode, but not both. How do we prevent the user from running it that way though? It's actually quite easy via the **mutually_exclusive_group** function. Let's pretend that options x and y cannot run at the same time and modify our code accordingly:

```
1 import argparse
2
3
4 def get_args():
5     """
6     parser = argparse.ArgumentParser(
7         description="A simple argument parser",
8         epilog="This is where you might put example usage"
9     )
10
11     group = parser.add_mutually_exclusive_group()
12     group.add_argument('-x', '--execute', action="store",
13                       help='Help text for option X')
14     group.add_argument('-y', help='Help text for option Y', default=False)
15
16     parser.add_argument('-z', help='Help text for option Z', type=int)
```

```

17     print(parser.parse_args())
18
19 if __name__ == '__main__':
20     get_args()

```

You will note that we have to create a mutually exclusive group. Then we add the options that need to be mutually exclusive to that group. The rest go into the regular parser group. Let's try running the code with both options like this:

```
1 python arg_demo3.py -x 10 -y 2
```

When I did this command, I ended up getting the following output:

```

1 usage: arg_demo3.py [-h] [-x EXECUTE | -y Y] [-z Z]
2 arg_demo2.py: error: argument -y: not allowed with argument -x/--execute

```

Obviously that didn't work and the argparse module told us why.

Wrapping Up

You now know how to create the basics of an argument parser. There are many other aspects of this module that you might be interested in, such as defining an alternate destination name for the argument to be saved to, using different prefixes (i.e. '+' instead of '-'), creating argument groups and more. I recommend checking out the documentation for more details.

Chapter 2 - The collections module

Python's collections module has specialized container datatypes that can be used to replace Python's general purpose containers (dict, tuple, list, and set). We will be studying the following parts of this fun module:

- ChainMap
- defaultdict
- deque
- namedtuple
- OrderedDict

There is a sub-module of collections called abc or Abstract Base Classes. These will not be covered in this chapter.

Let's get started with the ChainMap container!

ChainMap

A **ChainMap** is a class that provides the ability to link multiple mappings together such that they end up being a single unit. If you look at the documentation, you will notice that it accepts ***maps**, which means that a ChainMap will accept any number of mappings or dictionaries and turn them into a single view that you can update. Let's look at an example so you can see how this works:

```
>>> from collections import ChainMap
>>> car_parts = {'hood': 500, 'engine': 5000, 'front_door': 750}
>>> car_options = {'A/C': 1000, 'Turbo': 2500, 'rollbar': 300}
>>> car_accessories = {'cover': 100, 'hood_ornament': 150, 'seat_cover': 99}
>>> car_pricing = ChainMap(car_accessories, car_options, car_parts)
>>> car_pricing['hood']
500
```

Here we import **ChainMap** from our collections module. Next we create three dictionaries. Then we create an instance of our ChainMap by passing in the three dictionaries that we just created.

Finally, we try accessing one of the keys in our ChainMap. When we do this, the ChainMap will go through each map in order to see if that key exists and has a value. If it does, then the ChainMap will return the first value it finds that matches that key.

This is especially useful if you want to set up defaults. Let's pretend that we want to create an application that has some defaults. The application will also be aware of the operating system's environment variables. If there is an environment variable that matches one of the keys that we are defaulting to in our application, the environment will override our default. Let's further pretend that we can pass arguments to our application. These arguments take precedence over the environment and the defaults. This is one place where a ChainMap can really shine. Let's look at a simple example that's based on one from Python's documentation:

```

1  import argparse
2  import os
3
4  from collections import ChainMap
5
6
7  def main():
8      app_defaults = {'username': 'admin', 'password': 'admin'}
9
10     parser = argparse.ArgumentParser()
11     parser.add_argument('-u', '--username')
12     parser.add_argument('-p', '--password')
13     args = parser.parse_args()
14     command_line_arguments = {key:value for key, value
15                               in vars(args).items() if value}
16
17     chain = ChainMap(command_line_arguments, os.environ,
18                     app_defaults)
19     print(chain['username'])
20
21 if __name__ == '__main__':
22     main()
23     os.environ['username'] = 'test'
24     main()

```

Let's break this down a little. Here we import Python's **argparse** module along with the **os** module. We also import ChainMap. Next we have a simple function that has some silly defaults. I've seen these defaults used for some popular routers. Then we set up our argument parser and tell it how to handle certain command line options. You will notice that argparse doesn't provide a way to get a dictionary object of its arguments, so we use a dict comprehension to extract what we need. The other cool piece here is the use of Python's built-in **vars**. If you were to call it without an argument,

vars would behave like Python's built-in **locals**. But if you do pass in an object, then vars is the equivalent to object's `__dict__` property.

In other words, `vars(args)` equals `args.__dict__`. Finally create our ChainMap by passing in our command line arguments (if there are any), then the environment variables and finally the defaults. At the end of the code, we try calling our function, then setting an environment variable and calling it again. Give it a try and you'll see that it prints out **admin** and then **test** as expected. Now let's try calling the script with a command line argument:

```
1 python chain_map.py -u mike
```

When I ran this, I got **mike** back twice. This is because our command line argument overrides everything else. It doesn't matter that we set the environment because our ChainMap will look at the command line arguments first before anything else.

Now that you know how to use ChainMaps, we can move on to the Counter!

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. Let's try it out with a string!

```
>>> from collections import Counter
>>> Counter('superfluous')
Counter({'u': 3, 's': 2, 'e': 1, 'l': 1, 'f': 1, 'o': 1, 'r': 1, 'p': 1})
>>> counter = Counter('superfluous')
>>> counter['u']
3
```

In this example, we import **Counter** from collections and then pass it a string. This returns a Counter object that is a subclass of Python's dictionary. We then run the same command but assign it to the variable **counter** so we can access the dictionary a bit easier. In this case, we saw that the letter "u" occurs three times in the example string.

The Counter provides a few methods that might interest you. For example, you can call **elements** which will an iterator over the elements that are in the dictionary, but in an arbitrary order. You can kind of think of this function as a "scrambler" as the output in this case is a scrambled version of the string.

```
>>> list(counter.elements())
['e', 'l', 'f', 'o', 'r', 's', 's', 'p', 'u', 'u', 'u']
```

Another useful method is **most_common**. You can ask the Counter what the most common items are by passing in a number that represents what the top recurring “n” items are:

```
>>> counter.most_common(2)
[('u', 3), ('s', 2)]
```

Here we just ask our Counter what the top two most recurring items were. As you can see, it produced a list of tuples that tells us “u” occurred 3 times and “s” occurred twice.

The other method that I want to cover is the **subtract** method. The subtract method accepts an iterable or a mapping and the uses that argument to subtract. It’s a bit easier to explain if you see some code:

```
>>> counter_one = Counter('superfluous')
>>> counter_one
Counter({'u': 3, 's': 2, 'e': 1, 'l': 1, 'f': 1, 'o': 1, 'r': 1, 'p': 1})
>>> counter_two = Counter('super')
>>> counter_one.subtract(counter_two)
None
>>> counter_one
Counter({'u': 2, 'l': 1, 'f': 1, 'o': 1, 's': 1, 'e': 0, 'r': 0, 'p': 0})
```

So here we recreate our first counter and print it out so we know what’s in it. That we create our second Counter object. Finally we subtract the second counter from the first. If you look carefully at the output at the end, you will notice the that number of letters for five of the items has been decremented by one.

As I mentioned at the beginning of this section, you can use the Counter against any iterable or mapping, so you don’t have to just use strings. You can also pass it tuples, dictionaries and lists! Give it a try on your own to see how it works with those other data types.

Now we’re ready to move on to the defaultdict!

defaultdict

The collections module has a handy tool called **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. Let’s start by creating a regular Python dictionary that counts the number of times each word is used in a sentence:


```

1 sentence = "The red for jumped over the fence and ran to the zoo for food"
2 words = sentence.split(' ')
3
4 reg_dict = {}
5 for word in words:
6     if word in reg_dict:
7         reg_dict[word] += 1
8     else:
9         reg_dict[word] = 1
10
11 print(reg_dict)

```

If you run this code, you should see output that is similar to the following:

```

1 {'The': 1,
2  'and': 1,
3  'fence': 1,
4  'food': 1,
5  'for': 2,
6  'jumped': 1,
7  'over': 1,
8  'ran': 1,
9  'red': 1,
10 'the': 2,
11 'to': 1,
12 'zoo': 1}

```

Now let's try doing the same thing with defaultdict!

```

1 from collections import defaultdict
2
3
4 sentence = "The red for jumped over the fence and ran to the zoo for food"
5 words = sentence.split(' ')
6
7 d = defaultdict(int)
8 for word in words:
9     d[word] += 1
10
11 print(d)

```

You will notice right away that the code is much simpler. The defaultdict will automatically assign zero as the value to any key it doesn't already have in it. We add one so it makes more sense and it will also increment if the word appears multiple times in the sentence.

```

1 defaultdict(<class 'int'>,
2             {'The': 1,
3              'and': 1,
4              'fence': 1,
5              'food': 1,
6              'for': 2,
7              'jumped': 1,
8              'over': 1,
9              'ran': 1,
10             'red': 1,
11             'the': 2,
12             'to': 1,
13             'zoo': 1})

```

Now let's try using a Python list type as our default factory. We'll start off with a regular dictionary first, as before.

```

1 my_list = [(1234, 100.23), (345, 10.45), (1234, 75.00),
2            (345, 222.66), (678, 300.25), (1234, 35.67)]
3
4 reg_dict = {}
5 for acct_num, value in my_list:
6     if acct_num in reg_dict:
7         reg_dict[acct_num].append(value)
8     else:
9         reg_dict[acct_num] = [value]
10
11 print(reg_dict)

```

This example is based on some code I wrote a few years ago. Basically I was reading a file line by line and needed to grab the account number and the payment amount and keep track of them. Then at the end, I would sum up each account. We're skipping the summing part here. If You run this code, you should get some output similar to the following:

```

1 {345: [10.45, 222.66], 678: [300.25], 1234: [100.23, 75.0, 35.67]}

```

Now let's re-implement this code using defaultdict:

```

1  from collections import defaultdict
2
3
4  my_list = [(1234, 100.23), (345, 10.45), (1234, 75.00),
5             (345, 222.66), (678, 300.25), (1234, 35.67)]
6
7  d = defaultdict(list)
8  for acct_num, value in my_list:
9      d[acct_num].append(value)
10
11 print(d)

```

Once again, this cuts out the if/else conditional logic and makes the code easier to follow. Here's the output from the code above:

```

1  defaultdict(<class 'list'>,
2             {345: [10.45, 222.66],
3              678: [300.25],
4              1234: [100.23, 75.0, 35.67]})

```

This is some pretty cool stuff! Let's go ahead and try using a lambda too as our default_factory!

```

>>> from collections import defaultdict
>>> animal = defaultdict(lambda: "Monkey")
>>> animal['Sam'] = 'Tiger'
>>> print (animal['Nick'])
Monkey
>>> animal
defaultdict(<function <lambda> at 0x7f32f26da8c0>, {'Nick': 'Monkey', 'Sam': 'Ti\
ger'})

```

Here we create a defaultdict that will assign 'Monkey' as the default value to any key. The first key we set to 'Tiger', then the next key we don't set at all. If you print the second key, you will see that it got assigned 'Monkey'. In case you haven't noticed yet, it's basically impossible to cause a KeyError to happen as long as you set the default_factory to something that makes sense. The documentation does mention that if you happen to set the default_factory to None, then you will receive a KeyError. Let's see how that works:

```
>>> from collections import defaultdict
>>> x = defaultdict(None)
>>> x['Mike']
Traceback (most recent call last):
  Python Shell, prompt 41, line 1
KeyError: 'Mike'
```

In this case, we just created a very broken defaultdict. It can no longer assign a default to our key, so it throws a `KeyError` instead. Of course, since it is a subclass of `dict`, we can just set the key to some value and it will work. But that kind of defeats the purpose of the defaultdict.

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 gory 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. Let’s take a few moments to look at how you might create and use a deque.

```
>>> from collections import deque
>>> import string
>>> d = deque(string.ascii_lowercase)
>>> for letter in d:
...     print(letter)
```

Here we import the deque from our collections module and we also import the **string** module. To actually create an instance of a deque, we need to pass it an iterable. In this case, we passed it **string.ascii_lowercase**, which returns a list of all the lower case letters in the alphabet. Finally, we loop over our deque and print out each item. Now let’s look at a few of the methods that deque possesses.

```

>>> d.append('bork')
>>> d
deque(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
      'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'bork'])
>>> d.appendleft('test')
>>> d
deque(['test', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
      'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'bork'])
>>> d.rotate(1)
>>> d
deque(['bork', 'test', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l\
',
      'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'])

```

Let's break this down a bit. First we append a string to the right end of the deque. Then we append another string to the left side of the deque.. Lastly, we call **rotate** on our deque and pass it a one, which causes it to rotate one time to the right. In other words, it causes one item to rotate off the right end and onto the front. You can pass it a negative number to make the deque rotate to the left instead.

Let's finish out this section by looking at an example that's based on something from the Python documentation:

```

1  from collections import deque
2
3
4  def get_last(filename, n=5):
5      """
6      Returns the last n lines from the file
7      """
8      try:
9          with open(filename) as f:
10             return deque(f, n)
11     except OSError:
12         print("Error opening file: {}".format(filename))
13         raise

```

This code works in much the same way as Linux's **tail** program does. Here we pass in a filename to our script along with the n number of lines we want returned. The deque is bounded to whatever number we pass in as n. This means that once the deque is full, when new lines are read in and added to the deque, older lines are popped off the other end and discarded. I also wrapped the file opening **with** statement in a simple exception handler because it's really easy to pass in a malformed path. This will catch files that don't exist for example.

Now we're ready to move on and learn about the namedtuple.

namedtuple

The one that we'll be focusing on in this section is the **namedtuple** which you can use to replace Python's **tuple**. Of course, the namedtuple is not a drop-in replacement as you will soon see. I have seen some programmers use it like a struct. If you haven't used a language with a struct in it, then that needs a little explanation. A struct is basically a complex data type that groups a list of variables under one name. Let's look at an example of how to create a namedtuple so you can see how they work:

```
1 from collections import namedtuple
2
3 Parts = namedtuple('Parts', 'id_num desc cost amount')
4 auto_parts = Parts(id_num='1234', desc='Ford Engine',
5                   cost=1200.00, amount=10)
6 print(auto_parts.id_num)
```

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. Here's the difference in code:

```
>>> auto_parts = ('1234', 'Ford Engine', 1200.00, 10)
>>> auto_parts[2] # access the cost
1200.0
>>> id_num, desc, cost, amount = auto_parts
>>> id_num
'1234'
```

In the code above, we create a regular tuple and access the cost of the vehicle engine by telling Python the appropriate index we want. Alternatively, we can also extract everything from the tuple using multiple assignment. Personally, I prefer the namedtuple approach just because it fits the mind easier and you can use Python's **dir()** method to inspect the tuple and find out its properties. Give that a try and see what happens!

The other day I was looking for a way to convert a Python dictionary into an object and I came across some code that did something like this:

```
>>> from collections import namedtuple

>>> Parts = {'id_num': '1234', 'desc': 'Ford Engine',
            'cost': 1200.00, 'amount': 10}
>>> parts = namedtuple('Parts', Parts.keys())(**Parts)
>>> parts
Parts(amount=10, cost=1200.0, id_num='1234', desc='Ford Engine')
```

This is some weird code, so let's take it a piece at a time. The first line we import namedtuple as before. Next we create a Parts dictionary. So far, so good. Now we're ready for the weird part. Here we create our namedtuple class and name it 'Parts'. The second argument is a list of the keys from our dictionary. The last piece is this strange piece of code: (Parts)**. The double asterisk means that we are calling our class using keyword arguments, which in this case is our dictionary. We could split this line into two parts to make it a little clearer:

```
>>> parts = namedtuple('Parts', Parts.keys())
>>> parts
<class '__main__.Parts'>
>>> auto_parts = parts(**Parts)
>>> auto_parts
Parts(amount=10, cost=1200.0, id_num='1234', desc='Ford Engine')
```

So here we do the same thing as before, except that we create the class first, then we call the class with our dictionary to create an object. The only other piece I want to mention is that namedtuple also accepts a **verbose** argument and a **rename** argument. The verbose argument is a flag that will print out class definition right before it's built if you set it to True. The rename argument is useful if you're creating your namedtuple from a database or some other system that your program doesn't control as it will automatically rename the properties for you.

At this point you should be familiar enough with the namedtuple to use it yourself, so let's check out the OrderedDict!

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:

```
>>> d = {'banana': 3, 'apple':4, 'pear': 1, 'orange': 2}
>>> d
{'apple': 4, 'banana': 3, 'orange': 2, 'pear': 1}
```

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 have had a use case where I needed the keys sorted so I could loop over them in order. To do that, you can do the following:

```
>>> keys = d.keys()
>>> keys
dict_keys(['apple', 'orange', 'banana', 'pear'])
>>> keys = sorted(keys)
['apple', 'banana', 'orange', 'pear']
>>> for key in 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
```

Here we create our OrderedDict by sorting it on the fly using Python's sorted built-in function. The sorted function takes in the dictionary's items, which is a list of tuples that represent the key pairs of the dictionary. It sorts them and then passes them into the OrderedDict, which will retain their order. Thus when we go to print our the keys and values, they are in the order we expect. If you were to loop over a regular dictionary (not a sorted list of keys), the order would change all the time.

Note that if you add new keys, they will be added to the end of the `OrderedDict` instead of being automatically sorted.

Something else to note about `OrderDicts` is that when you go to compare two `OrderedDicts`, they will not only test the items for equality, but also that the order is correct. A regular dictionary only looks at the contents of the dictionary and doesn't care about its order.

Finally, `OrderDicts` have two new methods in Python 3: `popitem` and `move_to_end`. The `popitem` method will return and remove a (key, item) pair. The `move_to_end` method will move an existing key to either end of the `OrderedDict`. The item will be moved right end if the last argument for `OrderedDict` is set to `True` (which is the default), or to the beginning if it is `False`.

Interestingly, `OrderedDicts` support reverse iteration using Python's reversed built-in function:

```
>>> for key in reversed(new_d):  
...     print (key, new_d[key])  
...  
pear 1  
orange 2  
banana 3  
apple 4
```

Pretty neat, although you probably won't be needing that functionality every day.

Wrapping Up

We've covered a lot of ground in this chapter. You learned how to use a `defaultdict` and an `OrderedDict`. We also learned about a neat subclass of Python's list, the deque. Finally we looked at how to use a `namedtuple` to do various activities. I hope you found each of these collections interesting. They may be of great use to you in your own coding life.

Chapter 3 - Context Managers

Python came out with a special new keyword several years ago in Python 2.5 that is known as the **with statement**. This new keyword allows a developer to create context managers. But wait! What's a context manager? They are handy constructs that allow you to set something up and tear something down automatically. For example, you might want to open a file, write a bunch of stuff to it and then close it. This is probably the classic example of a context manager. In fact, Python creates one automatically for you when you open a file using the **with statement**:

```
1 with open(path, 'w') as f_obj:
2     f_obj.write(some_data)
```

Back in Python 2.4, you would have to do it the old fashioned way:

```
1 f_obj = open(path, 'w')
2 f_obj.write(some_data)
3 f_obj.close()
```

The way this works under the covers is by using some of Python's magic methods: **__enter__** and **__exit__**. Let's try creating our own context manager to demonstrate how this all works!

Creating a Context Manager class

Rather than rewrite Python's open method here, we'll create a context manager that can create a SQLite database connection and close it when it's done. Here's a simple example:

```

1  import sqlite3
2
3
4  class DataConn:
5      """
6
7      def __init__(self, db_name):
8          """Constructor"""
9          self.db_name = db_name
10
11     def __enter__(self):
12         """
13         Open the database connection
14         """
15         self.conn = sqlite3.connect(self.db_name)
16         return self.conn
17
18     def __exit__(self, exc_type, exc_val, exc_tb):
19         """
20         Close the connection
21         """
22         self.conn.close()
23         if exc_val:
24             raise
25
26 if __name__ == '__main__':
27     db = '/home/mdriscoll/test.db'
28     with DataConn(db) as conn:
29         cursor = conn.cursor()

```

In the code above, we created a class that takes a path to a SQLite database file. The `__enter__`-method executes automatically where it creates and returns the database connection object. Now that we have that, we can create a cursor and write to the database or query it. When we exit the with statement, it causes the `__exit__` method to execute and that closes the connection.

Let's try creating a context manager using another method.

Creating a Context Manager using contextlib

Python 2.5 not only added the with statement, but it also added the contextlib module. This allows us to create a context manager using contextlib's contextmanager function as a decorator. Let's try creating a context manager that opens and closes a file after all:

```

1  from contextlib import contextmanager
2
3  @contextmanager
4  def file_open(path):
5      try:
6          f_obj = open(path, 'w')
7          yield f_obj
8      except OSError:
9          print("We had an error!")
10     finally:
11         print('Closing file')
12         f_obj.close()
13
14 if __name__ == '__main__':
15     with file_open('/home/mdriscoll/test.txt') as fobj:
16         fobj.write('Testing context managers')
```

Here we just import **contextmanager** from **contextlib** and decorate our `file_open` function with it. This allows us to call `file_open` using Python's `with` statement. In our function, we open the file and then yield it out so the calling function can use it.

Once the **with** statement ends, control returns back to the `file_open` function and it continues with the code following the `yield` statement. That causes the `finally` statement to execute, which closes the file. If we happen to have an **OSError** while working with the file, it gets caught and `finally` statement still closes the file handler.

contextlib.closing(thing)

The `contextlib` module comes with some other handy utilities. The first one is the **closing** class which will close the thing upon the completion of code block. The Python documentation gives an example that's similar to the following one:

```

1  from contextlib import contextmanager
2
3  @contextmanager
4  def closing(db):
5      try:
6          yield db.conn()
7      finally:
8          db.close()
```

Basically what we're doing is creating a closing function that's wrapped in a contextmanager. This is the equivalent of what the closing class does. The difference is that instead of a decorator, we can use the **closing** class itself in our with statement. Let's take a look:

```

1  from contextlib import closing
2  from urllib.request import urlopen
3
4  with closing(urlopen('http://www.google.com')) as webpage:
5      for line in webpage:
6          # process the line
7          pass

```

In this example, we open a url page but wrap it with our closing class. This will cause the handle to the web page to be closed once we fall out of the with statement's code block.

contextlib.suppress(*exceptions)

Another handy little tool is the **suppress** class which was added in Python 3.4. The idea behind this context manager utility is that it can suppress any number of exceptions. Let's say we want to ignore the **FileNotFoundError** exception. If you were to write the following context manager, it wouldn't work:

```

>>> with open('fauxfile.txt') as fobj:
        for line in fobj:
            print(line)

```

Traceback (most recent call last):

```

Python Shell, prompt 4, line 1
builtins.FileNotFoundError: [Errno 2] No such file or directory: 'fauxfile.txt'

```

As you can see, this context manager doesn't handle this exception. If you want to ignore this error, then you can do the following:

```

1  from contextlib import suppress
2
3  with suppress(FileNotFoundError):
4      with open('fauxfile.txt') as fobj:
5          for line in fobj:
6              print(line)

```

Here we import **suppress** and pass it the exception that we want to ignore, which in this case is the **FileNotFoundError** exception. If you run this code, you will note that nothing happens as the file does not exist, but an error is also not raised. It should be noted that this context manager is **reentrant**. This will be explained later on in this section.

contextlib.redirect_stdout / redirect_stderr

The contextlib library has a couple of neat tools for redirecting stdout and stderr that were added in Python 3.4 and 3.5 respectively. Before these tools were added, if you wanted to redirect stdout, you would do something like this:

```
1 path = '/path/to/text.txt'
2
3 with open(path, 'w') as fobj:
4     sys.stdout = fobj
5     help(sum)
```

With the contextlib module, you can now do the following:

```
1 from contextlib import redirect_stdout
2
3 path = '/path/to/text.txt'
4 with open(path, 'w') as fobj:
5     with redirect_stdout(fobj):
6         help(redirect_stdout)
```

In both of these examples, we are redirecting stdout to a file. When we call Python's **help**, instead of printing to stdout, it gets saved directly to the file. You could also redirect stdout to some kind of buffer or a text control type widget from a user interface toolkit like Tkinter or wxPython.

ExitStack

ExitStack is a context manager that will allow you to easily programmatically combine other context managers and cleanup functions. It sounds kind of confusing at first, so let's take a look at an example from the Python documentation to help us understand this idea a bit better:

```
>>> from contextlib import ExitStack
>>> with ExitStack() as stack:
>>>     file_objects = [stack.enter_context(open(filename))
>>>                     for filename in filenames]
>>> ]
```

This code basically creates a series of context managers inside the list comprehension. The ExitStack maintains a stack of registered callbacks that it will call in reverse order when the instance it closed, which happens when we exit the the bottom of the **with** statement.

There are a bunch of neat examples in the Python documentation for contextlib where you can learn about topics like the following:

- Catching exceptions from `__enter__` methods
- Supports a variable number of context managers
- Replacing any use of try-finally
- and much more!

I highly recommend checking it out so you get a good feel for how powerful this class is.

Reentrant Context Managers

Most context managers that you create will be written such that they can only be used once using a **with** statement. Here's a simple example:

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def single():
...     print('Yielding')
...     yield
...     print('Exiting context manager')
>>> context = single()
>>> with context:
...     pass
...
Yielding
Exiting context manager
>>> with context:
...     pass
...
Traceback (most recent call last):
  Python Shell, prompt 9, line 1
    File "/usr/local/lib/python3.5/contextlib.py", line 61, in __enter__
        raise RuntimeError("generator didn't yield") from None
builtins.RuntimeError: generator didn't yield
```

Here we create an instance of our context manager and try running it twice with Python's `with` statement. The second time it runs, it raises a **RuntimeError**.

But what if we wanted to be able to run the context manager twice? Well we'd need to use one that is "reentrant". Let's use the `redirect_stdout` context manager that we used before!

```

>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print('Write something to the stream')
...     with write_to_stream:
...         print('Write something else to stream')
...
>>> print(stream.getvalue())
Write something to the stream
Write something else to stream

```

Here we create a nested context manager where they both write to **StringIO**, which is an in-memory text stream. The reason this works instead of raising a `RuntimeError` like before is that `redirect_stdout` is reentrant and allows us to call it twice. Of course, a real world example would be much more complex with more functions calling each other. Please also note that reentrant context managers are not necessarily thread-safe. Read the documentation before trying to use them in a thread.

Wrapping Up

Context managers are a lot of fun and come in handy all the time. I use them in my automated tests all the time for opening and closing dialogs, for example. Now you should be able to use some of Python's built-in tools to create your own context managers. Be sure to take the time to read the Python documentation on `contextlib` as there are lots of additional information that is not covered in this chapter. Have fun and happy coding!